A Block Design for Introductory Functional Programming in Haskell

Matthew Poole
School of Computing
University of Portsmouth, UK
matthew.poole@port.ac.uk

Abstract—**This paper describes the visual design of blocks for editing code in the functional language Haskell. The aim of the proposed blocks-based environment is to support students' initial steps in learning functional programming. Expression blocks and slots are shaped to ensure constructed code is both syntactically correct and preserves conventional use of whitespace. The design aims to help students learn Haskell's sophisticated type system which is often regarded as challenging for novice functional programmers. Types are represented using text, color and shape, and empty slots indicate valid argument types in order to ensure that constructed code is well-typed.**

I. Introduction

Blocks-based environments such as Scratch [1] and Snap! [2] offer several advantages over traditional text-based languages for novice programmers. Such systems tend to emphasize imperative programming using custom languages. There exists some work on blocks-based editing for traditional imperative languages such as Python [3], [4] and Grace [5] which aims to support students learning programming within formal educational settings.

Many students, particularly those who choose to specialize in computing within higher education, face further challenges when learning a second or third programming language or paradigm. Several educators believe that learning functional programming is an important step in the development of computer scientists and software engineers [6], [7]. Knowledge of functional programming can be seen as an important practical skill given the growing number of languages that emphasize the functional approach (e.g., Clojure, F#, Scala) or which include functional constructs (e.g., Python, JavaScript, Java).

Haskell is often regarded as a good language for learning functional programming due to its clean syntax and functional purity. Haskell is not a simple language however; its type system (particularly its concept of type classes) is quite complex, and compiler error messages (e.g., those of the popular Glasgow Haskell Compiler) are often very difficult for beginners to comprehend. Combined with the efforts required to learn a new paradigm, these issues can lead students to struggle. Helium [8], a subset of Haskell for learning functional programming, was designed to alleviate these problems. Helium does not include type classes and is focused on providing hints, warnings and improved error messages for the learner.

Instead of defining a simplified language or compiler, the approach taken here is to expose Haskell's type system to the learner, and to avoid syntax and type-based errors entirely through blocks-based program construction.

There exists some recent work in representing functional types within blocks-based environments. TypeBlocks [9] includes three basic type connector shapes (for lists, tuples and functions) which can be combined in any way and to any depth. The prototype blocks editor for Bootstrap [10], [11] represents each of the five types of a simple functional language using a different color, with a neutral color (gray) used for polymorphic blocks; gray blocks change color once their type has been determined during program construction. Some functional features have also been added to Snap! [12] and to a modified version of App Inventor [13].

This paper is structured as follows. In the next section we give a brief overview of the core of the Haskell language relevant to the block design ideas, which are then presented in Section III. Section IV concludes the paper and discusses future work.

II. Overview of Haskell

Haskell is a statically-typed pure functional language (there is no state and no side-effects, and functions can be passed to and returned from other functions). Haskell also features type inference; one of the consequences of this is that programmers are rarely required to explicitly assign types to their function definitions (and we will not consider type declarations here).

Haskell supports (parametric) polymorphism: expressions (including functions) can take more than one type. Using type variables (typically `a`, `b`, `c`, ...) to represent polymorphic types, we state the type of (say) the identity function `id` to be `a -> a` (written `id :: a -> a`). Here, `a` can represent any type, so `id` can be used, for example, in a context requiring `Int`s (to give `Int -> Int`) or `Char`s (to give `Char -> Char`).

Functions in Haskell are *curried*; all functions are considered to take a single argument. A function `f :: a -> (b -> c)`, more conventionally written `f :: a -> b -> c`, takes a single argument of type `a` and returns a function of type `b -> c`. Curried functions can be partially applied; for `x :: a`, the partial application `f x` is of type `b -> c`. A 'full' application is written `f x y` (where `y :: b`) and is of type `c`.

Haskell supports overloading (or ad-hoc polymorphism) through the use of type classes. A type class in Haskell can be viewed as a set of types which share certain operations;

the operations supported by a type class are known as its methods. A type can be declared as being an instance of a type class; an instance declaration includes type-specific (overloaded) definitions of each of the type class's methods. The Haskell standard library (the Prelude) includes definitions of many type classes and instances. For example, the `Num` type class includes the numeric operators '+', '−' and '⋆' as its methods. The types of each of these operators is `Num a => a -> a -> a`, which can be read as: "for every type `a` that is an instance of class `Num`, the operator has type `a -> a -> a`. The Prelude defines all numeric types (`Int`, `Float`, etc.) to be instances of class `Num`, each by giving overloaded definitions of these three operators; e.g., the `Int` instance defines them has integer operations of type `Int -> Int -> Int`.

Type classes can be related hierarchically, such that for a type to be declared as an instance of a subclass, it must also be an instance of the superclass. For example, the `Fractional` type class is a subclass of `Num` and includes the extra method '/' (for floating point division). Types `Float` and `Double` are declared as instances of `Fractional`, but `Int` is not.

III. Block Design

The main aims of the design are to represent fully all expression blocks' types, and for constructed code to be formatted according to convention.

A. *Basic and polymorphic types*

For expressions of 'basic' types (non-parameterized, non-tuple and non-function types), we label blocks using the name of its type together with a color. Fig. 1(a) gives the labels for Haskell's numeric, character and Boolean types. Type variables `a`, `b`, `c`, ... that represent any (polymorphic) type are not used in labels; instead they are replaced by gray hatch patterns such as those in Fig. 1(b). The intention is that where the same hatch pattern occurs within a set of connected blocks, it represents any common type; different hatch patterns represent potentially different types.

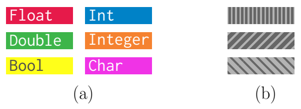

Fig. 1. Labels for (a) some of Haskell's basic and (b) polymorphic types.

B. *Block shape and behavior*

The basic element of a definition of a function `f` in Haskell is the *equation* that takes the form `f` $p_1 \ldots p_n$ = `r`, where each p_i is a *pattern* and `r` is the *result*. Some expressions (and therefore blocks) are syntactically valid only within patterns, others only in results, and some can appear in both patterns and results. We use different slot and block shapes to enforce construction of syntactically valid Haskell code.

Fig. 3 includes an equation for defining a function `f :: Shape -> Bool`, which assumes the existence of a `Shape` type. For this paper, we don't consider how `Shape` is defined, how the type of `f` is determined, or how equations are created and combined; the focus here is on the shape of the empty pattern and result slots and the 'type indicators' within these slots. The angled upper corners of the slots determine which varieties of blocks are syntactically legal, and the indicators determine valid types; both the block shape and type need to be compatible with the slot for a drop to be accepted.

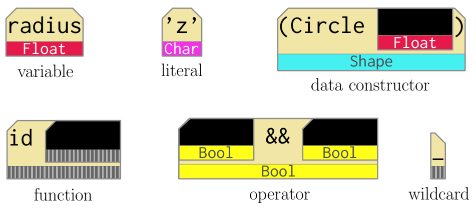

Fig. 2. Six varieties of blocks with angled corners and type labels that determine where they can be dropped.

Fig. 2 gives examples of blocks for six varieties of Haskell expressions. The top three blocks are angled on both upper corners, and can be dropped into any slot (those angled at the left or right and those not angled) if the type indicators match. The function and operator blocks cannot be dropped into slots angled on the right (i.e. they can only be used in results), and the wildcard block cannot be dropped into any slot angled at the left (it can only be used in patterns).

Fig. 3 illustrates the effects of dropping blocks into slots. The `radius` block is dropped legally into the unangled argument slot of the `Circle` block since the types match. Blocks take the shapes of the slots they are dropped into in order to show in what context they are being used, and so the angles disappear from the `radius` block. When the `Circle` block is then dropped into the left-hand slot of the equation it is reshaped to match the slot (it is now being used as part of a pattern). `Circle`'s (filled) argument slot is similarly reshaped as is the `radius` block it contains (since all elements of a pattern must themselves be patterns). If a block is removed from a slot it reverts to its original shape.

Note that parentheses are automatically added and removed as necessary. The `Circle` data constructor block includes parentheses initially as they are required in most contexts and they provide space for the block angle on the right; in some contexts (e.g. as a result) the parentheses and angled right corner disappear. Labeling types at the bottom of blocks will cause deeply nested code to be quite tall; however, this isn't considered a major issue since functions are usually defined using very few lines of code.

There are two polymorphic blocks in Fig. 2: the identity function `id :: a -> a`, and the wildcard '_', which is used in patterns to match any value and is assumed to be of type `a`. Fig. 4 illustrates the behavior of polymorphic blocks. In Fig. 4(a) the wildcard block is dropped into the left-hand (pattern) slot of an equation; this is valid since the block and slot shapes match and gray hatching matches any type. The wildcard's type changes to that of the slot, and we see here that the type name is abbreviated to preserve conventional code formatting. In Fig. 4(b), the `'z'` block is dropped into the argument slot of `id`; since the argument and result type are common (they are hatched the same way), both are recolored to match the type (`Char`) of the dropped block. The `'z'` block also changes shape to match its containing slot (it is now being used as part of a result expression).

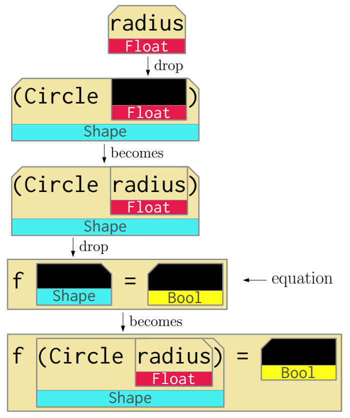

Fig. 3. Dropping blocks to form the left-hand-side (pattern) of an equation.

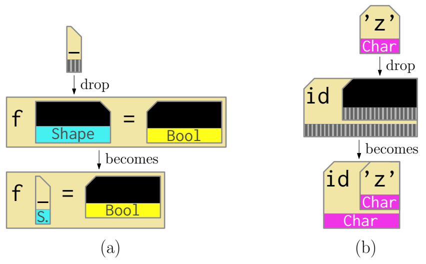

Fig. 4. Behavior of polymorphic blocks: (a) a wildcard block is dropped and retyped to `Shape`; (b) a `Char` block becomes the argument of `id`, which changes its type to `Char -> Char`.

C. Parameterized Types

Types in Haskell can be parameterized by one or more type parameters. One commonly used parameterized type is `Maybe a` which represents optional values; a value of type `Maybe a` contains either a value of type `a` (represented by the data constructor `Just a`) or is empty (`Nothing`). For example, the expression `Just 'z'` is of type `Maybe Char`. The `Either a b` type represents values with one of two possibly different types, using data constructors `Left a` or `Right b`. For example, the expression `Right 'z'` is of type `Either a Char`.

Fig. 5 shows how parameterized types can be represented, using color for the types themselves and with embedded areas above and to the right for the types of the parameter(s).

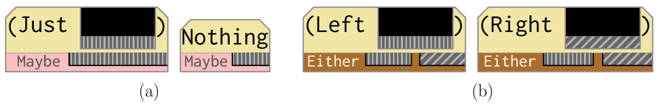

Fig. 5. Blocks for the data constructors of types (a) `Maybe a` and (b) `Either a b`.

D. Type classes

Labels for polymorphic blocks and slots whose type is constrained by a type class are colored gray and include the name of the type class involved. They are thus distinguished both from types (which use color) and from unconstrained polymorphic types (which use gray hatching).

Fig. 6 illustrates the labels and behaviors of blocks that involve type classes. The expression 3.14 in Haskell is of type `Fractional a => a` which is represented using the text `Fractional` on a gray background. Note that extra whitespace has been added around the value to allow the type class name to be given in full since there is no need to abbreviate type labels in disconnected blocks. This block is dropped into an argument slot of the '`*`' block (the operator '`*`' is of type `Num a => a -> a -> a`). Since `Fractional` is a subclass of `Num`, through type inference, the '`*`' block is retyped to `Fractional a => a -> a -> a`. Because `Float` is an instance of the type class `Fractional`, dropping the `radius` block into the remaining empty slot is permitted and leads, again through type inference, to the types of '`*`' and 3.14 becoming `Float -> Float -> Float` and `Float`, respectively.

It should be noted that the visual representations alone do not attempt to show how type classes are related or which types are members of which type classes. It would be clearly desirable for the system to highlight which slots are valid targets for any block selected by the user.

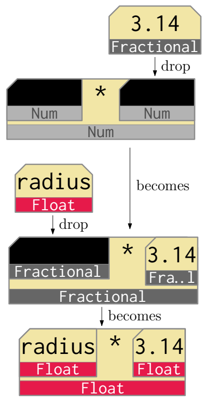

Fig. 6. Behavior of blocks featuring the type classes `Num` and `Fractional`.

E. Lists and Tuples

Lists are the fundamental collection type in functional languages. Lists in Haskell are homogeneous collections; a list of elements of type `a` has type denoted by `[a]`. Lists can be written with elements given between brackets `[···]`, and they can be built using the empty list `[]` (of type `[a]`) and the prepend (cons) operator '`:`' of type `a -> [a] -> [a]`. Lists are a parameterized type, and this is reflected in the shape of their visual representation. However, instead of a color and type name, an area of white is used, with the element type embedded in the upper-right region.

Fig. 7 shows a block construction of lists using the `[···]` notation and the '`:`' operator. Notice from the shapes of the blocks that both can be used within patterns (they are considered as data constructors), and that the `[···]` block includes controls for adding and removing slots for elements.

Tuples in Haskell are heterogeneous collections, typically containing only 2 or 3 elements; an example tuple value is `(True, 'a')` and this has type `(Bool, Char)`. Tuple types are represented as illustrated in Fig. 8 with a narrow white base and with curved white region(s) separating the

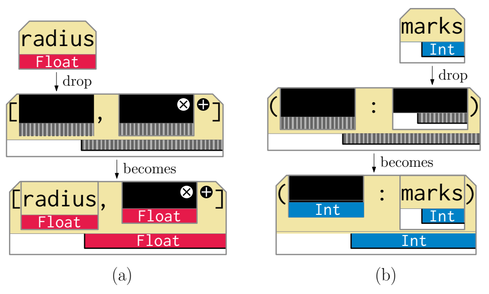

Fig. 7. (a) A list of type `[Float]` constructed using a `[···]` block; (b) a list of type `[Int]` constructed using a `‘:’` block.

component types. Note the control for adding extra slots (there is no control for removing the second slot since Haskell does not allow tuples of one element).

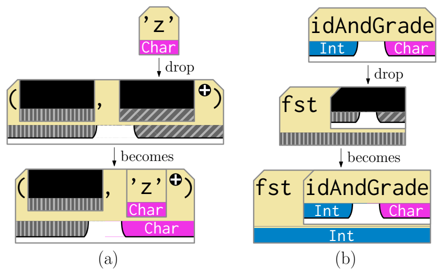

Fig. 8. (a) A `Char` block is dropped into an empty slot of a tuple block; (b) a variable block of type `(Int, Char)` is dropped into the argument slot of the projection function `fst :: (a, b) -> a`.

The visual representations for lists (and other parameterized types) and tuples clearly allow for nesting of types. Fig. 9 illustrates nestings of lists and of lists with tuples. Nesting can potentially be to any depth but, of course, representations could become difficult to interpret, especially for blocks of limited width. These issues could be alleviated by, for example, adding extra whitespace around a block's text (breaking formatting conventions) and by enforcing minimum lengths of identifier names. For most reasonable block constructions, however, the representations should be large enough to be clear.

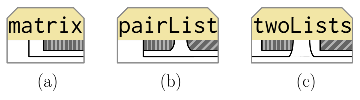

Fig. 9. Nested lists/tuple variable blocks of type (a) `[[a]]`, (b) `[(a, b)]`, and (c) `([a],[b])`.

F. Function types

Function types are represented with a white base and arrow separating the argument and result types. We illustrate this representation in Fig. 10 using the blocks for the function composition operator ‘`.`’ (of type `(b -> c) -> (a -> b) -> a -> c`) and three commonly used higher-order functions: `map :: (a -> b) -> [a] -> [b]`, `filter :: (a -> Bool) -> [a] -> [a]` and `foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b` from the Haskell Prelude.

The first argument slot of the `foldr` block shows the nesting of two functions, clearly illustrating that the type `a -> b -> b` should be interpreted as `a -> (b -> b)`. The third argument includes the `Foldable` type class of which lists are instances; so, for example, the `marks` variable block of Fig. 7(b) could be dropped here (and the shapes of the indicator and block type representations suggest this).

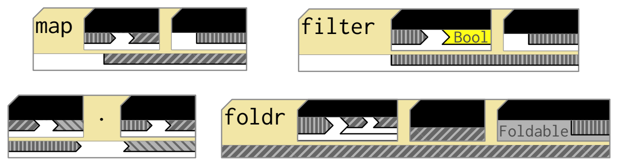

Fig. 10. Blocks for the function composition operator ‘`.`’ and the higher-order functions `map`, `filter` and `foldr`.

G. Partial function application

Partial function application is achieved through the addition of controls that allow functions' argument slots to be removed. In Fig. 11, the second slot of the `map` block is removed, resulting in a block of one slot (with controls for re-adding the second slot and for removing the remaining (first) slot). An operator such as ‘`*`’ can also be be partially applied by removing either or both of its arguments to give an *operator section*. In Fig. 11, the left argument slot is removed and the remaining slot is filled with the `factor` variable block to give the operator section `(* factor)` of type `Int -> Int`. Dropping this block into `map`'s empty slot gives a complete block for `map (* factor)` (of type `[Int] -> [Int]`). Note that the controls on the ‘`*`’ block disappear once it is dropped.

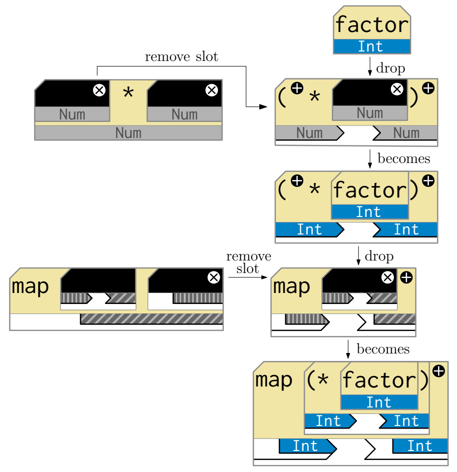

Fig. 11. Construction of the expression `map (* factor)` through partial application of ‘`*`’ and `map`.

IV. Conclusion and Further Work

We have presented a design for blocks that represent expressions in Haskell. The design ideas are intended to form the basis of an environment to help students to understand Haskell's type system and to scaffold their learning of functional programming. The next steps in the design are to include mechanisms for defining functions (most notably local definitions and guards), function type declarations and (algebraic) type definitions.

References

[1] "Scratch," scratch.mit.edu, accessed 11 July 2019.

[2] "Snap!" snap.berkeley.edu, accessed 11 July 2019.

[3] M. Poole, "Design of a blocks-based environment for introductory programming in Python," in *2015 Blocks and Beyond Workshop*. IEEE, 2015, pp. 31–34.

[4] ——, "Extending the design of a blocks-based Python environment to support complex types," in *2017 Blocks and Beyond Workshop*. IEEE, 2017, pp. 1–7.

[5] M. Homer and J. Noble, "Combining tiled and textual views of code," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on Software Visualization*. IEEE, 2014, pp. 1–10.

[6] J. Hughes, "Why functional programming matters," *The Computer Journal*, vol. 32, no. 2, pp. 98–107, 1989.

[7] Z. Hu, J. Hughes, and M. Wang, "How functional programming mattered," *National Science Review*, vol. 2, no. 3, pp. 349–370, 2015.

[8] B. Heeren, D. Leijen, and A. van IJzendoorn, "Helium, for learning Haskell," in *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*. ACM, 2003, pp. 62–71.

[9] M. Vasek, "Representing expressive types in blocks programming languages," Wellesley College, Honors thesis, 2012.

[10] "Bootstrap block editor," bootstrap-block-editor.appspot.com, accessed 11 July 2019.

[11] E. Schanzer, S. Krishnamurthi, and K. Fisler, "Blocks versus text: Ongoing lessons from Bootstrap," in *2015 Blocks and Beyond Workshop*. IEEE, 2015, pp. 125–126.

[12] B. Harvey and J. Mönig, "Lambda in blocks languages: Lessons learned," in *2015 Blocks and Beyond Workshop*. IEEE, 2015, pp. 35–38.

[13] S. Kim and F. Turbak, "Adapting higher-order list operators for blocks programming," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2015, pp. 213–217.