An Extremely Flexible Programming Language

Christian Heinlein

Hochschule Aalen (University of Applied Sciences)
Aalen, Baden-Württemberg, Germany
christian.heinlein@hs-aalen.de

September 2023

Abstract. MOSTflexiPL is a general-purpose programming language, whose syntax can be freely extended and customized by every programmer. Based on a small set of predefined operators, it is possible to define new operators with arbitrary syntax, which do not only cover prefix, infix, and postfix operators, but also control structures, type constructors, and declaration forms. The paper gives an overview of major concepts of MOSTflexiPL in a tutorial-like manner by giving numerous examples from everyday programming.

1 Introduction

MOSTflexiPL, which is an acronym for **mo**dular, **s**tatically **t**yped, **fl**exibly **ex**tens**i**ble **p**rogramming **l**anguage, is a general-purpose programming language, whose syntax can be freely extended and customized by every programmer. The logo used in the paper title shall express the extreme flexibility provided by the language allowing even fancy constructions unimaginable with conventional languages. (Therefore, it might be advisable to forget almost all familiar and seemingly necessary limitations of other languages to be able to fully recognize MOSTflexiPL's capabilities.)

A basic principle enabling that flexibility is: Everything is an expression, i. e., the application of an operator to subexpressions, where operators might possess any number of names and operands in an arbitrary order. Apart from well-known prefix, infix, and postfix operators, this also includes "circumfix" operators such as `(•)` (an operand depicted by the bullet sign enclosed in parentheses), control structures such as `if• then•else•end`, declaration forms such as `•:•` (a name and a type separated by a colon), and so on. Another basic principle is, that the language provides only a small set of predefined operators covering arithmetic and logic operations as well as basic control structures, which can be used to define arbitrary new operators.

As the name indicates, the language is statically typed – which imposes numerous challenges with respect to the already mentioned flexibility –, and it is currently implemented by a compiler and a run-time system written in C++.

The primary goal of this paper is to give a broad overview of MOSTflexiPL's major concepts in a tutorial-like manner by showing numerous examples taken from every-

day programming. The examples also demonstrate, that writing syntactic extensions in MOSTflexiPL is just as easy as writing normal code, which is a significant difference and advantage over other approaches to syntactic extensibility.

2 Simple Operator Declarations

To give a first example, the following simple declarations define operators computing the square and the absolute value, respectively, of an integer value `x`, which can afterwards be applied using well-known mathematical syntax, e. g., 5^2 or $|2-7|^2$:

```
(x:int) "²" -> (int = x * x);
"|" (x:int) "|" -> (int = if x > 0 then x else -x end)
```

An operator declaration generally consists of a *signature*, an arrow and a *result declaration*, where the signature is a sequence of *names* and *parameter declarations*, while the result declaration consists of a *type* (the result type of the operator), an equality sign, and the *implementation* of the operator, enclosed in parentheses. A name is either a sequence of letters and digits starting with a letter (denoting exactly this sequence of characters) or a sequence of arbitrary characters enclosed in quotation marks (denoting this sequence of characters without the quotation marks). A parameter declaration consists of a name, a colon, and a type, enclosed in parentheses. Finally, the implementation and the types mentioned above are – according to the basic principle mentioned in Sec. 1 – expressions. (At the moment, types are atomic expressions such as `int` or `bool`, but see Sec. 4 and Sec. 7 for more complex type expressions.)

When an operator application such as $|2-7|^2$ is evaluated at run time, the parameters of the operator are initialized from left to right by recursively evaluating the corresponding operands and then the value of the expression is determined by evaluating the implementation of the operator.

In the examples above, the implementation of the square operator uses the predefined multiplication operator `•*•`, while the implementation of the abs operator uses the predefined change sign operator `-•` as well as the conditional operator `if•then•else•end` that returns, according to the truth value of its first operand, either the value of its second or its third operand.

The semicolon used to separate the two operator declarations is a simple predefined infix operator that evaluates its left and right operand and returns the value of the latter and therefore is typically used to denote sequential execution of subexpressions. But – again according to the basic principle mentioned in Sec. 1 – since declarations are expressions, too, the semicolon is also used to separate multiple declarations. (Sec. 8 explains the precise meaning of the result value of an operator declaration.) In contrast to many other programming languages, however, semicolon must not be used at the end of a sequence of subexpressions, because it is an infix operator.

To give another example, the following declaration defines an operator that recursively computes the factorial of an integer value `n`, which can also be applied using

well-known mathematical syntax, e. g., `5!` or 5^2`!`:

```
(n:int) "!" -> (int = if n <= 1 then 1 else (n-1)! * n end)
```

The particular challenge for the compiler with a declaration like this is to already recognize and accept the new syntax defined by the declaration inside of its own implementation to allow recursive applications of the operator.

MOSTflexiPL does not provide a predefined syntax for function declarations and applications, because any desired syntax is actually covered by the general operator declaration syntax mentioned above. For example, the style used by many procedural languages with function applications of the form `max(2, 3)`:

```
max "(" (x:int) "," (y:int) ")"
                          -> (int = if x > y then x else y end)
```

Or the syntax of functional languages such as Haskell with function applications of the form `max 2 3`:

```
max (x:int) (y:int) -> (int = if x > y then x else y end)
```

Or even a more natural-language-like flavour with function applications of the form `max of 2 and 3`:

```
max of (x:int) and (y:int)
                          -> (int = if x > y then x else y end)
```

3 Exclude Declarations

The predefined operators of MOSTflexiPL obey common rules for precedence and associativity, e. g., multiplication and division bind stronger than addition and subtraction, and all of them are left-associative. In contrast, user-defined operators have no predefined precedence or associativity, which frequently leads to ambiguous expressions. For example, the expression `2 + 3`^{2} might not only have the intended meaning `2 + (3`^{2}`)`, but could also be interpreted by the compiler as `(2 + 3)`^{2} (where the parentheses shall only indicate the different groupings).

Exclude declarations can be used to resolve such ambiguities by specifying interpretations of expressions which are not intended, i. e., excluded, for example:

```
excl (2 + 3)^2 end
```

Here, `2` and `3` are arbitrary placeholders for integer operands. The effect of this declaration is that applications of the operator •+• (the operator at the top of the parenthesized subexpression) are excluded as operands of the operator •^{2}. Therefore, the expression `2 + 3`^{2} will now be unambiguously interpreted as the addition of `2` and the square of `3`, because the alternative interpretation as the square of the addition of `2` and `3` is now forbidden.

Because outside of exclude declarations, the predefined parentheses `(•)` are just a normal operator (that simply returns the value of its operand), they can still be used

for explicit grouping: In the expression `(2 + 3)²` with explicit parentheses around the addition, the operand of the operator `•²` is not an application of the operator `•+•`, but rather an application of the operator `(•)` (whose operand is an application of the operator `•+•`), which is not forbidden.

In general, the expression between `excl` and `end` can contain any number of parenthesized subexpressions, each of which is interpreted as described above. Therefore, the effect of the following exclude declaration is, that the square operator binds stronger than all basic arithmetic operators:

```
excl (1 + 2)²; (1 - 2)²; (1 * 2)²; (1 / 2)² end
```

To give another example, the following exclude declarations encode exactly the rules for precedence and associativity of the basic arithmetic operators that have been mentioned at the beginning of this section:

```
excl (1+2)*(3+4); (1-2)*(3-4); (1+2)/(3+4); (1-2)/(3-4) end;
excl 1*(2*3); 1*(2/3); 1/(2*3); 1/(2/3) end;
excl 1+(2+3); 1+(2-3); 1-(2+3); 1-(2-3) end
```

4 Constants and Variables

A declaration of the form *`name : type = init`* declares a constant with the given name and type whose value is obtained by evaluating the initializer expression *`init`*, e. g., `N : int = 5²`. If the type is omitted, e. g., `N := 5²`, it is automatically deduced from the type of the initializer. If the initializer is omitted, the constant receives a unique new value that is different from every other value of the type. While this is of limited usefulness for numeric types such as `int`, it is crucial for variable types described below and for user-defined types described later in Sec. 6.

For any type `T`, the type `T?` denotes memory cells containing values of type `T`. Therefore, a declaration such as `x : T?` defines `x` as a constant referring to a unique new memory cell that contains a value of type `T`, i. e., `x` actually denotes variable with content type `T`. The current value contained in such a variable can be queried with the prefix question mark operator `?•`, and it can be changed with the assignment operator `•=!•`. The initial value of a variable is nil, which is a predefined value for any type that is different from every other value of the type. Because a variable with content type `T` is itself a value of type `T?`, it might itself be stored in a variable of type `T??`. If the content of such a variable is queried prior to any assignment to the variable, the returned value is the nil value of type `T?`. If the content of this nil variable is queried in turn, it will be the nil value of type `T`, and assigning any value to such a nil variable has no effect. (This behaviour is roughly comparable to reading and writing the Unix special file `/dev/null`.)

By using variables and the predefined loop operator `while•do•end`, the factorial operator mentioned in Sec. 2 can also be implemented in a more procedural style:

```
(n:int) "!" -> (int =
    f : int?; f =! 1;
    i : int?; i =! 2;
    while ?i <= n do
        f =! ?f * ?i;
        i =! ?i + 1
    end;
    ?f
)
```

In the implementation of the operator, the variables `f` and `i` are declared and assigned their initial values as described above, where `i` is used as a loop counter running from `2` to `n`, while `f` accumulates the factorial value that is finally returned.

5 Optional, Alternative, and Repeatable Syntax Parts

To provide even more syntactic flexibility, the signature of an operator declaration might also contain optional, alternative, and repeatable parts using well-known EBNF syntax.

For example, the following declaration defines a variadic maximum operator that can be applied to any number of operands, e. g., `max of 1`, `max of 1 and 2`, `max of 1 and 2 and 3`, and so on:

```
max of (x:int) { and (y:int) } -> (int =
    m : int?; m =! x;
    { if y > ?m then m =! y end };
    ?m
)
```

According to EBNF, the curly brackets in the signature indicate that an application of this operator might contain the word `and` followed by an operand corresponding to the parameter `y` any number of times (zero or more). To access the different values of this parameter in the implementation of the operator, a corresponding curly bracket operator `{•}` is provided there, whose operand is repeatedly evaluated for every value of `y`. For the particular application `max of 1 and 2 and 3` this means, that the variable `m` declared in the implementation is initialized with the value of `x` (i. e., `1`), and then the `if` expression inside the curly brackets is evaluated in turn for `y` equal to `2` and to `3`, changing the value of the variable `m` to `2` and to `3`, respectively. Finally, the resulting value of `m` is returned.

To give another example, the following operator performs arbitrary calculations consisting of additions and subtractions, e. g., `calc minus 1 plus 2` or `calc 1 minus 2 plus 3`:

```
calc [minus] (x:int) { (plus|minus) (y:int) } -> (int =
    res : int?;
    res =! [-x | x];
```

```
    { res =! (?res + y | ?res - y) };
    ?res
)
```

In addition to the curly brackets denoting a repeatable part, the signature of this operator also contains square brackets denoting an optional part as well as round brackets containing two or more alternative parts separated by vertical bars. To find out in the implementation of the operator, whether the optional word `minus` after the word `calc` is present or not in a particular application of the operator, a corresponding square bracket operator `[•|•]` is provided, whose first or second operand, respectively, is evaluated accordingly. Similarly, a round bracket operator `(•|•)` with two operands corresponding to the round brackets with two alternatives in the signature is provided, whose first or second operand is evaluated according to whether the first or second alternative has been chosen in a particular application of the operator or – because in this example the round brackets are nested inside the curly brackets – in the respective pass through the curly brackets. The result value of a square or round bracket operator is the value of the operand that has been evaluated, while the result value of a curly bracket operator is the number of passes through these brackets. Taken together, these bracket operators allow the implementation of the operator to exactly determine the structure a particular operator application and to process the values of its operands in a rather concise manner.

Generally speaking, all three kinds of brackets can have any number of alternatives, except that round brackets must contain at least two, because round brackets with just one alternative are useless. Therefore, the corresponding bracket operators provided in the implementation of the operator have a corresponding number of operands separated by vertical bars, where the *i*-th operand is evaluated if the *i*-th alternative has been chosen in a particular operator application or pass through curly brackets. As an exception, an operator corresponding to square brackets has an additional optional operand, that is evaluated (if it is present) if none of the alternatives has been chosen.

Therefore, the `calc` operator could also be defined as follows:

```
calc [minus] (x:int) { plus (y:int) | minus (z:int) } -> (int =
    res : int?;
    res =! [-x | x];
    { res =! ?res + y | res =! ?res - z };
    ?res
)
```

The different kinds of brackets can be used any number of times in an operator signature, and they can be arbitrarily nested to describe rather complex syntactic constructs, for example:

```
dnf [na: not] (a:bool) { and [nb: not] (b:bool) }
    { or [nc: not] (c:bool) { and [nd: not] (d:bool) } } end
                                                  -> (bool =
    res : bool?;
```

```
        res =! [na: ~a | a];
        { res =! ?res & [nb: ~b | b] };
        {
            tmp : bool?;
            tmp =! [nc: ~c | c];
            { tmp =! ?tmp & [nd: ~d | d] };
            res =! ?res | ?tmp
        };
        ?res
    )
```

This operator can be used to express arbitrary logic expressions in disjunctive normal form (DNF), e. g., `dnf x or not u and v and w or y and not z end`, if `u`, ..., `z` denote values of type `bool`. The predefined operators of MOSTflexiPL for logic operations used in the implementation of the `dnf` operator are `~`• for negation, •`&`• for conjunction, and •`|`• for disjunction. To disambiguate the multiple square brackets in the signature and their corresponding bracket operators in the implementation, they are labeled with unique names and a colon after the opening bracket. While the curly brackets could be disambiguated in the same way, this is actually not necessary, because parameters defined inside of particular curly brackets are only visible in the operands of the corresponding bracket operator. Therefore, the bracket operator used first in the implementation must correspond to the first curly brackets in the signature, because parameter `b` is only visible in the operator corresponding to these brackets. For the same reason, the (outer) bracket operator used next in the implementation must correspond to the second (outer) brackets in the signature (due to the visibility of parameter `c`), while the bracket operator used inside of the former must correspond to the inner brackets in the signature (due to the visibility of parameter `d`).

6 Static Operators and User-Defined Data Structures

As already mentioned in Sec. 2, a constant declaration *`name : type`* defines a constant with the given name and type and a unique new value. Constants whose type is the predefined meta-type `type` denote unique new types, e. g.:

```
Color : type
```

Afterwards, any number of unique values or "objects" of such a type can also be defined as constants, e. g.:

```
red : Color;
blue : Color;
green : Color
```

As also mentioned in Sec. 2, constants of a variable type `T?` actually denote unique variables with content type `T`.

Additionaly, if the implementation of an operator (including the equality sign) is omitted, a unique new value of the result type is returned whenever the implementation would be evaluated.

If the arrow `->` in an operator declaration is replaced by a double arrow `=>`, the declared operator is a so-called *static operator*. In contrast to a normal operator defined with a single arrow (that is also called a *dynamic operator*), a static operator has a runtime memory to store the parameter and result values of all applications of the operator performed so far. If the parameter values of a particular application are equal to the corresponding values of an earlier application, the implementation of the operator is not evaluated again, but the result value stored from the earlier application is returned instead. Therefore, a static operator guarantees that applications to the same parameter values always return the same value.

While this could be used to automatically optimize operators with runtime-intensive implementations by means of memoization, this is in fact neither the primary goal nor the typical use of static operators. Instead, they can be used as follows to flexibly define data structures:

```
Point : type;
(p:Point) "@" x => (int?);
(p:Point) "@" y => (int?);

p1 : Point; p1@x =! 1; p1@y =! 2;
p2 : Point; p2@x =! 3; p2@y =! 4;

dx := ?p1@x - ?p2@x;
dy := ?p1@y - ?p2@y
```

According to the semantics of static operators and omitted operator implementations described above, the operator `•@x` (and likewise the operator `•@y`) guarantees, that applications to the same point object always return the same `int` variable, while applications to different point objects return different variables (that are also different from all other existing variables). Therefore, the variables returned by `p1@x` and `p1@y` can be used to store the `x` and `y` coordinates of `p1`, while the variables returned by `p2@x` and `p2@y` can be used to store the coordinates of `p2`. Therefore, the constants `dx` and `dy` defined at the end of the example denote the difference of the `x` and `y` coordinates, respectively, of the points `p1` and `p2` (i. e., both have the value `-2`).

Because it is always possible to add further operators like `•@x` and `•@y` later on, i. e., to modularly extend a type such as `Point` with new "attributes," these types are called *open types* [5]. Furthermore, it is possible that different objects of a type possess values for different subsets of attributes. According to the semantics of variables described in Sec. 4, querying the value of a missing attribute of an object simply returns nil.

7 Generic Operators

If an optional parameter appears in the type of another parameter of the same operator, its value can be automatically deduced from the type of the operand corresponding to the other parameter, and therefore, the former parameter is called a *deducible parameter*. This can be used to define generic operators similar to C++ templates and Java generics, for example:

```
[(T:type)] (x:T?) "<->" (y:T?) -> (T? =
    z := ?x; x =! ?y; y =! z; y
);

excl u : int?; u <-> (u <-> u) end;

v1 : int?; v1 =! 1;
v2 : int?; v2 =! 2;
v1 <-> v2
```

Because `v1` and `v2` both have type `int?`, `v1 <-> v2` is a correct application of the previously defined swap operator, where the parameters `x` and `y` are initialized with the explicit operands `v1` and `v2`, respectively, while the optional parameter `T` is implicitly initialized with the type `int` causing the type `int?` of the operands `v1` and `v2` to match the type `T?` of the corresponding parameters `x` and `y`. The implementation of this operator swaps the values contained in the variables `x` and `y` and returns the variable `y`. (The latter enables concatenated applications of the operator to a sequence of variables, e. g., `v1 <-> v2 <-> v3`, actually performing a leftward rotation of their values. To disambiguate expressions like that, the exclude declaration is necessary which makes the operator left-associative.)

Generic operators can also be used to generalize the idea of open types and attributes already introduced in Sec. 6:

```
(U:type) "-->" (V:type) => (type);
[(U:type) (V:type)] (u:U) "@" (a:U-->V) => (V?);

Point : type;
x : Point --> int;
y : Point --> int;

p1 : Point; p1@x =! 1; p1@y =! 2;
p2 : Point; p2@x =! 3; p2@y =! 4
```

For every pair of types `U` and `V`, `U-->V` is a unique type intended to represent attributes for type `U` with target type `V`. Therefore, the constants `x` and `y` represent attributes for type `Point` with target type `int`. Furthermore, for every combination of an object `u` of some type `U` and an attribute `a` for type `U` with some target type `V`, `u@a` is a unique variable with content type `V` that can be used to store the value of attribute `a` for object `u`. Therefore, expressions such as `p1@x` and `p2@y` have exactly

the same meaning as in Sec. 6, but the definition of the attributes `x` and `y` is much more convenient now if the operators `•-->•` and `•@•` are provided by a library.

To make things even more convenient, another operator `•.•` can be defined to simplify the querying of attribute values by allowing to write, e. g., `p1.x` instead of `?p1@x`:

```
[(U:type) (V:type)] (u:U) "." (a:U-->V) -> (V = ?u@a)
```

Furthermore, it is possible to define a more advanced generic operator to directly construct objects of an open type with a set of initial attribute values:

```
(U:type) "(" [(V1:type)] (a1:U-->V1) "=" (v1:V1)
        { "," [(V2:type)] (a2:U-->V2) "=" (v2:V2) } ")" -> (U =
    u : U;
    u@a1 =! v1;
    { u@a2 =! v2 };
    u
);

p1 := Point(x = 1, y = 2);
p2 := Point(x = 3, y = 4)
```

As a final improvement, the operator `•@•` can be redefined as follows to make it return nil instead of a unique new variable if either the object `u` or the attribute `a` is nil (note that a constant declaration returns the value of the constant and an `if` expression without an `else` part returns nil if the condition is not satisfied; the predefined operator `•=/•` tests for inequality):

```
[(U:type) (V:type)] (u:U) "@" (a:U-->V) => (V? =
    if u =/ nil & a =/ nil then
        v : V?
    end
)
```

The effect of this modification is that querying an attribute value of a nil object or the value of a nil attribute of any object always returns nil (because querying a nil variable returns nil) and that modifying such attributes has no effect (because modifying a nil variable has no effect). Without this modification, it would be possible to accidentally assign a value to an attribute of a nil object, that would afterwards be returned by querying this attribute of the nil object, for example:

```
Line : type;
beg : Line --> Point;
end : Line --> Point;

ln := Line(beg = Point(x = 1, y = 2));
ln.end@x =! 4;
ln.end.x
```

Because the `Line` object `ln` does not have a value for the attribute `end`, `ln.end` returns a nil point, whose attribute `x` is then (formally) assigned the value `4`. With the

original definition of the operator •@• given at the beginning of this section, ln.end@x would return a "real" variable that would store the assigned value 4. Therefore, ln.end.x would return this value, which is illogical because the line ln does not have a defined endpoint at all. With the modified definition of the operator •@• given above, ln.end@x returns a nil variable causing the assignment of the value 4 to have no effect, and therefore, ln.end.x also returns nil which is more logical.

The above operators can also be used to define generic open types such as lists using a Haskell-like syntax [|T|] to denote lists with element type T:

```
"[|" (T:type) "|]" => (type);
[(T:type)] head => ([|T|] --> T);
[(T:type)] tail => ([|T|] --> [|T|])
```

Note that the generic attributes head and tail are not defined as constants, because constants cannot have any parameters and therefore cannot be generic, but rather as static operators that return different values for different types T to make sure that list types with different element types have logically different attributes.

Furthermore, head and tail are remarkable because the value of their type parameter T cannot be deduced from their bare application (which is simply head or tail), but only from the context of such an application, e. g.:

```
ls1 : [|int|]; ls1@head =! 1;
ls2 : [|bool|]; ls2@head =! true
```

Because (i) the type of ls1 is [|int|], (ii) the operand types of the operator •@• must be U and U-->V for suitable types U and V, and (iii) the type of head must be [|T|]-->T for some suitable type T, the type of head in the expression ls1@head is uniquely deduced by the compiler as [|int|]-->int by solving this "constraint puzzle." Likewise, the type of head in the expression ls2@head will be [|bool|]-->bool.

Using the above definitions, the "cons" operator for constructing a list from a head element h and a tail list t can be defined as follows, again with a syntax •:• known from Haskell:

```
[(T:type)] (h:T) ":" [(t:[|T|])] -> ([|T|] =
    [|T|](head = h, tail = t)
);

excl (ls := 1) : end;

ls1 := 1 : 2 : 3 :;
ls2 := true : false :
```

As a convenient extension, the tail list can be omitted, because the parameter t is optional and therefore is automatically nil if the corresponding operand is missing in an application of the operator.

Note that it is not necessary to explicitly make the operator `•:•` right-associative by means of an exclude declaration, because an expression such as `1:2:3:` can only be interpreted as `1:(2:(3:))`, because any other syntactically possible interpretation (e. g., `((1:2):3):`) would not be type-correct. And in fact, the MOSTflexiPL compiler – in contrast to the compilers of many other programming languages – does not artificially separate the semantic analysis (i. e., type checking) from the syntactic analysis of the source code, but rather performs them together in close cooperation in order to rule out expressions which are not type-correct as early as possible.

On the other hand, the exclude declaration contained in the example is necessary to exclude an interpretation such as `(ls1:=1):2:3:`, which would in fact be type-correct.

8 Implicit Parameters

Another useful example of generic operators would be a generic maximum operator that can be applied to operands of any type `T`:

```
[(T:type)] max of (x:T) and (y:T) -> (T =
    if x > y then x else y end
);

max of 1 and 2;
max of p1 and p2
```

While the application of this operator to the `int` values `1` and `2` appears reasonable, its application to the points `p1` and `p2` does not make sense, because there is no operator `•>•` to compare points. And in fact, the compiler would already reject the above declaration of the maximum operator and not only its application to points, because there is no operator `•>•` that can be applied to the operands `x` and `y` of an arbitrary type `T` whose precise value is not known there.

To make the compiler accept the operator declaration, it is necessary to express that the operator might only be applied to operands of a type `T`, if there is an operator `•>•` which accepts two operands of that type `T` and which returns a value of type `bool`. This can be expressed with an *implicit parameter*:

```
[(T:type)] max of (x:T) and (y:T) [(+ (T) ">" (T) -> (bool))]
-> (T = if x > y then x else y end)
```

This requires explanations of some details which have not been mentioned yet:

- The name of a parameter including the subsequent colon can be omitted if it is not needed.
  Therefore, `(T) ">" (T) -> (bool)` is a correct operator declaration describing exactly the kind of operator that is required by the maximum operator.
- Furthermore, an operator declaration actually constitutes a type, i. e., the type of the declared operator, which includes the types of its parameters and its result as well as

its syntax.
Therefore, `((T) ">" (T) -> (bool))` is a correct declaration of an anonymous parameter whose type is the operator type `(T) ">" (T) -> (bool)`.

- Prefixing this parameter type with a plus sign marks the parameter as an implicit parameter of the maximum operator, which means that it is implicitly bound to an operator of the same type that is visible at the point where the maximum operator is applied. (Alternatively, it would be possible to pass an explicit operand of that type, cf. Sec. 9.)
  Therefore, applications of the parameter in the implementation of the maximum operator (i. e., `x > y`) will actually be forwarded to the operator that has been passed (either implicitly or explicitly).

For an application such as `max of 1 and 2` with operands of type `int` that means, that an operator with type `(int) ">" (int) -> (bool)`, i. e., the predefined •>• operator for integer values, is implicitly passed to the maximum operator and thus used in its implementation to compare the operands.

An application such as `max of p1 and p2` with operands of type `Point`, however, is rejected by the compiler, because there is no operator with type `(Point) ">" (Point) -> (bool)` that could be passed implicitly. This could be remedied, however, by defining such an operator before applications of the maximum operator to points:

```
(p1:Point) ">" (p2:Point) -> (bool = p1.x > p2.x)
```

Here, `p1` is considered greater than `p2` if the x coordinate of `p1` is greater than that of `p2`.

In fact, the operator that is implicitly passed for an implicit parameter is not required to have exactly the same type as the parameter. It is rather sufficient, that the parameter *can be replaced* by the operator according to the following definition: An operator or parameter can be replaced by another operator or parameter, if every correct application of the former is also a correct application of the latter with the same type.

For example, there is actually no predefined operator •>•, but rather a much more general comparison operator that also supports comparison chains of multiple operands such as `a > b = c >= d` with well-known semantics from mathematics. (In contrast to mathematical practice, however, it is even allowed to form "inconsistent" chains such as `a > b <= c`.) But because this operator can replace the implicit parameter according to the definition above, it can and will in fact be passed implicitly to applications of the maximum operator to integer operands.

Another implication of this replacement rule is, that an operator that has implicit parameters itself can be implicitly passed to an implicit parameter, if there are in turn matching operators for its own implicit parameters, and so on. For example:

```
[(T:type)] (x:T) "²" [((T) "*" (T) -> (T))] -> (T = x * x);
[(T:type)] (x:T) "⁴" [((T) "²" -> (T))] -> (T = x²²)
```

An application of the bisquare operator •⁴ to an integer value such as 5⁴ requires a square operator •² for integers, which in turn requires a multiplication operator •*• for integers, which is available as a predefined operator. Therefore, the expression 5⁴ is correct.

On the other hand, an application of the bisquare operator to a point would require a square operator for points, which would be available if there would be a multiplication operator for points, which is not the case, however. Therefore, an expression such as p1⁴ would be rejected by the compiler. Again, this could be remedied in principle by defining such a multiplication operator.

9 Higher-Order Operators

Operators with implicit parameters as described in the previous section are actually higher-oder operators, i. e., operators having parameters that are itself operators. This is not restricted to implicit parameters, however, but operators can also be passed explicitly to other operators.

To give a typical example from functional programming:

```
[(X:type) (Y:type)]
map (f: f (X) -> (Y)) (ls: [|X|]) -> ([|Y|] =
    if ls =/ nil then
        (f ls.head) : (map f (ls.tail))
    end
);

sq: (x:int) "²" -> (int = x * x);

sq: f (x:int) -> (int = x * x);
ls := (1 : 2 : 3 :);
map sq ls
```

According to the explanations given in Sec. 8, `(f: f (X) -> (Y))` is the declaration of a parameter with name `f` whose type is the operator type `f (X) -> (Y)`, i. e., `f` is used both as the name of the entire parameter and as the first name of the operator contained in its type. Therefore, `f ls.head` is an application of this operator to `ls.head`, while the `f` in `map f ls.tail` is used to pass this operator to the recursive invocation of `map`.

According to the same principle, `sq` is a constant whose type is the operator type given after the colon of the constant declaration and whose value is the operator contained in that type. That is, in fact, an exception to the rule given in Sec. 4, which must now be restated as follows: If the initializer in a constant declaration is omitted, the value of the constant is either the operator contained in its type – if this type is an operator type – or otherwise a unique new value as stated before. Therefore, `sq` can actually be used to refer to the square operator and to pass it to the `map` operator.

When an operator is passed explicitly, the requirement given in Sec. 8, that the operator must be able to replace the corresponding parameter, is relaxed in order to allow operators whose syntax is different from the syntax of the parameter as in the above example. (The operator's syntax is `•`[2], while the parameter's syntax is `f•`.) Instead, only the parameter and result types of the operator and the parameter must match, i. e., their names are completely ignored because they are not important. (The precise rules, which are currently developed in detail, are more complex, because if an operator has optional, alternative, or repeatable parts, at least some of its names might be important to disambiguate applications of this operator.)

Finally, it is also possible to return operators from other operators, for example:

```
add (y:int) -> (f (int) -> (int) =
    f: f (x:int) -> (int = x + y)
);

map (add 5) ls
```

Because the result type of the operator `add•` is an operator type `f (int) -> (int)`, its implementation must return an operator of that type. And because the type of the constant `f` is also an operator type — in fact, exactly the same operator type —, the value of this constant is, according to the restated rule above, the operator contained in that type. Finally, because a constant declaration returns the value of the constant, the implementation of the operator `add•` returns exactly this operator, which can then be passed, e. g., to an application of the map operator. Please note, that it is in fact necessary to declare that dummy constant (with an arbitrary name), because the operator declaration itself does not return the declared operator, but rather its type (cf. Sec. 8).

10 Outlook

The language MOSTflexiPL and its compiler are still under active development, and several useful features that have already been developed and prototypically implemented in older versions of the compiler, have not been integrated into the current compiler, including:

- "Call by expression" parameters, which are required to define control structures such as branches and loops, whose operands shall be evaluated conditionally or repeatedly.
- Import and export declarations, which are required to define user-defined scoping rules and locally confined syntax extensions.
- Virtual operators, which are required to define type aliases to abbreviate or abstract from complex types and to define declaration operators, i. e., to be able to extend even the syntax that is used to define new syntax.
- Dynamic redefinitions of operators [4], which allow amongst other things strictly modular extensions of existing code and thus support unanticipated software evolution.

- Basic operators for parallel execution and synchronization, which can be used to define more convenient and advanced constructs for parallel programming.
- User-defined literals, e. g., for types representing date and time values.
- User-defined whitespace and comments to allow any desired syntax for both block and line comments.
- More descriptive compiler messages in case of errors and ambiguities.
- Meta-operators, which are required to pass the values of a repeatable parameter to another operator accepting repeatable parameters.
  This is in fact a completely new idea that is still under development and has not been implemented in any of the compiler versions yet.

A feature that is already implemented in the current version of compiler, but has not been described in this paper, is type deduction: Basically, it is possible to omit almost all types from declarations as long as they can be deduced by the compiler, i. e., the types of constants (this has in fact been mentioned in Sec. 4), parameters, and results of operators.

11 Related Work

During the history of programming language development, the idea of an extensible programming language has appeared every now and then.

One of oldest and most well-known examples is Lisp [9] with its different dialects and flavors. Similar to MOSTflexiPL, Lisp does neither distinguish between operators and functions nor between predefined and user-defined operators/functions. By defining new functions – or macros, whose syntactic appearance is identical to that of functions – a programmer is actually extending the language all the time. Another parallel to MOSTflexiPL is the fact that language extensions are defined in the language itself, and that a very small language core is sufficient for that purpose. However, there are also essential differences: First of all, Lisp does not possess a static type system. Furthermore, Lisp expressions must always be parenthesized, which significantly restricts the possibilities for defining new syntax. Finally, MOSTflexiPL does not have a "procedural" macro engine, i. e., no user code will be executed at compile time in order to perform syntactic transformations. In summary, MOSTflexiPL has considerable advantages over Lisp (complete syntactic freedom and static type safety), while the deliberately omitted procedural macro facility has not been perceived as a major limitation yet.

Dylan [3] is a more modern language that has been strongly influenced by Lisp's ideas. It also supports syntactic extensibility in the language itself (actually in a rewrite macro system which is an integral part of the language). Even though the programmer has more freedom than with Lisp's simple s-expressions, there are also strict syntactic limitations which cannot be exceeded. In contrast, the operator concept of

MOSTflexiPL offers virtually unlimited syntactic freedom. Apart from that, Dylan does not have a static type system either.

Many different languages, e. g., Haskell [7], Prolog [2], and Scala [8], allow the user to extend at least the syntax of expressions by defining new operator symbols. Since functional languages, just as MOSTflexiPL, do not distinguish between expressions and statements, the syntax of statements (e. g., control structures) becomes also extensible in principle. However, the syntax of types and declarations still remains fixed. In MOSTflexiPL, however, the basic principle "*everything* is an expression" implies that *all* parts of the language can be extended simply by defining new operators.

An approach whose basic ideas and objectives are almost identical to that of MOSTflexiPL is "$\pi-$ a Pattern Language" [6]. The concept called pattern there – which is "the only language construct in π" – directly corresponds to an operator in MOSTflexiPL: It possesses a syntax, composed of names (or symbols) and placeholders for operands, and an associated meaning corresponding to the implementation of a MOSTflexiPL operator. Thus, both approaches provide the same virtually unlimited syntactic flexibility that ultimately stems from the lack of any predefined grammar.

A significant difference and advantage of MOSTflexiPL over π is once again the static type system, since π is completely dynamically typed. In fact, the endeavour to reconcile extreme flexibility on the one hand with a maximum of static checkability on the other hand has been and still is the most ambitious challenge in the development of MOSTflexiPL.

Apart from that, MOSTflexiPL provides several other useful facilities not found in π, e. g., implicit and deducible parameters (where the latter are dispensable in a dynamically typed language) or import, export, and exclude declarations which allow, amongst others, user-defined scoping rules and locally confined syntax extensions.

Finally, MOSTflexiPL might also be considered an adaptive grammar formalism [1, 10]. Because "everything is an expression," there is a single non-terminal symbol `X` denoting expressions. Every operator declaration induces a new production for `X` whose right hand side can be derived from the operator's signature by treating the operator's names as terminal symbols and replacing explicit parameters with the non-terminal `X`. The type information associated with the parameters and the result type of the operator can be added as grammar attributes. Import and export declarations control the set of currently active productions, while exclude declarations can be used to rule out some otherwise possible derivations.

12 Conclusion

MOSTflexiPL is a programming language currently under development whose syntax can be extended and customized by its users in a virtually unlimited way, where a rather small number of core constructs is sufficient to support a broad range of different programming styles. Therefore, it can be used, amongst others, as an extensible general purpose programming language, but also as a host language for developing

domain-specific languages. It possesses a static type system and is implemented by a compiler and a run-time system written in C++.

Acknowledgements

I want to thank all the students that have contributed to the development of MOST-flexiPL with their master's thesis, bachelor's thesis, or student's project (in chronological order): Marco Perazzo, Stefan Billet, Miriam Klement, Frank-Stephan Schierle, Rainer Ißler, Sascha Simon, Christian Homeyer, Dominik Biener, Tobias Sachon, Maximilian Blenk, Jakob Loskan, Niklas Brendle, Lukas Neubauer, David Sugar, Lukas Pietzschmann.

I also want to thank the colleagues at the Compilers and Languages Group of Jens Knoop at TU Wien for many fruitful discussions during my last sabbatical.

Finally, I want to thank my father in heaven for every challenge that I have been able to master with him.

References

[1] H. Christiansen: "A survey of adaptable grammars." *ACM SIGPLAN Notices* 25 (11) November 1990, 35–44.

[2] W. F. Clocksin, C. S. Mellish: *Programming in Prolog* (Fourth Edition). Springer-Verlag, Berlin, 1994.

[3] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

[4] C. Heinlein: "Global and Local Virtual Functions in C++." *Journal of Object Technology* 4 (10) December 2005, 71–93, http://www.jot.fm/issues/issue_2005_12/article4.

[5] C. Heinlein: "Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance." *Journal of Object Technology* 6 (3) March/April 2007, 101–151, http://www.jot.fm/issues/issue_2007_03/article3.

[6] R. Knöll, M. Mezini: "$\pi-$ a Pattern Language." In: *Proc. 24th Ann. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)* (Orlando, FL, October 2009). *ACM SIGPLAN Notices* 44 (10) October 2009, 503–521.

[7] S. Marlow (ed.): *Haskell 2010 Language Report*. HaskellWiki, 2010.
http://haskell.org/definition/haskell2010.pdf

[8] M. Odersky: *The Scala Language Specification* (Version 2.9). Programming Methods Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), May 2011.

[9] G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.

[10] Wikipedia Contributors: *Adaptive Grammar*.
https://en.wikipedia.org/wiki/Adaptive_grammar (2023-08-30)