# Mindwalls - A prototype

Enrique Urra

May 2019

## 1 The Brick Representation

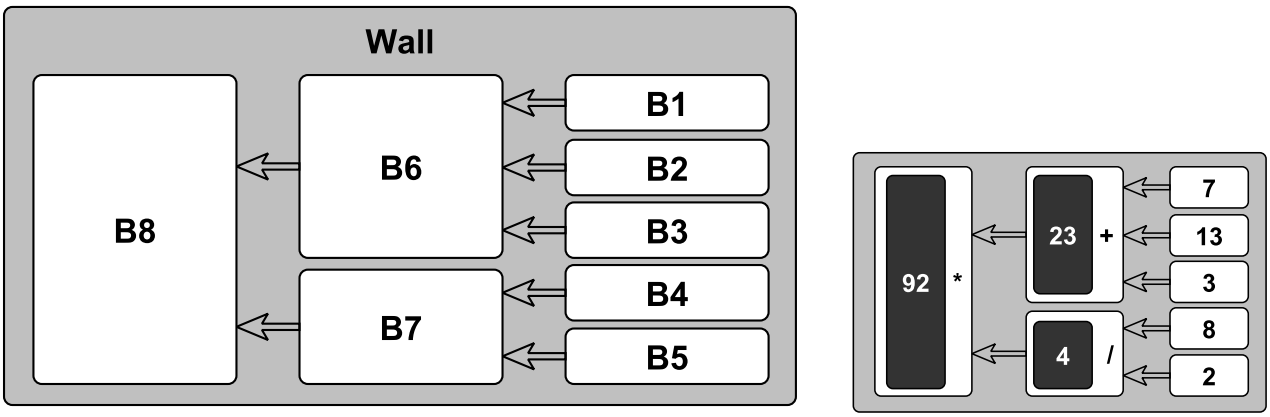


Figure 1: A simple view of how computations are represented in Mindwalls.

The leftmost part of figure 1 shows how computations are represented in Mindwalls. The most granular component is the *brick* (any *Bx* element). Bricks are simple *functional* entities, i.e., executable components that provide a single output given a specific input set. As any functional entity of other programming languages, bricks can cover a wide range of behavior *abstraction*. For example, there can exist specific bricks that always return the same static data with no inputs, and there can exist bricks that encapsulate complex computations with dynamic results depending on the input set.

Bricks can be arranged into a tree-like structure by using the outputs of several child bricks as input of a parent brick, thus, there are *terminal* (no childs) and *non-terminal* (at least one child) bricks. In figure 1, B1, B2 and B3 are terminal bricks and childs of B6, while B6, B7 and B8 are non-terminal bricks. In a *functional programming* style, a complete computation chain starts from the input of several terminal bricks (e.g., bricks whose output is static data) and finish with the output of the tree root (a brick at which all other computations converge). In Mindwalls, a single tree structure that produces an output at the root corresponds to a *wall*. In the figure 1, the output of B8 (the tree root) is computed from the results of B6 and B7 (its inputs), while the output of B7 is computed from the results of B4 and B5. An example of this computation is presented in the rightmost part of figure 1, using the same structure of the left, but implemented with simple arithmetic operators, whose intermediate results are presented as black boxes.

Despite this simple representation of computations in Mindwalls, which is based in well-known concepts of the functional programming world, there are several design challenges that should be considered to make this representation useful in the context of a visual language:

- **How bricks can be modularly combined and arranged within a visual workspace, in a way such that computations at different abstraction levels can be constructed and reused**. *Modularity* is a critical and cross-cutting topic in the programming world. It is the central premise in the proposal of design pattern catalogs, which have been greatly evolved in time [1]. As J. Hugues states in its seminal paper regarding functional programming [2], the main contribution of such languages is modularity capabilities through two different mechanisms: higher-order functions and lazy evaluation. Modularity

allows to combine the building blocks of programming languages for escalating from cohesive components towards complex and fully-featured software architectures. However, in this escalating process, *abstraction* is the property that allows to decouple such components, to adapt and to change them in a sustainable manner. Even if the general concept of bricks is based on functional premises, the challenge is to provide an efficient application of specific brick implementations (i.e., the building blocks) based on such general concept, and to effectively visualize relationships between these implementations with modularity and abstraction in mind.

- **How to manipulate bricks for building programs with the efficiency and flexibility that source code (text) could actually provide**. It is widely recognized that using text for programming provides several benefits for productivity, regarding visual means, such as lower viscosity (i.e., small changes becomes easier), lower space density, better navigation and source control [3], which allows an improved transition towards advanced programming skills [4, 5]. However, these observations born usually from the experience of visual languages whose interactivity is primarily based on explicit translations of text-based program components into visual means. For example, block-based visual languages only replicate the structure of text-based imperative languages, improving the visualization of control structures and other program components, however, manipulation of these components, commonly based on drag-and-drop, is not fluid and flexible as text. Therefore, the challenge is to find interactivity methods on a visual environment that better approach to the text-based programming experience, while preserving benefits that a visual environment should provide.

- **How to evaluate and improve a brick-based implementation through a minimum feedback loop possible**. It is common to associate difficulties for learning programming to a natural *geek gene* [6]. However, current research do not support this asseveration, and it gives more importance on the tools used when programming, and particularly, how they make less or more efficient the Read-Eval-Print Loop (REPL), i.e., the need of constantly jump from a process of editing code to a process of running tests over current code. Testing changes in code provides feedback of consistency, correctness and efficiency of the current program, allowing to continue on the next modifications or to improve the current ones. In text-based languages, this *feedback loop* is necessary bigger when the programmer has no tools for immediately visualize the impact of code changes. Consequently, the creative process in programming is constantly interrupted by testing tasks, and it requires enough abstraction skills to temporally track in mind the impact of code changes until they are actually tested. In the reality of current programming languages, training these skills is the *core* for evolving from a beginner to an advanced programming experience. However, this is also the main drawback of text-based programming languages, and several trends in language design explore alternatives. In its online article [7], Bret Victor explains in detail some principles, theoretically and prototyped, which are applicable on both the language and the programming environment for improving the feedback loop. Also, the term *live programming* has gained recognition [8] as the edition of code of a running program to immediately see the effects of code changes [9, 10]. In this line, the challenge is to gather several ideas from other domains to implement effective tools that could improve the feedback loop when working with bricks.

In the following sections, solutions for the above design challenges will be presented in the context of Mindwalls, which will serve as a roadmap for implementing a prototype of the language.

## 2 Brick types, relationships and visualizations

Any programming language must provide several *building blocks* that allow to build programs from scratch. These building blocks should be enough cohesive and enough abstract to combine them with flexibility and incremental complexity, starting from simpler structures and evolving towards sophisticate constructs. In Mindwalls, the brick concept provide a general framework in which all these building blocks are defined, i.e., all building blocks in Mindwalls are bricks in the end.

Figure 2 shows the building blocks in Mindwalls, how they are visually organized, how they relate and their interactions, which allow to build programs. From the most specific to the most general, these building blocks can be defined as follows:

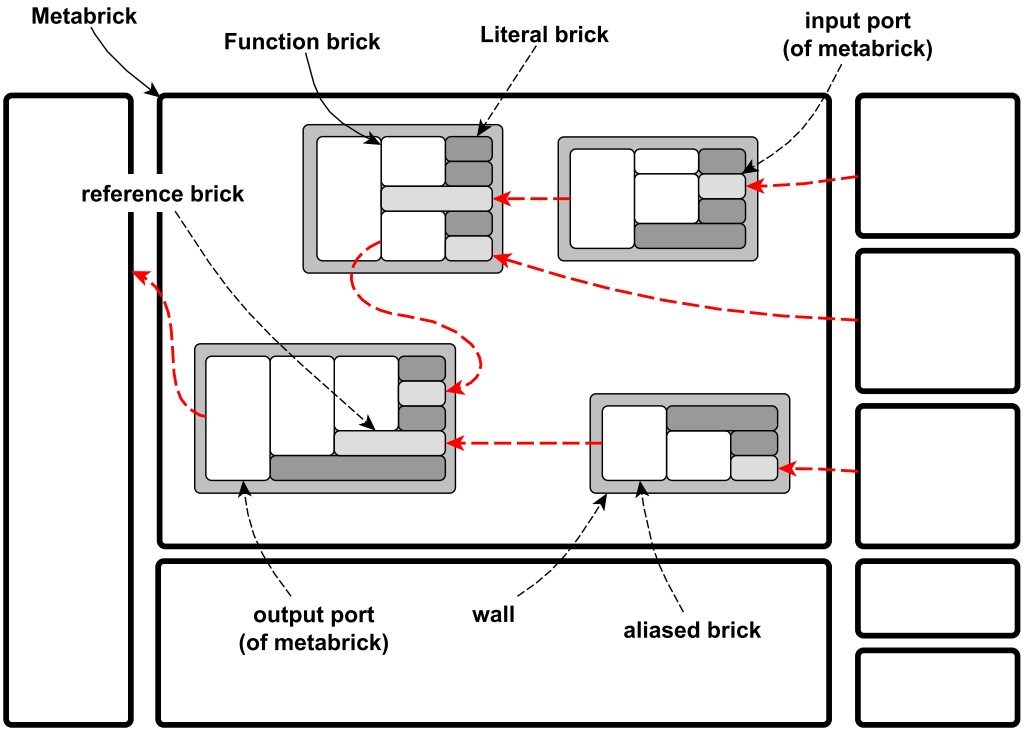


Figure 2: Building blocks of Mindwalls, and how they relate in a program structure.

- **Literal brick**: static data of any type (text, number, object, etc.) which could be used as input of other bricks.
- **Function brick**: computed brick that provides some output according to specific inputs.
- **Wall**: tree structure, composed by other inner bricks, whose root correspond to the brick at which all subcomputations converge. A wall is a brick itself, whose inputs correspond to the terminal bricks, and whose output is the value provided at the tree root.
- **Aliased brick**: it is a brick for which a reference name has been given, and whose output value (static or computed) can be referenced within the structure of any wall.
- **Reference brick**: it is a brick that references the value of an aliased brick, reusing its output value to be used within the structure of any wall.
- **Metabrick**: high-level brick, which encapsulate a set of walls that are related by references of their aliases. A metabrick is a brick itself, whose inputs correspond to specific *input ports* of specific walls within the metabrick, and whose output is the value provided by a specific *output port* of a specific wall within the metabrick. Properly abstracted, a metabrick can be part of the structure of a container wall in the form of a function brick.

In the following, the design intended for the above building blocks (plus other specific components) will be described, as a part of Mindwalls roadmap.

## 2.1 Literal bricks

Literal bricks are the most granular component of Mindwalls, allowing to specify static data that can be used as input of other bricks. Literal bricks do not use any input, and their corresponding output is the static data that they provide. Because of this, literal bricks are terminal bricks within the structure of a wall.

In terms of complexity, literal bricks can be any type of static data, such as strings, numbers, objects, etc. Figure 3 shows how this data could be visualized through literal bricks. Note that the type system of a lower-level language could be used as first implementation of Mindwalls, e.g., the use of Javascript-like objects rather than defining a custom object type.

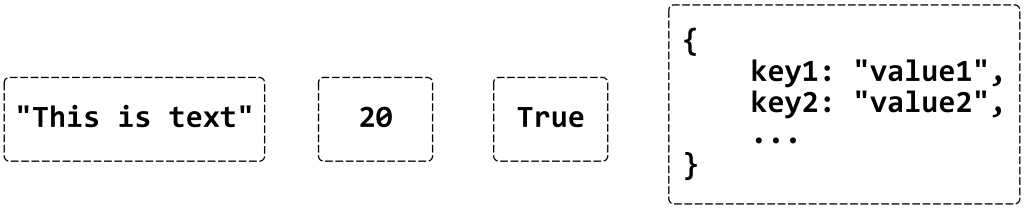


Figure 3: Literal bricks, which include strings, numbers, boolean values and objects (Javascript-like notation).

## 2.2 Function bricks

Function bricks are cohesive components that encapsulate specific code of a low-level language for implementing a simple computation based on specific inputs. They are non-terminal bricks within the structure of a wall, whose inputs can be literal bricks or other function bricks as well. Function bricks using the output of other function bricks as input is a straightforward implementation of higher-order functions.

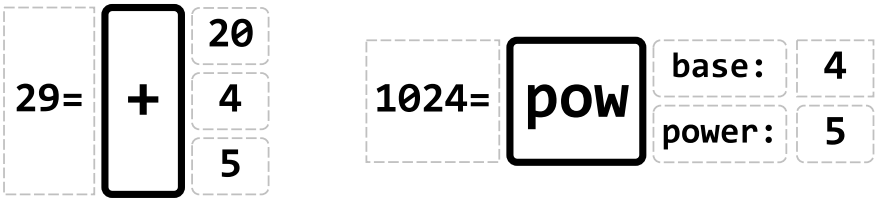


Figure 4: Two types of function bricks. The array function (left) returns 29 as output, and the map function (right) returns 1024 as output.

Figure 4 shows examples of function bricks. Visually, functions have their inputs at right, the function itself at middle, and the output at left. Two general function types are considered:

- **Array functions**: functions for which inputs are ordered and the order affects the output. Typical use cases for this type are simple arithmetic operators.
- **Map functions**: function for which the order of its inputs do not affect the output, but each input should be associated with a specific parameter name. This type covers a wide range of functions with both optional and required parameters. At the same time, parameter names are visible and explicit, allowing to arrange them in different order according to the programmer needs.

## 2.3 Walls

Walls are special bricks that result from the composition of multiple function bricks in a tree structure, using the higher-order function approach. They represent constrained and concrete computations that may be a part of a high-level process. As an individual building block, they are not meant to be abstracted into reusable components, however, they can be a specific part of a bigger abstraction, such as metabricks.

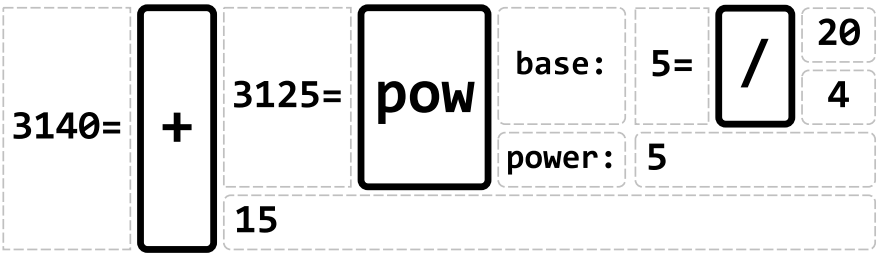


Figure 5: Example of wall, using several bricks from figure 4.

Figure 5 shows how walls are visualized, in which several function brick types have been composed to compute a final result (the leftmost output). This visualization is a trade-off between the compactness

provided by text representation such as '`pow(base: (20 / 4), power: 5) + 15`' or '`(+ (pow base:(/ 20 4) power:5) 15)`', and expressiveness of intermediate results and the data flow (right to left) produced by the composition of functions within the wall. In this way, the programmer has immediate feedback for identifying possible errors within the expression design, while in traditional text representations it is required to apply debugging tools and/or writing/capturing intermediate outputs in the code.

## 2.4 Brick aliasing and referencing

In imperative programming languages, the standard mechanism for storing intermediate data is the usage of *variables*. They allow to associate specific identifiers with temporal data, which can serve as input and/or output of different processes. Commonly, these variables should be defined and initialized at instructions different from the code in which they are actually used.

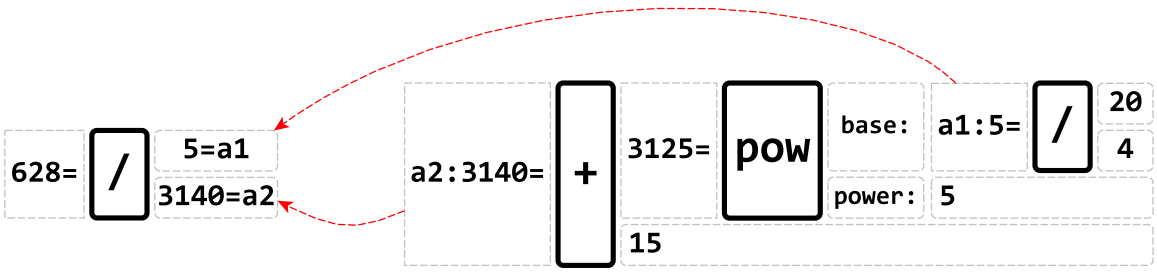


Figure 6: Example of how brick aliases and references are defined, used and how they produce wall connections.

In Mindwalls, variables are implemented as *aliases*. Rather than being defined and assigned at specific instructions, aliases are defined and assigned through brick outputs. Figure 6 shows how aliases are visually defined and used. Any brick output can be "tagged" with an alias, even those outputs which are an intermediate result of the complete wall process (e.g., the `a1` alias in the figure). A special type of brick, a *reference*, can be used as repository for the computed value of any alias. This implements a data flow between different walls, allowing to connect the outputs of a source wall to the inputs of a target wall.

## 2.5 Metabricks and ports

Any programming language should provide mechanisms for encapsulating complexity and abstracting it for its further reuse. This allows to create systems that escalate from granular components towards bigger architectures. With walls, it is possible to implement a specific computation that can be interconnected through aliases and references with other walls. However, a bigger number of walls should be necessary to build a medium-to-big size program. It is desirable to apply the *divide-and-conquer* philosophy against such incremental complexity, and to formalize specific inputs and outputs for each encapsulated component.

Mindwalls provide *metabricks* as a mechanism for encapsulation. Figure 7 shows a metabrick visualization, internally and externally. A metabrick is the main working space for the programmer, in which the user interface focuses. During the coding process, the programmer frequently will be switching between one metabrick and another, editing the encapsulated code on them. What is actually encapsulated, i.e., the internal view of a metabrick, is a set of walls which are interconnected through aliases and references. On the other side, the external view of a metabrick is a function brick, i.e., any metabrick can be abstracted and reused to the form of a function brick.

The interface of a metabrick to its external context is defined as a set of *ports*. There are *input ports* (`i1` and `i2` in the figure), whose outputs are values provided from the external context (such as function arguments), and there is an *output port* (`o` in the figure), an unique brick for which its output value is the output of the metabrick itself (such as a return value of a function).

Through the abstraction of a complete metabrick into a function brick, it is possible to reuse the metabrick implementation into other walls that are defined on a parent or higher-level metabrick. Mechanisms such as drill-up or drill-down (section **??**) allow to traverse this hierarchical organization of multiple metabricks for addressing different abstraction level complexities.

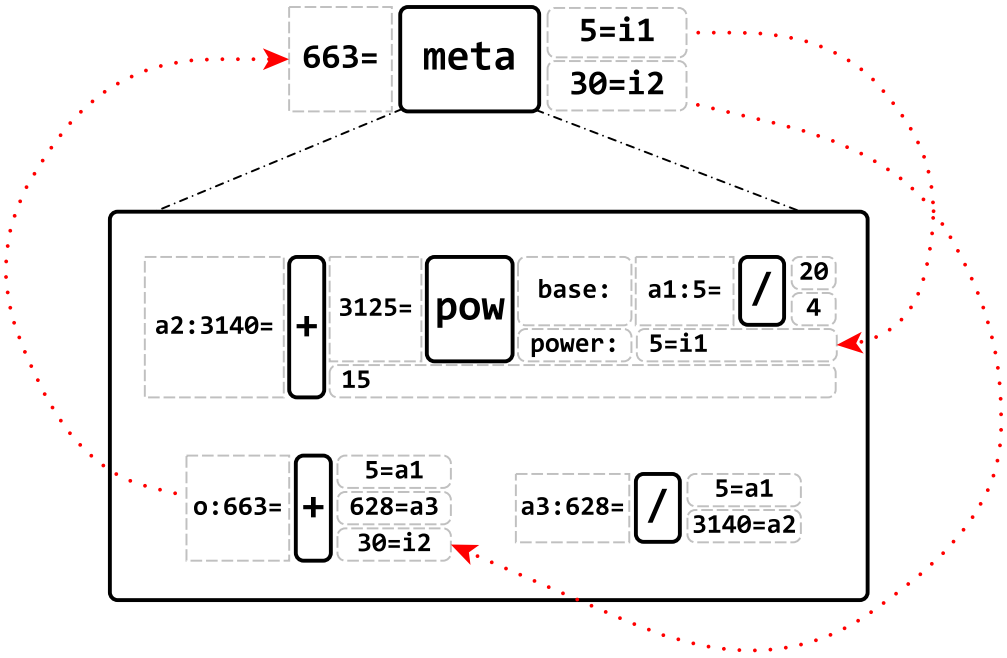


Figure 7: Example of a metabrick, visualized as a function brick (upper figure, external view) and as a set of encapsulated walls (lower figure, internal view).

## References


[1] B. B. Mayvan, A. Rasoolzadegan, and Z. G. Yazdi, "The state of the art on design patterns: A systematic mapping of the literature," *Journal of Systems and Software*, vol. 125, pp. 93 – 118, 2017.

[2] J. Hughes, "Why Functional Programming Matters," *The Computer Journal*, vol. 32, pp. 98–107, 01 1989.

[3] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. A. Turbak, "Learnable programming: Blocks and beyond," *CoRR*, vol. abs/1705.09413, 2017.

[4] D. Weintrop and U. Wilensky, "Comparing block-based and text-based programming in high school computer science classrooms," *ACM Trans. Comput. Educ.*, vol. 18, pp. 3:1–3:25, Oct. 2017.

[5] P. Sengupta, A. Dickes, A. V. Farris, A. Karan, D. Martin, and M. Wright, "Programming in k-12 science classrooms," *Commun. ACM*, vol. 58, pp. 33–35, Oct. 2015.

[6] N. C. C. Brown and G. Wilson, "Ten quick tips for teaching programming," *PLOS Computational Biology*, vol. 14, pp. 1–8, 04 2018.

[7] B. Victor, "Learnable programming," *Worrydream. com*, 2012.

[8] P. Rein, S. Ramson, J. Lincke, R. Hirschfeld, and T. Pape, "Exploratory and live, programming and coding: A literature study comparing perspectives on liveness," *CoRR*, vol. abs/1807.08578, 2018.

[9] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, "It's alive! continuous feedback in ui programming," *SIGPLAN Not.*, vol. 48, pp. 95–104, June 2013.

[10] S. McDirmid, "The promise of live programming," in *Proceedings of the 2nd International Workshop on Live Programming, LIVE*, vol. 16, 2016.