The MNL: A Block-Based Functional Programming Language with Reactive Blocks

Steven Lolong
University of Tübingen
Tübingen, Germany
steven.lolong@uni-tuebingen.de

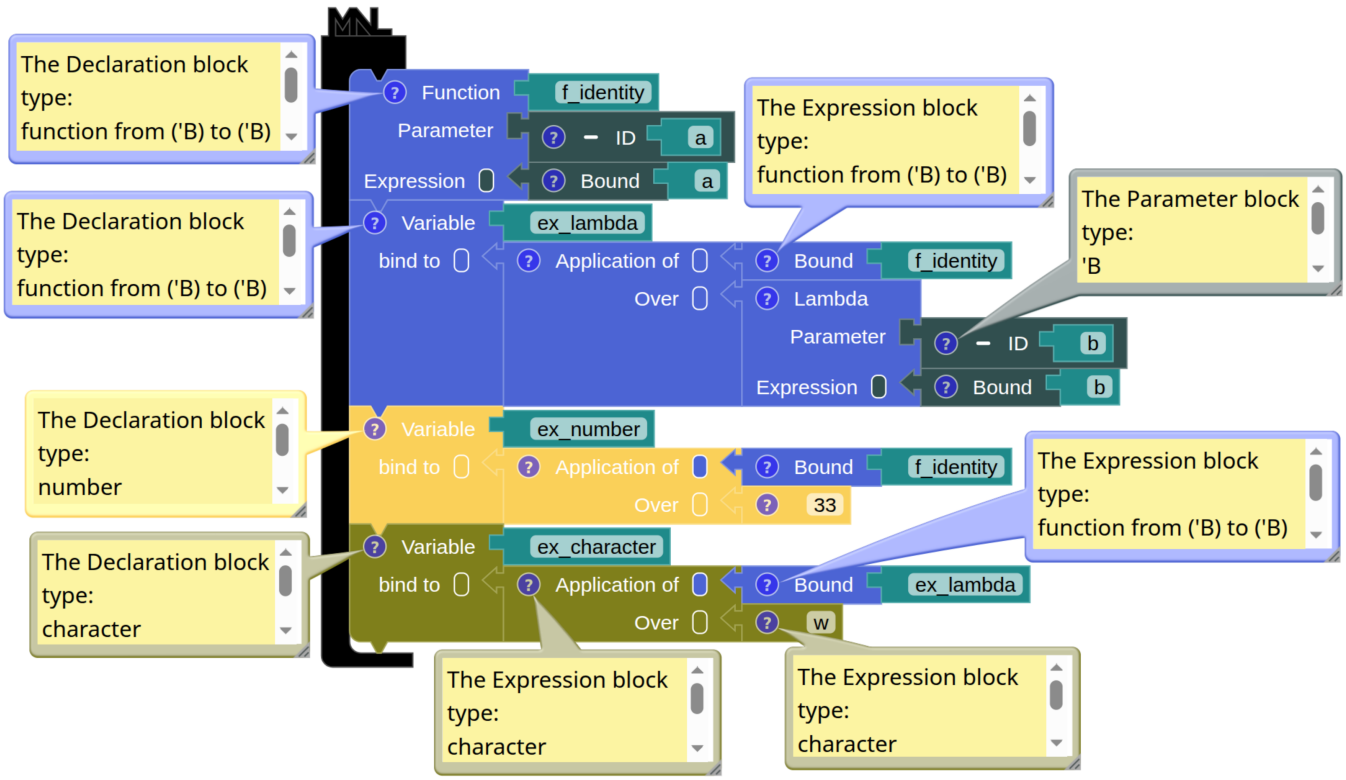


Figure 1. Identity function and its application

Abstract

The complexity of functional programming languages can pose a challenge for learners. However, the use of block-based languages in learning programming can lower the barriers to the learning process. While many block languages have been created, they often lack essential features of functional languages and do not include the type inference.

This gap serves as motivation to develop a block-based functional programming language that provides visual information about three programming language conventions. The development process begins with designing text syntax, transforming it into blocks, drafting typing rules for visual languages, and building a new functional block language called Macaca Nigra Language.

Case studies of Macaca Nigra Language demonstrate that it effectively provides visual clues through shapes and colors regarding the three conventions of programming languages.

CCS Concepts: • **Software and its engineering** → **Visual languages**; **Functional languages**.

Keywords: visual programming language, block-based programming language, functional programming language, reactive type system, type inference, polymorphic type, reactive block

PAINT '25, Singapore, Singapore

ACM ISBN 979-8-4007-2160-1/25/10
https://doi.org/10.1145/3759534.3762684

ACM Reference Format:
Steven Lolong. 2025. The MNL: A Block-Based Functional Programming Language with Reactive Blocks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3759534.3762684

1 Introduction

The advantages of functional programming, such as purity, immutability, and referential transparency, can facilitate reasoning about software correctness and concurrency [13] [23]. However, learning functional programming with its advanced features, such as higher-order functions (HoF) and implicit type inference, can be difficult for beginners, as well as for those accustomed to imperative programming.

In general, a programming language follows three conventions: syntax, semantics, and pragmatics. Syntax is the legal structure of sentences, semantics is the meaning of sentences, and pragmatics is the use of language. Novice programmers may find it difficult to understand one or more of the conventions [14]. Furthermore, Kurihara explains that if programmers do not understand syntax, they will make syntax writing errors, such as missing curly braces and punctuation marks. Semantic incomprehension prevents programmers from understanding the behavior of expressions and functions. Likewise, a lack of understanding of pragmatics causes programmers to have difficulty understanding when and how to use expressions and functions.

In parallel, visual programming offers more convenience than text programming. The use of icons, symbols, diagrams, and forms that closely resemble the programmer's mental representation in a visual language makes visual programming easier for novice programmers to understand. Another advantage of using graphics is that it can present much more information than text [21].

The advantages of displaying information through visual programming languages, including block programming languages, have caused block-based programming environments to gain substantial traction as effective tools for introducing computational thinking and programming concepts. It can also lower the barriers to learning programming, especially for beginners and in educational environments [26][32]. Abstracting the complexity of text-based syntax through visual metaphors allows users to focus on logical structure and algorithmic reasoning. Furthermore, visual languages with reactive systems can provide real-time feedback to programmers as event-driven changes occur. Feedback, such as suggestions and error messages, is convenient for novice programmers.

While block programming languages like Scratch [16], Blockly [9], App Inventor [34], and others [30] are effective tools for beginner and basic application development, they have limitations in expressiveness. Specifically, they do not support key features found in functional programming languages, including HoF, anonymous functions (Lambda), and guarantees of type soundness. These features are essential for both the study and practice of application development using functional languages.

1.1 Main Ideas

The absence of block language support for key features of functional programming languages has led to the development of a block-based functional programming language. This research introduces the Macaca Nigra Language (MNL), which is designed to be a block-based functional programming language. MNL supports Lambda abstraction, HoF, and implicit type inference. The language aims to offer programmers visual clues related to syntactic, semantic, and pragmatic conventions. Its reactive blocks can provide feedback when errors occur and offer visual suggestions through the color of connectable blocks.

1.1.1 Avoid Syntax Misconstruction. In terms of syntactic conventions, syntax errors can easily be avoided in block languages, where programmers do not need to type out syntax explicitly. However, syntax errors can still occur. For instance, in Standard ML (SML), Listing 1 may appear normal for imperative languages like C or mixed languages like JavaScript. However, according to SML's grammar, this is an error because the production rules for sequence expressions are specific to expressions and cannot be combined directly with declarations. To properly combine declarations and expressions, programmers can use the let-binding production rule, as shown in Listing 2.

Listing 1. SML example of misconstruction syntax

```
fun isAlphabetNumber(num) = (
  val mx = 92;
  val mn = 67;
  if((num >= 67) andalso (num <= 92)
  then true;
  else false;
)
```

Listing 2. SML example with correct syntax

```
fun isAlphabetNumber (num) = let
  val mx = 93;
  val mn = 67;
in
  if((num >= 67) andalso (num <= 92))
  then true;
  else false;
end
```

Syntax construction errors can be prevented by providing visual clues about production rules. This article discusses the transition from text-based grammar to block grammar, offering visual clues that help avoid syntax construction errors.

1.1.2 A Meaningful Syntax.

Text languages often utilize abbreviations in their syntax, which can sometimes hinder the learning process due to the lack of clarity. For example, in Listing 3, the term *incrr*(2) in the second line has a different meaning compared to *incrr* in the third line, even though their syntax appears almost identical. The expression *incrr*(2) represents the application of the *incrr* function over the value 2, while *incrr* pertains to the binding of values. The MNL grammar layout provided in this article illustrates how to effectively use meaningful whole words within block languages.

Listing 3. Javascript example of bound and function application

```
const incrr = (n) => n + 1
let three = incrr(2)
let anotherIncrr = incrr
let four = anotherIncrr(3)
```

1.1.3 One Color per Term Type.

In non-strongly typed languages like JavaScript, it is possible to use operators with operands of different types. For example, in the expression ''H' + 17', ''H'' is a string and '17' is a number. The result of this operation is ''H17'', and the data type of the result is a string. This practice of implicit type conversion can be problematic when building applications, as it may lead to unexpected errors in the output. Strongly typed programming languages do not allow such operations. Errors caused by semantic incomprehension, such as this, can be avoided by providing visual clues about the term type to the programmer. The section 3.2 explains how to create type rules based on the visual dimensions to help prevent these errors.

1.1.4 Advanced Suggestion.

The output of the debugging process for incomplete expressions is a notification indicating that the expression is incomplete, as shown in Listing 4. However, despite the expression being incomplete, it is still possible to infer the term type after the word "else" in the first line or after "if" in the second line. Advanced suggestions can help to determine the appropriate type match. Incomplete expressions can also assist novice programmers in understanding semantic conventions. This paper outlines the design and implementation techniques used in building MNL to provide advanced type suggestions visually.

Listing 4. Scala example of incomplete expressions

```
val err_1 = if(false) 3 else ;
val err_2 = 100 + (if(true) else 5);
```

1.1.5 Smart Constructor.

The pragmatic incomprehension of when and how to apply the function can lead to errors. For example, in Listing 5, the second line demonstrates an issue where the number of members in a tuple is less than the number of parameters required. Additionally, it is worth noting that the function's application in the third and fourth lines yields the same output, despite the differing parameters. This article includes a section that explains how to design a smart constructor block for tuples, which automatically adjusts the minimum required number of tuple members.

Listing 5. Scala example of an incomplete parameter

```
def add_two_inhabitants[A](a_tuple: (Int, A, Int
    )) = a_tuple(0) + a_tuple(2);
val incomplete = add_two_inhabitants((3, 4));
val comp_1 = add_two_inhabitants((3, true, 4));
val comp_2 = add_two_inhabitants((3, "add", 4));
```

1.2 Contributions

In this work, we provide a formal explanation of the syntax and semantics of MNL and describe the transformation of text grammar into block grammar. Additionally, the design of reactive blocks is outlined, and their capabilities are demonstrated through a case study. The formal representation clarifying the syntax and semantics, along with the syntactic transformation from text to blocks of MNL, leads to several contributions, which can be summarized as follows:

- We are introducing a new block-based functional programming language that harnesses the power of a reactive system.
- We present a comprehensive set of techniques aimed at facilitating the transition from text-based syntax to block-based syntax in an effective manner.
- We have implemented new typing rules and type constraints that blend two visual dimensions: shape and color, along with their implementation techniques in the reactive blocks.

2 Block-based Programming Language

Visual programming languages are taxonomically classified into flowcharts, data flow diagrams, spreadsheets, puzzle pieces (blocks), interface management systems, and forms [21]. Block programming languages utilize blocks that resemble puzzle pieces to structure expressions within the language. Each block has a unique shape or color that indicates how they connect with one another [17].

Block programming languages can facilitate vocabulary learning by minimizing cognitive load. They achieve this by breaking down code into smaller, more manageable components, which helps avoid syntax errors that can occur with text-based programming [9]. Some examples of block language development for domain-specific languages (DSLs) include Sonification Blocks for music [2], Autoblocks for software engineering [3], MIT App Inventor for education and computational thinking [22], and Robotics applications [33].

There are three options for developing block languages: creating a language from scratch, utilizing existing libraries, and extending an existing block language. Some libraries available for block language development include Blockly

[9], OpenBlocks [27], and Droplet [5]. Additionally, there are extensible block languages such as Scratch[1], MIT App Inventor[2], and Snap![3].

The development of MNL uses a library-based approach, specifically utilizing the Blockly library[4]. This choice facilitates enhanced functionality and flexibility in our programming efforts. The graphical user interface (GUI) of the Blockly library supports event-driven programming, which allows events to be triggered by various inputs such as keyboard events, mouse movements, gestures, and changes in visual properties like color, position, and shape. This capability facilitates the development of block languages tailored for reactive systems.

3 The Macaca Nigra Language

MNL extends the lambda calculus expressions denoted as $e ::= a \mid \lambda a.e \mid e\ e$ [12] by incorporating additional features, including name binding (declaration), let-binding, constants, conditional expressions, sequences of expressions, operators, and types.

The development process of MNL starts with the creation of a text-based syntax. Next, we define grammar types based on non-terminals and combine them with term types to generate robust typing rules that ensure the soundness of the blocks. Finally, the text-based syntaxes are converted into block-based syntax.

3.1 The Syntax

Syntax construction begins with the development of text-based grammars and types. Grammar is essential for demonstrating how tokens are organized to create valid expressions [8]. The three key elements of grammar are production rules, non-terminals, and terminals. Production rules define the relationships among production rules, terminals, and non-terminals. Terminals consist of fixed symbols, while non-terminals are variables that can have one or more associated production rules, terminals, and non-terminals.

MNL grammars are divided into two categories: basic and core. The basic grammar consists of non-terminal groupings with associated production rules, as well as commonly used terminals. These terminals include digits for numbers (0 to 9), letters from the Roman alphabet, booleans for true and false values, characters for ASCII, and strings. The core grammar, illustrated in the Appendix, Figure 14.

The clarity of SML'97's core grammar [28] [20] has inspired the style of many MNL grammars, including constructs like let binding, sequences, and case analysis.

The MNL language is designed with block-language capabilities in mind, meaning that users do not need to type out syntax. As a result, all terminals use whole words rather than abbreviations. This practice offers a brief insight into the semantics and purpose of the production rules. For example, to retrieve a field from a record, MNL uses the expression **Get field** *uid* **from Record** *exp*. This practice contrasts with the SML language, which employs a notation such as **#** *lab* [28]. Using whole words in MNL ensures that each production rule forms a simple, yet meaningful sentence, making it easier for users to understand the intended operations.

Including a meaningful word can address the issue of insufficient information. For instance, the distinction between application and binding expressions, as demonstrated in Listing 3, can be clarified by adding a word that conveys the semantic meaning of the expression. The production rule for function application includes a terminal that conveys the meaning of the expression, represented as **Application of** exp_1 **Over** exp_2. Additionally, the use of bound variables adds a terminal denoted as **Bound** *uid*. This approach provides clearer information, helping to avoid confusion for programmers.

To avoid confusion when discussing non-terminals, we use the term child non-terminal and parent non-terminal. For example, in the production rule *dec* ::= **Variable** *uid* **bind to** *exp*, *dec* is considered the parent non-terminal, while *uid* and *exp* are the child non-terminals.

The core grammar consists of nine non-terminals, namely *uid*, *con*, *pat*, *mtc*, *field*, *param*, *exp*, *dec*, and *prog*, which become the main tie in the construction of production rules. Each of the nine non-terminals has a unique name, which is intended to prevent errors in formulating the production rules.

3.2 The Term Type

The type system in programming languages serves as an essential tool to ensure program correctness and detect errors when applying operations to specific values [24]. The type system classifies types based on the operators that can be applicable to them. As a functional language with a type system that checks before applying certain operators, MNL has term types for primitive, field, polymorphic, tuple, record, list, function, variable, and match-do.

The design of MNL term type annotations adheres to the same principles used in grammar design, which emphasizes the use of whole words to convey meaning. For instance, consider the term type annotation for the tuple ("Hello", 7). In Scala, it is defined as **(String, Int)**, while in SML, it is represented as **int*string**. In contrast, MNL expresses this as **tuple of (string and number)**, which demonstrates that the terminal annotation design for term types in MNL is capable of conveying more detailed meanings. The term types of MNL are listed in the Appendix, Figure 15.

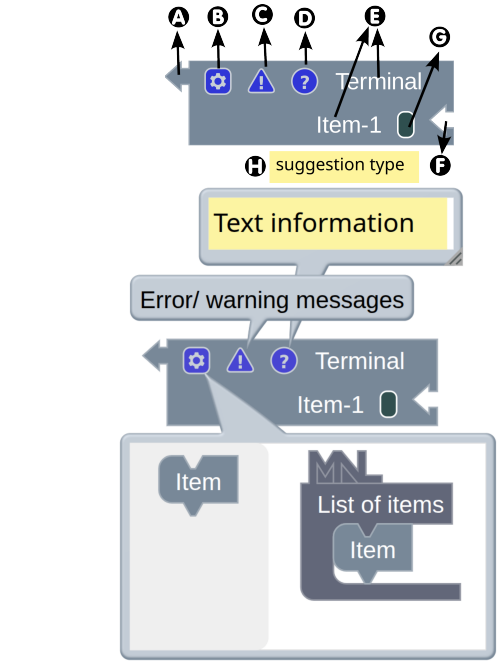
[1] https://scratch.mit.edu/
[2] https://appinventor.mit.edu/
[3] https://snap.berkeley.edu/
[4] https://developers.google.com/blockly

3.3 Syntax Metamorphosis

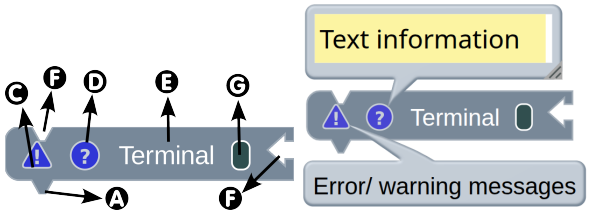
The metamorphosis from text grammar to block grammar begins with designing the block anatomy, then mapping each text production rule to the block structure, such as shape, color, and terminal position. Next is the updating of the type system to match the block language structure. The final part is the construction of MNL typing rules.

3.3.1 Block Anatomy. The primary objective of block anatomy design is to provide programmers with information and visual clues regarding the grammar (production rules, non-terminals, and terminals), term types, suggestions for term types, errors, and details about the block itself. After thoroughly analyzing the text grammar and carefully adapting the distinctive features of Blockly, we have designed the MNL block anatomy, which is depicted in Figure 2. The guiding principles of this innovative design can be elegantly summarized as follows:

1. One block for one production rule.
2. All rules in non-terminal declarations utilize blocks that can be organized vertically.
3. Each block contains information about parent non-terminals, child non-terminals, terminals, term types, suggestions for term types, error messages, suggestion messages, additional messages, and a toolbox for adding or removing non-terminals.
4. Non-terminals serve as connections between production rules. MNL uses the shape of the connecting block (notch) as a clue for non-terminals. As shown in Figures 2a and 2b, there are output notches for parent non-terminals and input notches for child non-terminals.
5. The color of the block represents the term type.
6. The mutator icon is a clickable icon that displays a toolbox for adding or subtracting the number of child non-terminals. This icon is specific to production rules that have several child non-terminals that can be added or subtracted by the programmer.
7. The debugger icon is a clickable icon for displaying error messages in text form.
8. Message icon is a clickable icon for displaying additional messages in text form.
9. Terminal is a location for terminals.
10. Suggestion box is the location for the term type suggestion in the form of colors.
11. SB-tooltip is a tooltip to display term type suggestions in text form.
12. The characteristic of a vertical block is that it has only one notch at the top and only one notch at the bottom for the non-terminal parent.
13. For vertical blocks, a mutator is unnecessary since the quantity can be increased by stacking blocks at the top or bottom.
14. The horizontal block is characterized by having only one notch on the left for the non-terminal parent.
15. The non-terminal child will be located on the right side or inside the block.



(a) Horizontal block design



(b) Vertical block design

legend:
A = output notch, B = mutator icon, C = debugger icon,
D = message icon, E = terminal, F = input notch,
G = suggestion box, H = SB-tooltip.

Figure 2. Block anatomy

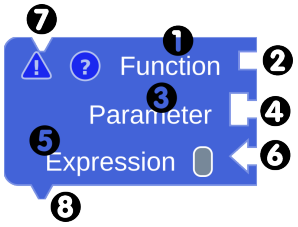
3.3.2 Text to Block. The transformation of text-based grammar production rules to blocks involves transforming production rules into blocks that can be connected. The rules outlined in section 3.3.1 support this transformation by providing a direct method to align the components of the production rules with the block structure. For illustration, Figure 3 demonstrates this transformation.

Figure 3a depicts the original text production rule, while Figure 3b shows the corresponding block generated from that rule. The non-terminal parent of the function abstraction is *dec*, while its child non-terminals are *uid*, *param*, and *exp*. The resulting block has three input notches, one

notch for each child non-terminal. These notches will connect the block with other blocks. Different shapes for each non-terminal can help programmers avoid syntax misconstruction.

In Figure 3b, the terminals of this production rule are **Function, Parameter** and **Expression**, which are represented as text on the block. All production rules with a non-terminal parent *dec* will feature a vertical block design, as shown in 2b, and others will have a horizontal design as in Figure 2a.

(a) Production rule of function abstraction



(b) The function abstraction block

Figure 3. Text to block of function abstraction

Another example is the transformation of a record constructor production rule as illustrated in Figure 4. Figure 4a displays the text production rule, while Figure 4b shows the corresponding block generated from this rule. The terminals in this production rule are **Record constructor** (1) and **Field** (2), which are displayed on the block as text. This production rule has a dynamic number of terminals **Field** and non-terminals *exp*, necessitating the use of a mutator (3) that can adjust the number of these terminals and non-terminals by adding or removing them as needed.

(a) Production rule of record constructor

(b) The block of record constructor

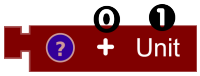
Figure 4. Text to block of record constructor

3.3.3 Reducing the Number of Blocks. Merging multiple production rules that share the same non-terminal parent into a single block can reduce the overall number of blocks. For example, the combination of the production rules *unit* and *identifier* as illustrated in Figure 5a. This approach consolidates two production rules. The implementation of the *unit* production rule is shown in Figure 5b, while Figure 5c depicts the implementation of the *identifier* production rule.

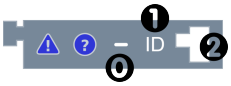
In this merged block, there is a clickable icon (0) that allows for easy transformation between the *unit* and *identifier* production rules, enabling the programmer to switch between them as needed swiftly. The merging not only reduces the number of blocks but also gives programmers the flexibility to select the production rule that fits the context without needing to rearrange blocks.

Furthermore, this merger simplifies the visual layout of the blocks, so that the programmer can concentrate more on the program logic, rather than on the block arrangement. Interestingly, this merge does not change the meaning of the expression, as both production rules serve to express the parameters in the expression.

(a) Non-terminal parameter



(b) Text to block of parameter - unit



(c) Text to block of parameter - identifier

Figure 5. Text to block of non-terminal parameter

3.3.4 Text to Color. MNL not only provides visual clues through shapes to help assemble correct syntax, but also emphasizes the importance of term types in programming. Term types are a fundamental concept in programming languages, as they convey information about the type of data used in an expression. Understanding term types enables programmers to learn how to utilize them and what operations can be performed on them.

Therefore, MNL represents term types through colors. Each term type has a distinct color, allowing the programmer to easily distinguish the term type being constructed. This color also helps in identifying the term type errors while constructing blocks. The transformation of text types in the Appendix, Figure 15 to colors can be seen in the Appendix, Figure 16.

The use of colors to represent term types has its limitations, especially when it comes to complex term types presented within a single block. An excessive variety of colors

can make it challenging for programmers to differentiate between these term types. Therefore, MNL only represents the top-level term types in color. More complex term types, such as functions, tuples, records, and lists, are also represented by color but only at the top level. Detailed information about these complex term types is provided through text.

For instance, in Figure 1, the function declaration is displayed with the color corresponding to the function type, while the specific details about the function are accessible via the message icon when clicked.

3.3.5 Block Type. Type construction for visual languages differs from that of text languages. Type checking in text languages operates in a single dimension, consisting only of a stream of letters for term types, with syntax structure checking occurring at the parsing stage. On the other hand, visual languages can have two or more dimensions, necessitating more complex typing rules to represent each dimension.

MNL utilizes two visual dimensions within a single block to provide clues to the programmer: shape for grammar and color for term types. Non-terminals determine how production rules can be interconnected, which is represented by the shape of the block notch. For this reason, a grammar type (GT) is introduced, where the keyword used is taken from non-terminals. Hence, the block type is a conjunction of the grammar type and the term type, as illustrated in Figure 17.

The rule of one block for one production rule in text-to-block transformation and the block type rule have created a gap in the availability of term types for the supporting production rules, especially those of the non-terminal *uid*. To fill this gap, a new term type has been introduced for all supporting production rules: *nothing*. This type indicates that the supporting production rule has no applicable operators. Additionally, it means that no values are available for use in it.

With the addition of the term type *nothing*, the term types for all production rules in MNL are now complete and can be used to build typing rules.

3.3.6 Typing Rules. Typing rules are a set of rules used to determine the type of an expression in a programming language. These rules are used to ensure that the expression written by the programmer conforms to the expected type. Type rules are also used to ensure that expressions written by programmers do not generate type errors when executed.

The typing rules in MNL are derived from the commonly used typing rules in functional programming languages. However, MNL includes an additional encoding for the visual aspects of shapes. For instance, Figure 6a illustrates a type rule for conditions typically found in text-based languages, represented by a single type (one dimension), written as T. In contrast, Figure 6b presents an MNL block type rule, where the block type is represented as $Expression \otimes T$.

$$\frac{\Gamma \vdash exp_1 : boolean \quad \Gamma \vdash exp_2 : T \quad \Gamma \vdash exp_3 : T}{\Gamma \vdash \textbf{Condition When } exp_1 \textbf{ Is true } exp_2 \textbf{ Otherwise } exp_3 : T}\ \text{T-Condition}$$

(a) Typing rule of the Condition

$$\frac{\Gamma \vdash exp_1 : Expression \otimes boolean \quad \Gamma \vdash exp_2 : Expression \otimes T \quad \Gamma \vdash exp_3 : Expression \otimes T}{\Gamma \vdash \textbf{Condition When } exp_1 \textbf{ Is true } exp_2 \textbf{ Otherwise } exp_3 : Expression \otimes T}\ \text{BT-Condition}$$

(b) Refinement typing rule of the Condition

Figure 6. Typing rules refinement

In one-dimensional languages, there is no need for a typing rule regarding the names of variables or parameters. However, in block languages, this rule is essential to ensure that the type evaluation result of a well-typed block ($b : GT \otimes T$) will produce a value or allow for further evaluation of b (i.e., $b \rightarrow b'$).

With the introduction of the ID typing rule, the MNL typing rules for variable and function declarations, parameters, and bounds differ from those in text-based languages, where the ID typing rule is integrated into the premises, as illustrated in the Appendix, Figure 19. This typing rule also ensures that the names used in the variable, function, or parameter do not conflict with existing names in the context (Γ). Additionally, for the bound typing rule, the ID typing rule ensures that the name used in the bound is present in the Γ.

In addition to the ID typing rules, MNL established three supplementary typing rules in the normal form: constant, pattern, and parameter. As illustrated in Appendix, Figure 18.

If the part of the grammar is omitted, the MNL typing rules will align with the typing rules in Pierce's functional programming language [24]. Therefore, the proof of the term type for the MNL typing rule can utilize Pierce's proof. Regarding the grammar type proof, this can be accomplished directly, as the grammar type features only one non-terminal parent, denoted as GT, and all its production rules are terminal.

3.3.7 Constraint Typing Rules. Constraint typing rules are crucial in the type inference process, particularly for inferring the types of functions. The construction of MNL's constraint typing rules uses the method proposed by Pierce

[24]. However, a notable distinction is the inclusion of grammar types for each rule. An example of a constraint typing rule for a condition can be found in Figure 7.

$$\frac{\begin{array}{c}\Gamma \vdash b_1 : Exp \otimes T_1 \quad |_{X_1} \; C_1 \\ \Gamma \vdash b_2 : Exp \otimes T_2 \quad |_{X_2} \; C_2 \\ \Gamma \vdash b_3 : Exp \otimes T_3 \quad |_{X_3} \; C_3 \\ X_1, X_2, X_3 \; non\; overlapping \\ C' = C_1 \cup C_2 \cup C_3 \cup \{T_1 = boolean, T_2 = T_3\}\end{array}}{\begin{array}{c}\Gamma \vdash \textbf{Condition When } b_1 \textbf{ Is true } b_2 \textbf{ Otherwise } b_3 \\ : Exp \otimes T_2 \quad |_{X_1 \cup X_2 \cup X_3} \; C'\end{array}} \text{CT-Cond}$$

Figure 7. Constraint typing rule of the Condition

4 Reactive System

The MNL reactive system is a system designed to respond to changes in data or state quickly and efficiently, utilizing reactive programming principles that enable programmers to compose visual syntax that dynamically adapts to changes in data. Reactive programming is an approach that enables the system to respond to changes in data or state quickly and efficiently, thereby allowing the development of responsive and adaptive applications that can adapt to changes that occur [4].

The reactive system in MNL is centered on the interactions between blocks. Each block in MNL is called a reactive block because it can detect events and respond. Events can be changes in variable values, the addition or subtraction of child blocks, changes in system state, or interactions with users. Meanwhile, a reaction is a change that occurs in the system in response to an event performed by the programmer. Reactions can take various forms, such as changes in block type, color, or shape, as well as notifications regarding errors or needs. It can also be a change in the context used to determine the term type.

4.1 Reactive Blocks

Typing rule checking consists of two main components: grammar type checking and term type checking. Grammar type checking occurs when a programmer attempts to connect two blocks. If the non-terminal parent represented by the output notch of one block matches the non-terminal child represented by the input notch of the other block, the two blocks can be connected. Otherwise, the two blocks will be separated.

The validation algorithm is detailed in the Appendix, Algorithm 1. In this algorithm, the Event refers to the action that triggers the response, the Reaction denotes how the block responds, and the Initial represents the property established when the block is created. Lines 1 through 6 outline the activities that are initiated by the Event.

Term type checking is performed when changes are made to a block, such as connecting or disconnecting a block, changing its properties, or changing the number of child blocks. This process ensures that the inferred term types conform to the typing rules. When the term type does not match the typing rules, the system will provide an error message to the programmer. This feedback will not only indicate what requirement has not been met but will also offer suggestions to help the programmer adjust the inferred term types to ensure compliance with the typing rules.

Based on the typing rule, MNL grammar production rules can be categorized into two parts. The first category consists of production rules with monomorphic term types (mono-color blocks), such as arithmetic operator production rules that have a term type of *number*. The second category is production rules with polymorphic term types such as production rules of $parameter - identifier$, *bound variable*, *application*, $variable\ definition$, etc.. Blocks with monomorphic term types will have a fixed color. In contrast, blocks with polymorphic term types will have a changeable color (chameleon block) according to the term type generated after the type inference process.

The state of a block is categorized into three different conditions: complete, partially complete, and incomplete. The complete state means that all input notches are connected to child blocks. The partially complete state is a state where only some of the input notches are connected to child blocks. Meanwhile, the incomplete state means that none of the input notches have been connected to child blocks.

Reactive blocks will always validate the term type of a block, even if it is in a partially complete or incomplete state. In the partially complete or complete state, information from the child block is crucial, including the term type. The goal is to provide accurate information to the programmer, even if the state of the child block is also partially complete, incomplete, or contains an error. To address this need, each block requires a default term type to ensure it is well-typed, thus preventing errors during the inference process. The default term type of each block must comply with the typing rules.

4.1.1 The Mono-color Block. The reactive behavior of the mono-color block can be seen in the Appendix, Algorithm 2. During block initialization, three essential properties are established: the default term type (T), $T_{premises}$, and *color*. The term type of the properties T and $T_{premises}$ must conform to the typing rules. The T property will remain constant, while the $T_{premises}$ property will change. Since the $T_{premises}$ property is mutable (see Line 9), it will always be set to the term type that matches the type rule every time a change in the block occurs (see Line 2). The *color* property of the block will be aligned with the T property, ensuring that the block's color remains unchanged. The *error* property on Line 1 serves to hold all errors that occur. Programmers can

view the errors in text form by clicking on the debugger icon. If the block is a child block, then the information from the *error* property will be used by the parent block to determine whether the type inference process can be performed or not.

4.1.2 The Chameleon Block. In the Appendix, Algorithm 3 for managing the reactive behavior of chameleon blocks highlights three key aspects that differentiate the handling of chameleon blocks from that of mono-color blocks. First, the default term type, where the default type of the Chameleon block is *polymorphic*, which is presented in gray as shown in Figure 8. Second, whenever a change occurs to the block, the default term type is reset to *polymorphic* before type inference is performed (see Line 1). Third, there is a process for updating the T property based on the inferred term type as in Line 10. Type inference uses the algorithm proposed by Milner [19]. Finally, the *color* property is updated to reflect the T property. The change in the *color* property makes the color of the block change as shown in Figure 10b.

Suppose the T property contains complex term type information that cannot be fully represented by color. In that case, the programmer can view a detailed textual description of the term type description by clicking on the message icon.

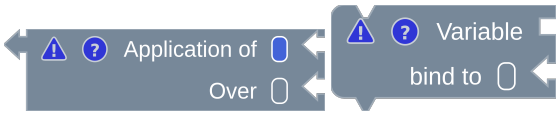


Figure 8. The chameleon block default color

4.1.3 The Function Block. The default term type for function blocks is **from** *polymorphic* **to** *polymorphic*, which is represented with blue color. The T property will change according to the term type generated by the inference process, but the *color* property will not change. The immutable *color* property for a complex term type, such as function, tuple, list, and record, aims to reduce the complexity of color usage that can make it difficult for programmers to recognize all color combinations. The algorithm is detailed in the Appendix, specifically in Algorithm 4.

However, if the T property changes while the *color* property remains the same, it can result in a lack of information for child blocks. To address this issue, an update action is enforced for the child blocks (Line 15). Additionally, term type inference for functions can lead to infinite loops, so it is essential to perform a check beforehand to prevent this (Line 10).

4.2 Coloring the Suggestion Box

The suggestion box provides information about the appropriate term type corresponding to the input notch next to it, which is indicated by color. The construction of term type hints for the suggestion box utilizes the same constraint typing rules used in type inference for a function. However, a key difference lies in the naming of polymorphic types.

In the constraint typing rule for the block condition (as shown in Figure 7), polymorphic types for X_2 and X_3 are substituted with the type variable 'A during function block type inference. In contrast, when constructing the term type for the suggestion box, the polymorphic type 'any' is still used (displayed in gray), as shown in Figure 9a.

The construction of the term type for the suggestion box is the last step in each algorithm. Positioning it in the final step aims to provide accurate hints, even when the block state is incomplete, as in Figure 9a, partially complete as in Figures 9b, or complete but not comply with the typing rules as in Figure 9c.

However, when the block state is complete and the term type complies with the type rules, the term type for the suggestion box must comply with the typing rules as shown in Figure 9d.

When the block state is incomplete and the block is connected to its parent block, then the term type construction for the suggestion box will depend on the parent block, as illustrated in Figure 9e. However, if the block state is complete or partially complete, then the term type construction for the suggestion block will depend on the child block, as shown in Figure 9f.

4.3 The Smart Block

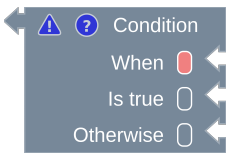
The smart block is the tuple constructor block, which adjusts the minimum number of members to be created. By default, this block starts with one member, as illustrated in Figure 10a. However, suppose this block is connected to another parent block where the term type input is a tuple. In that case, the tuple constructor block will adjust the minimum number of members as shown in Figure 10b. The adjustment of the minimum number of members is done before building the term type for the suggestion box, as in Line 25 in Algorithm 3. As a result, when constructing the term type for the suggestion box, each member of the tuple constructor block will have an associated suggestion box for every input notch.

5 Case Studies

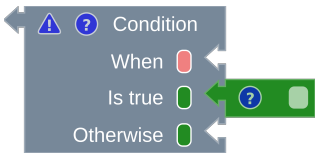
The case study is divided into three parts. The first part examines how reactive blocks convey information about term types through the use of color and text. The second part explores how reactive blocks inform programmers about errors. Finally, the third part shows HoF abstraction and its application.

5.1 Reactive with Color and Text

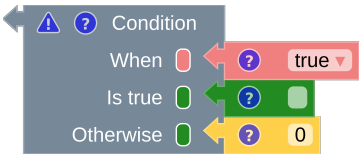
In Figure 1, an identity function is illustrated along with three of its applications. The figure illustrates how the term



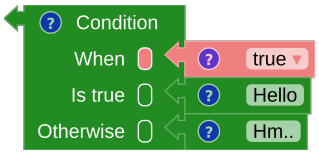
(a) The incomplete block



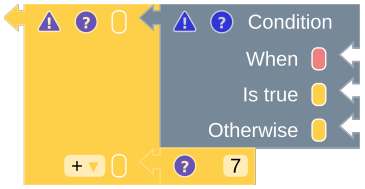
(b) The partial complete block



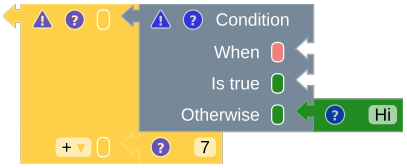
(c) The complete block-1, but do not comply with type rules



(d) The complete block-2

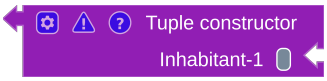


(e) The suggestion type is inferred from the Parent block

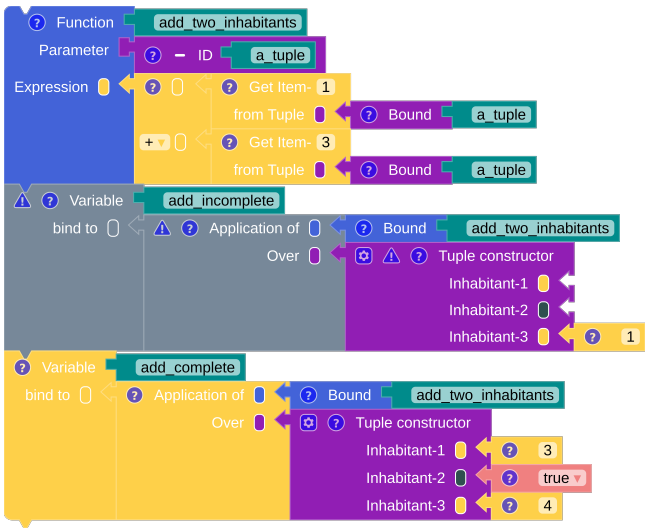


(f) The suggestion type is inferred from the Child block

Figure 9. Suggestion box



(a) The tuple constructor block



(b) Automatically adjusts the amount of input

Figure 10. The smart block

type is represented through color and text for parameters, bindings, applications, and variable blocks, aligning with its typing rules. It indicates that the information provided to the user is accurate. The color changes are visible to the programmer, while the textual term type can be accessed by clicking on the message icon.

5.2 Error Messages

Figure 11 illustrates that the reactive block can provide error information, including incomplete entries (❶) and naming conflicts (❷). The suggestion block is also capable of displaying information in text format (❸) as an additional hint when the suggested term type cannot be detailed using color. Furthermore, the debugging system can offer information about errors when the term type of the connected term block does not comply with the type rules, as seen in Figure 12.

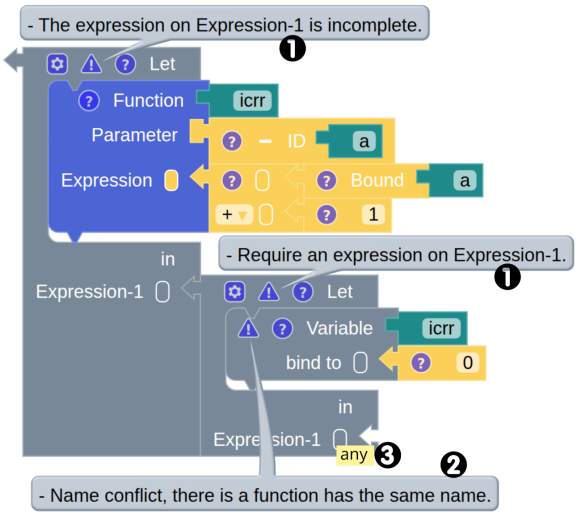


Figure 11. Incomplete block and name conflict

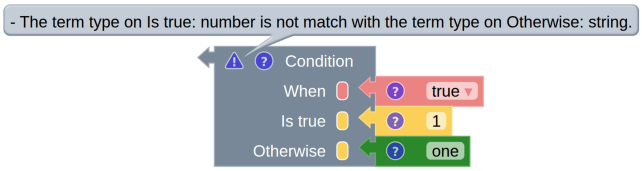


Figure 12. Does not comply with typing rules.

5.3 Higher Order Function

In Figure 13, there is a function with the term type from (*string*) to (from (*string*) to (from (*string*) to (*string*)). The first application to *string* returns a new function value with a term type from (*string*) to (from (*string*) to (*string*)) which is bound to the variable $first\_app$. Then the variable $first\_app$ with the function term type is reapplied to *string* and returns a value with the function term type (*string*) to (*string*) and is bound to the variable *second_app*.

This case illustrates MNL's capability for abstraction and HoF usage, where reactive blocks can perform type inference seamlessly. Reactive blocks can visually display the term type of the result of applying a function to a term type through color and text.

6 Related Work

Related work began with methods for transforming textual grammars into block grammars and continued with the capabilities of other block-based programming languages.

6.1 Grammar Transformation

The development time for MNL closely aligns with the research conducted by Merino et al. [17, 18] on transforming text-based grammar into block-based grammar. Some transformation techniques discussed in Section 3.3 share similarities, such as using one block for each production rule. For example, statements are arranged in a vertical layout while expressions use a horizontal layout. Non-terminals are represented by shapes, terminals (or syntax) are depicted as labels, and certain non-terminals that accept input from programmers are illustrated with input forms such as text boxes or combo boxes.

The inlining techniques that combine multiple rules into a single block serve a similar purpose to those outlined in Sub section 3.3.3. However, the merging of rules in Sub section 3.3.3 occurs dynamically through programming, allowing for the addition or subtraction of labels and inputs.

While Merino et al.'s research focused on grammar transformation, the heuristics in Section 3.3 introduced optimizations for visual cues. These include clickable icons to indicate errors and provide additional information, such as the term type in text form. Color-boxes are also used to suggest term types, accompanied by tooltips that offer details regarding the suggested term types in text form.

6.2 Partial Parsing

Partial parsing in MNL is employed to determine the term type in the suggestion box, similar to the research conducted by Beckmann et al. [6, 7]. The goal of partial parsing in MNL is to gather the essential information needed for calculating term types in the suggestion box. In contrast, Beckmann et al.'s research focuses on identifying potential block arrangements that can be used for empty or incomplete inputs.

In MNL, when computing the term type for the suggestion box and the block state is partially complete, all empty inputs are replaced with the tree structure of a new block with a term type "any." Conversely, Beckmann et al.'s approach substitutes empty or incomplete inputs with new tree structures that are contextually appropriate.

Additionally, if the block state is incomplete and it is connected to a parent block, partial parsing in MNL temporarily replaces the node of the incomplete block in the parent tree with a new complete block node marked as "bound." This allows for the computation of the term type for the suggestion box within that block.

6.3 Related Environments

There are several BPLs that use shapes or colors to distinguish term types, such as Scratch, Typeblock, Polymorphic block, OCaml Blockly, and Bootstrap. Typeblock uses shapes (notches) to indicate complex term types such as lists, tuples, and functions. There is no parametric polymorphism, and no color is used to represent term types. The goal of this project is to assist programmers in understanding the term types used in functional programming [31].

Extending Typeblock, the Polymorphic block introduces the use of colors and shapes to describe term types and variables. It combines shapes and colors in a block connector. In this design, shapes are used to distinguish different types, while colors represent variables [15]. The polymorphic part is the shape of the connector, which is dependent and will change when it has a matching type.

Another BPL that uses shapes to distinguish types is Scratch. Scratch has elliptical shapes for integers, angled shapes for booleans, and small flat trapezoids are used for statements [29]. The Snap! grammar structure consists of two main categories: declarations and expressions. Declarations can be arranged vertically, while expressions are arranged horizontally. The shape of the notch in the expression group varies depending on the type of value produced [11]. Additionally, colors are used to categorize blocks based on their function. Both Scratch and Snap! Using a complete English word for the terminal as a label.

OCaml Blockly utilizes a combination of shapes and colors to represent different types, with each type assigned a unique shape. Simple types are depicted with distinct shapes, while polymorphic types combine both shape and color. OCaml Blockly categorizes non-terminals into two main groups: declarations and expressions. The supporting non-terminals are integrated into blocks, such as names for "let" and "type" declarations [1].

Another approach to distinguishing term types comes from Bootstrap [5]. Bootstrap is a block-based editor that uses colors to represent five term types, including gray for polymorphic types. The gray color for polymorphic blocks will

[5]https://bootstrapworld.org/

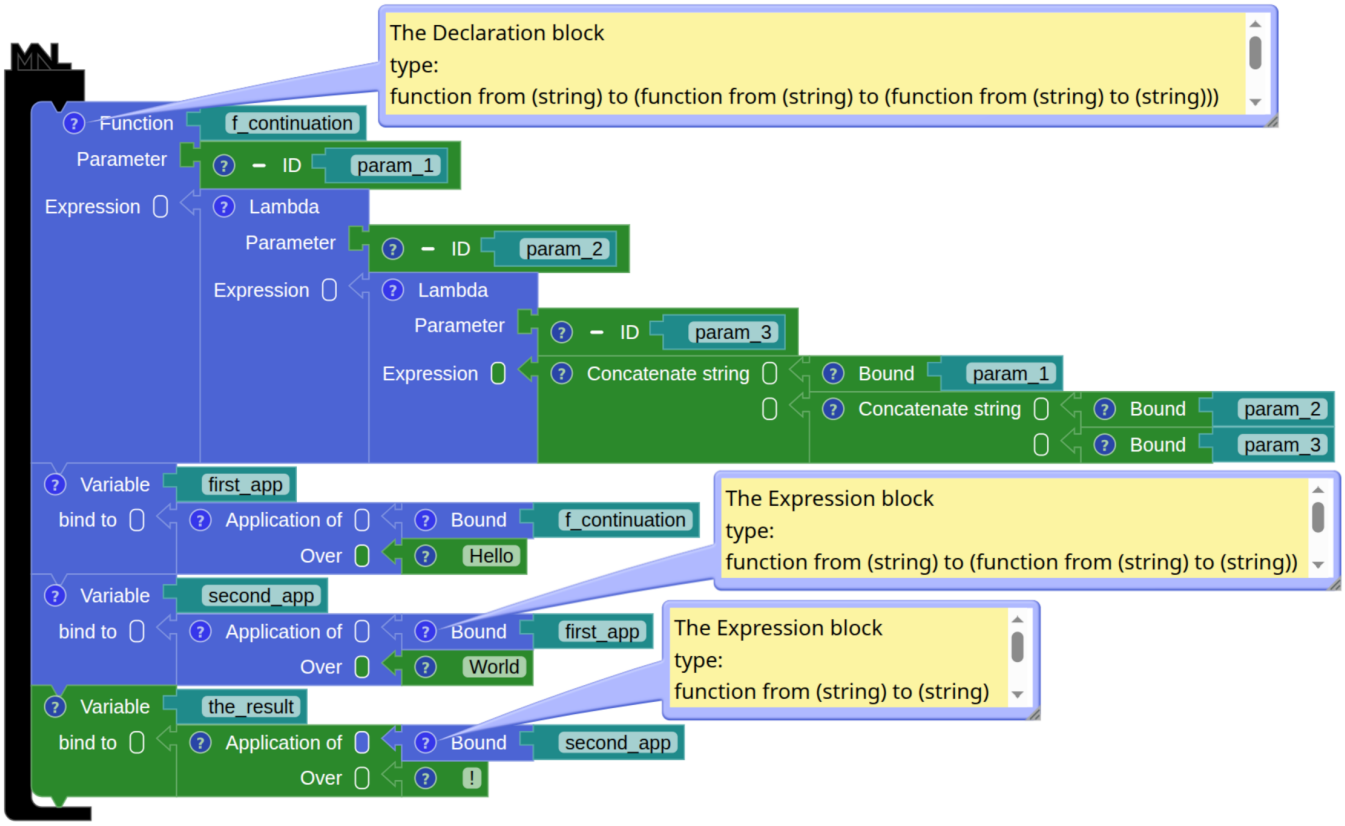


Figure 13. Higher order function

change after their type is determined during program construction. Besides Bootstrap, Poole also uses color and text to distinguish term types. However, for polymorphic types, he used a hatching pattern. Furthermore, Poole designed color-shape combinations for container types such as lists and tuples [25].

However, none of them explain how to distinguish production rules in the grammar to help programmers avoid syntax errors. In contrast, MNL designed an approach that not only provides information about types but also information about grammar. MNL leverages the advantages of visual languages to present multi-dimensional information. Consequently, MNL encodes not just one but three types of information: grammar, term type, and meaning. It integrates color and text to represent term types, shapes to indicate grammar, and natural language to convey meaning, all within a single block. When programmers examine the block, they can identify grammar through shapes, term types through color and text, and the natural language text to enhance their understanding of the meaning. The inclusion of visual notation for shape and color in MNL aligns with Green's research on cognitive dimensions [10].

Another difference is that the result of term type inference in MNL provides visual clues through the color of the suggestion box about the term types that can be connected, even when the block state is incomplete or partially complete. Furthermore, reactive blocks, especially the smart block, can adjust the minimum number of tuple members for function application.

7 Discussion

The separation of the non-terminal *uid* into a separate block in MNL aims to inform the programmer that *uid* is the supporting non-terminal. However, it is essential for variable, parameter, and function declarations. This separation not only introduces a new block but also adds a new data type *nothing* to the type rules. Meaning, additional time for the term type computation. Therefore, if the number of blocks and efficiency are the primary focus, the block for *uid* can be removed and *uid* incorporated into the block using a text box.

The implementation of term type checking is integrated into the block code structure and triggered by programmer actions or changes in the block's properties, making the block reactive. However, the price to pay is high processor utilization due to the real-time computation of term types every time a change occurs in the block. If processor usage

can cause problems, the type of computation can be driven by a single event, such as the click event.

Compilation processing in computers is a one-dimensional information processing method, specifically string stream processing. In contrast, visual languages can encompass multi-dimensional information. This work presents a technique for transforming multi-dimensional information from a visual language into a one-dimensional textual format suitable for typing rules. It employs conjunctions to merge the two dimensions. Furthermore, encoding multi-dimensional visual information can utilize either disjunction or conjunction. When conjunctions is used, both pieces of information must be present for the combination to succeed. Conversely, if one piece of information is optional, a disjunction should be used.

The use of color in MNL has limitations when it comes to displaying complex term types, such as the term type for HoF: from (*string*) to (from (*string*) to (from (*string*) to (*string*)) or complex container types. Thus, MNL relies on text to convey information about complex term types. Additionally, combining patterns and colors could be considered as an alternative way to represent complex term types.

8 Conclusion

This work presents several interesting findings. First, it demonstrates heuristic techniques for transforming text-based syntax into block-based syntax. Second, it outlines how to construct typing rules and constraint typing rules that convert two-dimensional information into one-dimensional data, along with their implementation for type validation in reactive systems. Finally, it demonstrates techniques for implementing constraint typing rules within reactive blocks, providing hints about the term type that can be connected even when the block state is incomplete, partially complete, or contains errors.

All of the above techniques are used to build reactive systems, especially a reactive block with its capabilities to provide visual clues about syntax, semantics, and pragmatic conventions to the programmer. The capabilities of MNL reactive blocks have been demonstrated in Section 5.

One aspect to consider for further development is providing information to programmers with limited color perception or those who are visually impaired.

Green's research on cognitive dimensions of notations [10] provides a helpful framework for analyzing the use of shape, color, and text dimensions in MNL. In the next step, it is essential to examine whether representing each production rule with a single block is optimal, as there may be more effective ways to convey grammatical information to programmers. Additionally, we should evaluate whether the suggestion box for term types is adequate in helping programmers understand term type calculations, in order to prevent incorrect function applications.

A Grammar, Type, Typing Rules, and Algorithm

This appendix presents the text-based core grammars, the term types and their colors, algorithms, as well as the new typing rules.

uid ::=	**identifier:**
Identifier *letter* <*letter* | *digit*>*	
con ::=	**constant:**
string *string* | **number** *number* | **character** *character* | **boolean** *boolean* | **unit**	
pat ::=	**pattern:**
con | **any**	
mtc ::=	**matching:**
is *pat* **do** *exp*	
field ::=	**field:**
Field *uid* = *exp*	
param ::=	**parameter:**
Unit | **ID** *uid*	*Unit or identifier*
exp ::=	**expression:**
con	*constant*
| **Application of** exp_1 **Over** exp_2	*function Application*
| **Bound** *uid*	*bound variable*
| **Lambda Parameter** *param* **Expression** *exp*	*lambda abstraction*
| **Condition When** exp_1 **Is true** exp_2 **Otherwise** exp_3	*condition*
| **Sequence of Expression-1** $exp_1 \ldots$ **Expression-n** exp_n	*sequence of expr., n ≥ 2*
| **Let** *dec* **In** ⟨ **Expression-n** *exp*⟩$^+$	*let binding, , n ≥ 1*
| **Switch** *exp* **Case-1** $mtc_1 \ldots$ **Case-n** mtc_n	*case analysis, n ≥ 2*
| **List constructor Empty list**	*empty list*
| **List constructor** ⟨**Inhabitant-n** exp_n⟩$^+$	*list constructor, n ≥ 1*
| **Tuple constructor** ⟨**Inhabitant-n** exp_n⟩$^+$	*tuple constructor, n ≥ 1*
| **Record constructor** ⟨**Field-n** exp_n⟩$^+$	*record constructor, n ≥ 1*
| *description* exp_1 *operator* exp_2	*binary operator*
| *description exp*	*unary operator*
dec ::=	**declaration:**
Function *uid* **Parameter** *param* **Expression** *exp*	*function abstraction*
| **Variable** *uid* **bind to** *exp*	*variable definition*
| $dec_1 \ldots dec_n$	*sequence, n ≥ 2*
prog ::=	**program:**
MNL *dec*	*the root of the grammar*

legend:
- **Bold** = **terminal**
- *Italic* = *non terminal*
- ⟨⟩ = group
- ... = sequence
- X^+ = one or more X

Figure 14. Core grammar

FT ::=	**field type:**
field *letter* <*letter* | *digit*>* **with type** *T*	
PT ::=	**primitive type:**
number | **string** | **character** | **boolean** | **unit**	
T ::=	**type:**
PT	*primitive*
| **any**	*polymorphic*
| **list of** *T*	*list*
| **tuple of** (T_1 ⟨**and** T_n⟩*)	*tuple*, $n \geq 2$
| **record of** {FT_1 ⟨**and** FT_n⟩*}	*record*, $n \geq 2$
| '⟨*uppercase*⟩+	*type variable*
| **function from** (*T*) **to** (*T*)	*function*
| **match** *PT* **do** *T*	*match-do*
| **nothing**	*nothing*

Figure 15. Type

Type	Color	Type	Color
number	#FED049	nothing	#008B8B
string	#228b22	function	#4363D8
character	#808000	tuple	#911EB4
boolean	#F08080	record	#FF8C00
unit	#800000	match-do	#9A6324
any	#778899	type variable	#2F4F4F
list	#000075	error	#FF0000

Figure 16. Type's Color

GT ::=	**grammar type:**
Identifier | **Expression** | **Pattern** | **Match-Do** | **Field** | **Parameter** | **Declaration** | **Program**	
BT ::=	**block type:**
$GT \otimes T$	

Figure 17. Type of the Block

Algorithm 1: GT validation

Event : Connecting block
Reaction : Connect or disconnect blocks
Initial : GT_{input} ← **GT of input notch**

$GT_{childBlock}$ ← **GT of the child block's output notch**
if $GT_{input} = GT_{childBlock}$ **then**
  connect blocks
else
  disconnect blocks
end

Algorithm 2: Mono color block

Action : any changes on the block
Reaction : update message
Initial : T ← **the block typing rule**; *color* ← **color of** T; $T_{premises}$ ← **premises of the block typing rule**

error ← ∅
$T_{premises}$ ← **premises of the block typing rule**
BlockState ← **state of the block**
if *BlockState* = *complete* **then**
  childBlocksErrors ← **errors from child blocks**
  if *childBlocksErrors* = ∅ **then**
    T_{childs} ← **T of the child blocks**
    if $T_{premises} \equiv T_{childs}$ **then**
      Update premises `// if necessary`
    else
      error ← "type mismatch"
    end
  else
    error ← *childBlocksErrors*
  end
else if *BlockState* = *partialComplete* **then**
  childBlocksErrors ← **errors of child blocks**
  error ← *childBlocksErrors* ∪ "partial complete error"
else
  error ← "incomplete error"
end
suggestionBox ← **update suggestion type colorbox**

Algorithm 3: Chameleon block

Action : any changes on the block
Reaction : update the block color and type, or update error messages
Initial : $T \leftarrow$ *polymorphic*; $color \leftarrow$ **color of** T; $T_{premises} \leftarrow$ **premises of the block typing rule**

```
T ← polymorphic
error ← ∅
T_premises ← premises of the block typing rule
BlockState ← state of the block
if BlockState = complete then
    childBlocksErrors ← errors of child blocks
    if childBlocksErrors = ∅ then
        T_childs ← T of the child blocks
        if T_premises ≡ T_childs then
            T ← type inference
            Update premises // if necessary
        else
            error ← "type mismatch"
        end
    else
        error ← childBlocksErrors
    end
else if BlockState = partialComplete then
    childBlocksErrors ← errors of child blocks
    error ← childBlocksErrors ∪ "partial complete error"
else
    error ← "incomplete error"
end
color ← color of T
// the tuple constructor block only
Update smart block
suggestionBox ← update suggestion type colorbox
```

Algorithm 4: Function block

Action : any changes on the block
Reaction : update the type, update nested function's type, or update error messages
Initial : $T \leftarrow$ **the default function type**; $color \leftarrow$ **color of** T; $T_{premises} \leftarrow$ **premises of the block typing rule**

```
T ← the default function type
T_premises ← premises of the block typing rule
error ← ∅
childBlocksError ← ∅
BlockState ← state of the block
if BlockState = complete then
    childBlocksErrors ← errors of child blocks
    if childBlocksErrors = ∅ then
        T_child ← T of the child block
        if circularity detected then
            error ← 'circularity detected'
        else
            T ← type inference
            T_premises ← Update premises
            trigger updates for all child blocks
            if T_premises ≠ T_childs then
                error ← 'type mismatch'
            end
        end
    else
        error ← childBlocksErrors
    end
else if BlockState = partialComplete then
    childBlocksErrors ← errors of child blocks
    error ← childBlocksErrors ∪ 'partial complete error'
else
    error ← 'incomplete error'
end
suggestionBox ← update suggestion type colorbox
```

$$\frac{}{\Gamma \vdash \blacklozenge\, v : Exp \otimes T}\ \text{BT-Const}$$

$$\frac{}{\Gamma \vdash \blacklozenge\, v : Pat \otimes T}\ \text{BT-Pat}$$

$$\frac{}{\Gamma \vdash \textbf{ID}\ n : Idr \otimes nothing}\ \text{BT-Id}$$

$$\frac{}{\Gamma \vdash \textbf{Unit} : Par \otimes unit}\ \text{BT-ParamEmpty}$$

$$\frac{\Gamma \vdash b : Idr \otimes nothing}{\Gamma \vdash \textbf{ID}\ b : Par \otimes T}\ \text{BT-Param}$$

legend (abbr. of GT):
Dec = Declaration, Exp = Expression, Pat = Pattern,
Idr = Identifier, Par = Parameter, ♦ = terminal in constant/ pattern

Figure 18. Typing rules in normal form and refinement typing rule of Parameter

$$\frac{\Gamma \vdash b_1 : Idr \otimes nothing \quad \Gamma \vdash b : Par \otimes T \in \Gamma}{\Gamma \vdash \textbf{Bound}\ b_1 : Exp \otimes T}\ \text{BT-BoundParam}$$

$$\frac{\Gamma \vdash b_1 : Idr \otimes nothing \quad \Gamma \vdash b : Dec \otimes T \in \Gamma}{\Gamma \vdash \textbf{Bound}\ b_1 : Exp \otimes T}\ \text{BT-BoundDec}$$

$$\frac{\Gamma \vdash b_1 : Idr \otimes nothing \quad \Gamma, b_2 : Par \otimes T_1 \vdash b_3 : Exp \otimes T_2}{\Gamma \vdash \textbf{Function}\ b_1\ \textbf{Parameter}\ b_2\textbf{.}\ \textbf{Expression}\ b_3 : Dec \otimes T_1 \longrightarrow T_2}\ \text{BT-DecFunc}$$

$$\frac{\Gamma \vdash b_1 : Idr \otimes nothing \qquad \Gamma \vdash b_2 : Exp \otimes T}{\Gamma \vdash \textbf{Variable}\ b_1\ \textbf{bind to}\ b_2 : Dec \otimes T}\ \text{BT-DecVar}$$

Figure 19. Refinement typing rules

References

[1] Kenichi Asai. 2025. OCaml Blockly. *Journal of Functional Programming* 35, 12 (2025), e12. doi:10.1017/S0956796825000073

[2] Jack Atherton and Paulo Blikstein. 2017. Sonification Blocks: A Block-Based Programming Environment For Embodied Data Sonification. In *IDC '17: Proceedings of the 2017 Conference on Interaction Design and Children*. Association for Computing Machinery, New York, NY, United States, 733–736. doi:10.1145/3078072.3091992

[3] Atlassian. 2019. AutoBlocks for Jira (v2019). https://docs.adaptavist.com/pages/viewpage.action?pageId=101617186.

[4] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2013. A survey on reactive programming. *ACM Computing Survey* 45, 52 (2013), 1–34. Issue 4. doi:10.1145/2501654.2501666

[5] D. Bau. 2015. Droplet, A Block-based Editor for Text Code. *Journal of Computing Sciences in Colleges* 30, 6 (June 2015), 138–144.

[6] Tom Beckmann, Patrick Rein, Toni Mattis, and Robert Hirschfeld. 2022. Partial Parsing for Structured Editors. In *SLE 2022: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. Association for Computing Machinery, New York, NY, USA, 110–120. doi:10.1145/3567512.3567522

[7] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *CHI '23: Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3544548.3580785

[8] William R. Cook. 2013. Anatomy of Programming Language. https://www.cs.utexas.edu/~wcook/anatomy/anatomy.htm.

[9] Neil Fraser. 2015. Ten things we've learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, F. Turbak, D. Bau, J. Gray, C. Kelleher, and J. Sheldon (Eds.). 49–50. doi:10.1109/BLOCKS.2015.7369000

[10] T. R. G. Green. 1989. Cognitive Dimensions of Notations. In *In A. Sutcliffe and L. Macaulay (Eds.) People and Computers V*. Cambridge, UK: Cambridge University Press, 443–460.

[11] B. Harvey and J Mönig. 2010. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists?. In *Constructionism*. 1–10.

[12] J. Roger Hindley and Jonathan P. Seldin. 2008. *Lambda-Calculus and Combinators, an Introduction* (2nd ed.). Combridge University Press.

[13] John Huges. 1989. Why Functional Programming Matters. *Computer Journal* 32, 2 (1989), 98–107.

[14] Azusa Kurihara, Akira Sasaki, and Ken Wakita. 2015. A Programming Environment for Visual Block-Based Domain-Specific Languages. In *Proceedings of the 2015 International Conference on Soft Computing and Software Engineering (SCSE'15)*, Dr. Mehdi Bahrami (Ed.). doi:10.1016/j.procs.2015.08.452

[15] Sorin Lerner, Stephen R. Foster, and William G. Griswold. 2015. Polymorphic Blocks: Formalism-Inspired UI for Structured Connectors. In *33rd Annual CHI Conference 33rd Annual CHI Conference on Human Factors in Computing Systems*. CHI, Association for Computing Machinery, New York, NY, United States, 3063–3072. doi:10.1145/2702123.2702302

[16] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education* 10, 4 (2010), 1–15. doi:10.1145/1868358.1868363

[17] Mauricio Verano Merino, Tom Beckmann, Tijs van der Storm, Robert Hirschfeld, and Jurgen J. Vinju. 2021. Getting Grammars into Shape for Block-Based Editors. In *SLE'21: Software Language Engineering*. Association for Computing Machinery, 83–98. doi:10.5281/zenodo.5534113

[18] Mauricio Verano Merino and Jurgen J. Vinju. 2021. Getting Grammars into Shape for Block-Based Editors. In *SLE'20: Software Language Engineering*. Association for Computing Machinery, 283–295. doi:10.1145/3426425.3426948

[19] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978).

[20] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. MIT Press, London, England.

[21] Brad A. Myers. 1990. Taxonomies of visual programming and program visualization. *Journal of Visual Language and Computing* 1, 1 (1990), 97–123. doi:10.1016/S1045-926X(05)80036-9

[22] E. W. Patton, M. Tissenbaum, and F. Harunani. 2019. MIT App Inventor: Objectives, Design, and Development. In *Computational Thinking Education*, Siu-Cheung Kong and H. Abelson (Eds.). Springer, 31–39.

[23] Simon Peyton Jones. 2003. *Haskell 98 language and libraries: The revised report*. Cambridge University Press.

[24] Benjamin C. Pierce. 2002. *Types and Programming Language*. MIT Press, Cambridge, MA.

[25] Matthew Poole. 2019. A Block Design for Introductory Functional Programming in Haskell. In *2019 IEEE Blocks and Beyond Workshop*, M. Sherman and F. Turbak (Eds.). IEEE, Piscataway, NJ, 31–53. doi:10.1109/BB48857.2019.8941214

[26] Mitchel Resnick, John Maloney, Andres Monroy-Hernandex, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: programming for all. *Communications of The ACM* 52, 11 (2009). doi:10.1145/1592761.1592779

[27] R. V. Roque. 2007. *OpenBlocks: An Extendable Framework for Graphical Block Programming System*. Master's thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science.

[28] Andreas Rossberg. 2022. Standard ML Grammar. https://people.mpi-sws.org/~rossberg/sml.html.

[29] Scratch. 2022. Scratch for Developers. https://scratch.mit.edu/developers.

[30] Ben Selwyn-Smith, Craig Anslow, and Michael Homer. 2022. Blocks, Blocks, and More Blocks-Based Programming. In *PAINT 2022: Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*. Association for Computing Machinery, New York, NY, United States, 35–47. doi:10.1145/3563836.3568726

[31] Marie Vasek. 2012. *Representing Expressive Types in Blocks Programming Languages*. Master's thesis. Honors Thesis, Wellesley College.

[32] David Weintrop. 2019. Block-based programming in computer science education. *Commun. ACM* 62, 8 (2019), 22–25. doi:10.1145/3341221

[33] D. Weintrop, A. Afzal, J. Salac, P. Francis, B. Li, D. C. Shepherd, and D. Franklin. 2018. Evaluating CoBlox: A Comparative Study of Robotics Programming Environments for Adult Novices. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, United States, 1–12. doi:10.1007/978-981-13-6528-7

[34] David Wolber. 2011. App inventor and real-world motivation. In *SIGCSE'11*. 601–606. doi:10.1145/1953163.1953329

Received 2025-07-09; accepted 2025-08-11