The Practice of a Compositional Functional Programming Language*

Timothy Jones[1] and Michael Homer[2]

[1] Montoux, New York, NY, USA
tim@montoux.com
[2] Victoria University of Wellington, Wellington, New Zealand
mwh@ecs.vuw.ac.nz

Abstract. Function composition is a very natural operation, but most language paradigms provide poor support for it. Without linguistic support programmers must work around or manually implement what would be simple compositions. The Kihi language uses only composition, makes all state visible, and reduces to just six core operations. Kihi programs are easily stepped by textual reduction but provide a foundation for compositional design and analysis.

Keywords: function composition · concatenative programming.

1 Introduction

Programming languages exist in many paradigms split along many different axes. For example, there are imperative languages where each element of the code changes the system state somehow before the next step (C, Java); there are declarative languages where each element of the code asserts something about the result (Prolog, HTML); there are functional languages where each element of the code specifies a transformation from an input to an output (Haskell, ML, Forth). Functional languages can be further divided: there are pure functional languages (Haskell), and those supporting side effects (ML). There are languages based on applying functions (Haskell) and on composing them (Forth).

It is composition that we are interested in here. Forth is a language where the output or outputs of one function are automatically the inputs of the next, so a program is a series of function calls. This family is also known as the *concatenative languages*, because the concatenation of two programs gives the composition of the two: if `xyz` is a program that maps input A to output B, and `pqr` is a program that maps B to C, then `xyzpqr` is a program that maps A to C. Alternatively, they can be analysed as languages where juxtaposition of terms indicates function composition, in contrast with applicative functional languages like Haskell where it indicates function application.

Many concatenative languages, like Forth, are stack-based: operations push data onto the stack, or pull one or more items from it and push results back on. This is sometimes regarded as an imperative mutation of the stack, but functions

* This is a post-peer-review, pre-copyedit version of an article published in Springer LNCS. The final authenticated version is available online at: https://doi.org/10.1007/978-3-030-02768-1_10

in these languages can also be regarded as unary transformations from a stack to another stack. Stack-based languages include Forth, PostScript, RPL, and Joy, and other stack-based systems such as Java bytecode can be (partially) analysed in the same light as well. Most often these languages use a postfix syntax where function calls follow the operations putting their operands on the stack.

Concatenative, compositional languages need not be stack-based. A language can be built around function composition, and allow programs to be concatenated to do so, without any stack either implicit or explicit. One such language is Om [2], which uses a prefix term-rewriting model; we present another here.

In this paper we present Kihi, a compositional prefix concatenative functional language with only six core operations representing the minimal subset to support all computation in this model, and a static type system validating programs in this core. We also present implementations of the core, and of a more user-friendly extension that translates to the core representation at heart.

A Kihi program consists of a sequence of terms. A term is either a (possibly empty) parenthesised sequence of terms (an *abstraction*, the only kind of value), or one of the five core operators:

- **apply**, also written $\cdot$: remove the parentheses around the subsequent abstraction, in effect splicing its body in its place.
- **right**, or $\rightarrow$: given two subsequent values, insert the rightmost one at the *end* of the body of the left. In effect, partial application of the left abstraction.
- **left**, or $\leftarrow$: given two subsequent values, insert the rightmost one at the *start* of the body of the left. A "partial return" from the first abstraction.
- **copy**, or $\times$: copy the subsequent value so that it appears twice in succession.
- **drop**, or $\downarrow$: delete the subsequent value so that it no longer appears in the program.

These operations form three dual pairs: abstraction and apply; right and left; copy and drop. We consider abstraction an operation in line with these pairings.

At each step of execution, an operator whose arguments are all abstractions will be replaced, along with its arguments, with its output. If no such operator exists, execution is stuck. After a successful replacement, execution continues with a new sequence. If more than one operator is available to be reduced, the order is irrelevant, as Kihi satisfies Church-Rosser (though not the normalisation property), but choosing the leftmost available reduction is convenient.

This minimal core calculus is sufficient to be Turing-complete. We will next present some extensions providing more convenient programmer facilities.

2 Computation

Combined with application, the left and right operators are useful for shuffling data in and out of applications. The left operator in particular is useful for reordering inputs, since each subsequent use of $\leftarrow$ moves a value to the left of the value that it used to be to the right of. The `swap` operation, which consumes two values and returns those values in the opposite order, can be defined from the

core operators as · ← ← `()`. For instance, executing `swap x y` reduces through the following steps: · ← ← `() x y` ⟶ · ← `(x) y` ⟶ · `(y x)` ⟶ `y x`.

The `under` operation · ← executes an abstraction "below" another, preserving its second argument for later use and executing the first with the remaining program as its arguments. The flexibility of `under` demonstrates the benefit of a compositional style over an applicative one. We do not need to reason about the number of inputs required by the abstraction, nor the number of return values: it is free to consume an arbitrary number of values in the sequence of terms, and to produce many values into that sequence as the result of its execution.

As Kihi is a compositional language, composing two operations together is as simple as writing them adjacently. Defining a composition operator that consumes two abstractions *as inputs* and returns the abstraction representing their composition is more involved, since the resulting abstraction needs to be constructed by consuming the abstractions into the output and then manually applying them. The `compose` operation is defined as → → `(· under (·))`. This operation brings two abstractions into the abstraction defined in the operation, which will apply the rightmost first and then the left. The leftmost abstraction can consume outputs from the rightmost, but the right cannot see the left at all.

2.1 Data Structures

Abstractions are the only kind of value in Kihi, but we can build data structures using standard Church-encodings. In the Church-encoding of booleans, `true` and `false` both consume two values, with `true` returning the first and `false` the second. In Kihi, `false` is equivalent to `(↓)`: since the drop operation removes the immediately following value, the value that appears after that (in effect, the second argument) is now at the head of the evaluated section of the sequence. The definition of `true` is the same, but with a swapped input: `(↓ swap)`.

The definition of standard boolean combinators like `and` and `or` each involve building a new abstraction and moving the boolean inputs into the abstraction so that, when applied, the resulting abstraction behaves appropriately as either a `true` or `false` value. For instance, `or` can be defined as → → `(· · swap true)`. The result of executing · `or x y` is · · `x true y`: if `x` is true, then the result is an application of `true`, otherwise the result is an application of `y`.

In the Church-encoding of the natural numbers, a number n is an abstraction that accepts a function and an initial value, and produces the result of applying that function to its own output n times. In this encoding, `zero` is equivalent to `false`, since the function is ignored and the initial value is immediately returned. In Kihi, the Church-encoding of the successor constructor `suc` is → `(· under (·) swap under (×))`. For an existing number `n` and a function `f`, executing · `suc n f` produces the sequence · `f · n f`: apply `n` to `f`, then apply `f` once more to the resulting value. Once again, the function can be flexible in the number of inputs and outputs that it requires and provides: so long as it provides as many as it requires, it will perform a reduction with a constant number of inputs. For an unequal number of inputs to outputs, the function will dynamically consume or generate a number of values proportional to the natural number that is applied.

2.2 Recursion

To be Turing-complete, the language must also support recursion. The recursion operation `Y` is defined in Kihi as `→ × → (· under (→ → (·) ×))`. For any input `f`, executing `Y f` first produces the abstraction `(· under (→ → (·) ×) f)`, and then copies it and partially applies the copy to the original, producing the abstraction `(· under (→ → (·) ×) f (· under (→ → (·) ×) f))`. Applying this abstraction ultimately produces an application of `f` to the original abstraction: `· f (· under (→ → (·) ×) f (· under (→ → (·) ×) f))`. Once again, `f` is free to consume as many other inputs after its recursive reference as it desires, and can also ignore the recursive abstraction as well.

2.3 Output

Operators cannot access values to their left, so a value preceded by no operators can never be modified or affected later in execution. As a result, any term that moves to the left of all remaining operators is an output of the program. Similarly, any program can be supplied inputs on the right. A stream processor is then an infinite loop, consuming each argument provided on its right, transforming the input, and producing outputs on its left: a traditional transformational pipeline is simply a concatenation of such programs with a data source on the end.

A program (or subprogram) can produce any number of outputs and consume any number of inputs, and these match in an arity-neutral fashion: that is, the composition does not require a fixed correspondence between producer and consumer. It is *not* the case that all outputs of one function must be consumed by the same outer function, as is usually the case when construction a compositional pipeline in imperative or applicative languages.

3 Name Binding

The core calculus of Kihi does not include variables, but the language supports name binding by translation to the core. The `bind` form takes as its first argument syntax that defines the name to bind.

```
bind (x) (t ...) value
```

The name `x` is bound to the value `value` inside the term `(t ...)`, which is then applied. For the translation to make sense as a compile-time transformation, the name and body must be present in their parenthesised form in the syntax, but the `value` does not need to be present; a `bind` may appear inside an abstraction with no input as in `(bind (x) (t ...))`, in which case the bound value will be the first input of the abstraction.

The transformation brings the bound value leftwards, jumping over irrelevant terms, and leaving a copy behind wherever the bound name occurs. To translate a `bind` form to the core, for each term `t` inside `(t ...)`:

$$a ::= (t \ ...)$$
$$t ::= \cdot \mid \leftarrow \mid \rightarrow \mid \downarrow \mid \times \mid a$$

Fig. 1. Redex Language Definition

$$t ::= .... \mid \mathsf{bind} \mid x$$

Fig. 2. Redex Binding Extension

$$(t \ ... \cdot (t_t \ ...) \ v \ ...) \longrightarrow (t \ ... \ t_t \ ... \ v \ ...) \quad \text{[Apply]}$$
$$(t \ ... \leftarrow (t_t \ ...) \ v_t \ v \ ...) \longrightarrow (t \ ... \ (v_t \ t_t \ ...) \ v \ ...) \quad \text{[After]}$$
$$(t \ ... \rightarrow (t_t \ ...) \ v_t \ v \ ...) \longrightarrow (t \ ... \ (t_t \ ... \ v_t) \ v \ ...) \quad \text{[Before]}$$
$$(t \ ... \downarrow v_d \ v \ ...) \longrightarrow (t \ ... \ v \ ...) \quad \text{[Drop]}$$
$$(t \ ... \times v_c \ v \ ...) \longrightarrow (t \ ... \ v_c \ v_c \ v \ ...) \quad \text{[Copy]}$$

Fig. 3. Redex Reduction Relation

1. If `t` is the name `x` to be bound, replace it with $\times$, to leave one copy of the value behind in its place and another to continue moving left.
2. If `t` is an abstraction `v`, replace it with `swap` $\rightarrow$ `(bind (x) v)` $\times$ and then expand the resulting `bind` form, to bind a copy of the value in `v` and swap the original value to the other side of the abstraction.
3. Otherwise replace `t` with $\cdot \leftarrow$ `(t)`, to 'jump' the value leftwards over the operator.

Finally, prepend $\downarrow$ to delete the final copy of the value, and remove the parentheses. Translate nested binds innermost-outwards to resolve shadowing.

4 Implementations

Kihi has been implemented as mechanisation of the semantics, a practical Racket language, and a web-based tool that visualises executions.

4.1 Redex

An implementation of Kihi's core calculus in the Redex [3] semantics language is presented in Figure 1. The syntax corresponds to the syntax we have already presented. The reduction rules for this language are shown in Figure 3. The semantics presented here proceeds right-to-left: this can easily be made unordered by matching on t instead of v on the right side of each rule. When the semantics are unordered, the Redex procedure `traces` shows every possible choice of reduction at each step, ultimately reducing to the same value (or diverging).

The binding language extension is also encoded into Redex, with syntax defined in Figure 2. The `expand` translation from this language to the original calculus is defined in Figure 4. A malformed bind will produce a term that is not a valid program in the original calculus.

$$
\begin{aligned}
&\textsf{expand} : t \to t \\
&\textsf{expand}[\![(\textsf{bind}\ (x)\ v\ t_{tail}\ ...)]\!] = (\downarrow\ t_{bound}\ ...\ t_{cont}\ ...) \\
&\quad \text{where } (t_{bound}\ ...) = \textsf{bind-body}[\![x, \textsf{expand}[\![v]\!]]\!], (t_{cont}\ ...) = \textsf{expand}[\![(t_{tail}\ ...)]\!] \\
&\textsf{expand}[\![(t\ t_{tail}\ ...)]\!] = (\textsf{expand}[\![t]\!]\ t_{cont}\ ...) \\
&\quad \text{where } (t_{cont}\ ...) = \textsf{expand}[\![(t_{tail}\ ...)]\!] \\
&\textsf{expand}[\![t]\!] = t \\
&\textsf{bind-body} : x\ v \to v \\
&\textsf{bind-body}[\![x, (t\ t_{tail}\ ...)]\!] = (t_{bound}\ ...\ t_{cont}\ ...) \\
&\quad \text{where } (t_{bound}\ ...) = \textsf{bind-name}[\![x, t]\!], (t_{cont}\ ...) = \textsf{bind-body}[\![x, (t_{tail}\ ...)]\!] \\
&\textsf{bind-body}[\![x, ()]\!] = () \\
&\textsf{bind-name} : x\ t \to (t\ ...) \\
&\textsf{bind-name}[\![x, x]\!] = (\times) \\
&\textsf{bind-name}[\![x, v]\!] = (\textsf{swap}\ \to (\downarrow\ t\ ...)\ \times) \\
&\quad \text{where } (t\ ...) = \textsf{bind-body}[\![x, v]\!] \\
&\textsf{bind-name}[\![x, t]\!] = (\cdot \leftarrow (t))
\end{aligned}
$$

Fig. 4. Redex Binding Expansion

Figure 5 presents an extension to the core calculus adding a simple type system. A type $\to S\ T$ describes the change in *shape* from the given inputs to the resulting outputs of executing a term. A shape is a sequence of types, and describes the type of every value that will be available to the right of a term on execution.

$$
\begin{aligned}
S, T, U &::= (A\ ...) \\
A, B, C &::= (\Rightarrow S\ T)
\end{aligned}
$$

Fig. 5. Redex Type Extension

A Kihi program is typed by the shape judgement, defined in Figure 6. The empty program does not change shape, and a non-empty program composes the changes in shape applied by their terms. Kihi terms are typed by the type judgement, defined in Figure 7. For instance, the type of $\times$ begins with a shape $(A\ B\ldots)$ and produces a shape $(A\ A\ B\ldots)$, representing the duplication of a value of type A.

The type system does not include a mechanism for polymorphism, and there is no way to abstract over stacks. As a result, every type must include the type of every value to its right, even if it is not relevant to that operation's semantics, so it is difficult to write a type that describes a broad range of possible programs.

The complete Redex implementation is available from https://github.com/zmthy/kihi-redex.

4.2 Racket

Kihi has also been implemented as a practical language in Racket. This version provides access to existing Racket libraries and supports some higher-level constructs directly for efficiency, but otherwise is modelled by the Redex. The Racket implementation is available from https://github.com/zmthy/kihi and operates as a standard Racket language with `#lang kihi`. The distribution includes some example programs, documentation, and a number of predefined utility functions.

4.3 Web

For ease of demonstration, a web-based deriving evaluator is available. This tool accepts a program as input and highlights each reduction step in its evaluation. At each step, the operation and operands next to be executed are marked in blue, the output of the previous reduction is underlined, and the rule that has been applied is noted. The program can be evaluated using both left- and right-biased choice of term to illustrate the different reduction paths, and Church numerals and booleans can be sugared or not. It supports many predefined named terms which alias longer subprograms for convenience.

The web evaluator can be accessed at http://ecs.vuw.ac.nz/~mwh/kihi-eval/ from any web browser. It includes several sample programs illustrating the tool and language, with stepping back and forth, replay, and reduction highlighting.

As a debugging aid, the evaluator includes two special kinds of term as extensions: for any letter X, `^X` is an irreducible labelled marker value, while `` `X `` reduces to nothing and has a side effect. These can be used to observe the propagation of values through the program and the order terms are evaluated.

The web evaluator also allows expanding a Kihi program to core terms (that is, using only the six operations of abstraction, application, left, right, copy, and drop). This expansion performs the full reduction of the `bind` syntax to core, and desugars all predefined named terms. In the other direction, a program can be reduced to the minimal equivalent program (including shrinking unapplied abstractions). Embedded is a command-line JavaScript implementation for node.js that also supports these features.

$$\frac{}{\mathsf{shape}[\![(), S, S]\!]}\ [\mathsf{Identity}]$$

$$\frac{\mathsf{type}[\![t_s, T, U]\!] \quad \mathsf{shape}[\![(t\ ...), S, T]\!]}{\mathsf{shape}[\![(t_s\ t\ ...), S, U]\!]}\ [\mathsf{Operate}]$$

Fig. 6. Redex Shape System

$$\frac{\text{shape}[\![v, S, T]\!]}{\text{type}[\![v, (A\ ...), ((\Rightarrow S\ T)\ A\ ...)]\!]}\ \text{[Abstraction]}$$

$$\frac{}{\text{type}[\![\cdot, ((\Rightarrow (A\ ...)\ (B\ ...))\ A\ ...), (B\ ...)]\!]}\ \text{[Apply]}$$

$$\frac{}{\text{type}[\![\leftarrow, ((\Rightarrow S\ (A\ ...))\ B\ C\ ...), ((\Rightarrow S\ (B\ A\ ...))\ C\ ...)]\!]}\ \text{[Left]}$$

$$\frac{}{\text{type}[\![\rightarrow, ((\Rightarrow (B\ A\ ...)\ T)\ B\ C\ ...), ((\Rightarrow (A\ ...)\ T)\ C\ ...)]\!]}\ \text{[Right]}$$

$$\frac{}{\text{type}[\![\times, (A\ B\ ...), (A\ A\ B\ ...)]\!]}\ \text{[Copy]}$$

$$\frac{}{\text{type}[\![\downarrow, (A\ B\ ...), (B\ ...)]\!]}\ \text{[Drop]}$$

Fig. 7. Redex Type System

5 Related Work

Kihi bears comparison to Krivine machines [9], Post tag system languages [11], and other term-rewriting models. We focus on the compositional nature of execution in Kihi rather than the perspective of these systems and will not address them further in this space.

As a simple Turing-complete language without variables, Kihi also has similar goals to the SK calculus [1]. The core calculus of Kihi has five operators, compared to SK's two, but functions in Kihi are afforded more flexibility in their input and output arities. The K combinator can be implemented in Kihi as $\downarrow$ `swap`, and the S combinator as $\cdot$ `under (under (·) swap under (×))`. While the reverse is possible, it requires implementing a stack in SK so we do not attempt it here.

Forth [10] is likely the most widely-known concatenative language. Forth programs receive arguments on an implicit stack and push their results to the same stack, following a postfix approach where calls follow their arguments. While generally described in this imperative fashion, a Forth program is also (impurely) functional and compositional when examined from the right perspective: each function takes a single argument (the entire stack to date) and produces a single output (a new stack to be used by the next function); from this point of view functions are composed from left to right, with the inner functions preceding the outer. The library design and nomenclature of the language favour the imperative view, however. The implicit nature of the stack requires the programmer to keep a mental picture of its state after each function mutates it in order to know

which arguments will be available to the next, while Kihi's approach allows the stepped semantics of our tools while retaining a valid program at each stage.

The Joy language [12] is similar to Forth and brought the "concatenative" terminology to the fore. Joy incorporates an extensive set of combinators [4] emphasising the more functional elements of the paradigm, but is still fundamentally presented as manipulating the invisible data stack.

5.1 Om

The Om language [2] is closest to Kihi in approach. Described as "prefix concatenative", in an Om program the operator precedes its arguments and the operator plus its arguments are replaced in the program by the result, as in Kihi. The language and implementation focus on embedability and Unicode support and are presented in terms of rewriting and partial programs, rather than composition. Despite some superficial similarity, Om and Kihi do not have similar execution or data models and operate very differently.

Om's brace-enclosed "operand" programs parallel Kihi's abstractions when used in certain ways. In particular, they can be `dequote`d to splice their bodies into the program, as in Kihi's `apply`, and Om's `quote` function would be Kihi ← `()`. They can also have the *contents* of other operands inserted at the start or end of them: to behave similarly to Kihi's ← and → operators requires double-quoting, because one layer of the operand quoting is always removed, so that `->[expression] {X} {{Y}}` is analogous to → `(X) (Y)`; to get the unwrapping effect of `->[expression]` in Kihi would be → ← `(·)`. Om has a family of "arrow functions" `->[...]`, `<-[...]`, `[...]->`, and `[...]<-` for manipulating programs interpreted in various ways, but in general these do not relate to Kihi's arrow operators. An operand "program" can be interpreted as a Unicode string, list, dictionary, or function, and Om has distinguished functions for treating the program as each of these interpretations and moving elements in and out, or deconstructing elements (for example, turning `{ABC}` into `{A}{BC}`), contrasting with the uniform lower-level treatment in Kihi.

Single-step abstract execution of an Om program results in another Om program with the same result up to side effects. The Om implementation does not provide single-stepping as an option, but a program lacking necessary arguments pauses to wait for them to be supplied after evaluating as far as possible.

6 Future work

The separation of the six operations in Kihi allows exploration of the subset of programs that omit one or more of the operations. Copy-free programs parallel linear logic, while drop-free programs have similarities with the λ_I calculus and SCBI calculus, and left-free programs do not reorder terms. These subsets and their equivalences or limitations are worth further investigation.

While Kihi core is Turing-complete, it is impractical for large programs. Building on the core to create a more usable compositional language, building out

a useful set of default functions, and extending Kihi with more convenient data structures, is ongoing work. We are currently extending our past work on module systems and code reuse [6,8,7] to this end, and also on visual programming [5] for novices or end-users. We are interested in exploring domains and tasks where this computational style is beneficial, and integrating it into other systems.

Efficient implementation and representation of Kihi is another live issue. Construction of a suitable virtual machine or compiler for Kihi raises questions of executing the computational model and encoding the operations.

7 Conclusion

Kihi is a compositional functional language with practical higher-level functionality but only six core operations with simple semantics. A key aspect of Kihi's flexibility is the arity-neutral fashion in which functions can compose. We have presented Kihi and the tools we have built to execute and explore the language and the compositional model. These tools are capable of interacting with a broader ecosystem as well as illustrating execution paths and allowing a programmer to explore different facets of computation than most conventional languages and tools provide.

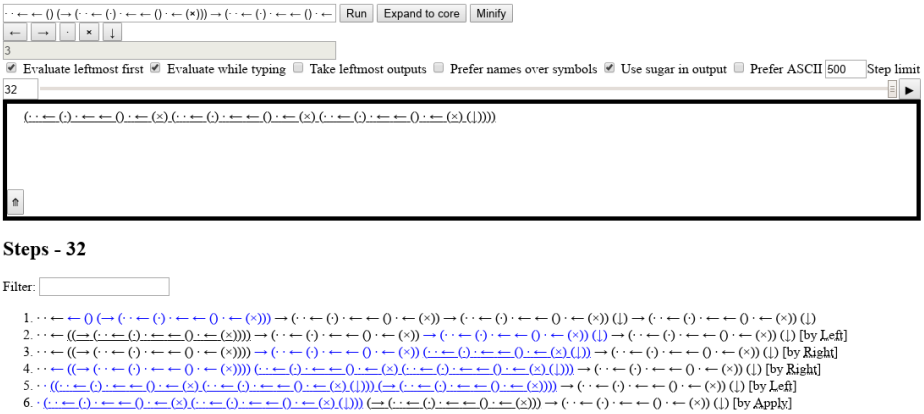
References

1. Curry, H.B.: Grundlagen der kombinatorischen logik. American Journal of Mathematics **52**(3), 509–536 (1930)
2. Erb, J.: Om programming language web site. https://sparist.github.io/Om/
3. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press (2009)
4. Frenger, P.: The JOY of Forth. SIGPLAN Not. **38**(8), 15–17 (Aug 2003). https://doi.org/10.1145/944579.944583
5. Homer, M., Noble, J.: A tile-based editor for a textual programming language. In: VISSOFT 2013. pp. 1–4 (Sept 2013). https://doi.org/10.1109/VISSOFT.2013.6650546
6. Homer, M., Bruce, K.B., Noble, J., Black, A.P.: Modules as gradually-typed objects. DYLA '13, ACM (2013). https://doi.org/10.1145/2489798.2489799
7. Jones, T., Homer, M., Noble, J.: Brand Objects for Nominal Typing. In: ECOOP 2015. LIPIcs, vol. 37, pp. 198–221. Dagstuhl, Germany (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.198
8. Jones, T., Homer, M., Noble, J., Bruce, K.: Object Inheritance Without Classes. In: ECOOP 2016. LIPIcs, vol. 56, pp. 13:1–13:26. Dagstuhl, Germany (2016). https://doi.org/10.4230/LIPIcs.ECOOP.2016.13
9. Krivine, J.L.: A call-by-name lambda-calculus machine. Higher Order Symbol. Comput. **20**(3), 199–207 (Sep 2007). https://doi.org/10.1007/s10990-007-9018-9
10. Moore, C.: 1x Forth (1999)
11. Post, E.L.: Formal reductions of the general combinatorial decision problem. American Journal of Mathematics **65**(2) (1943)
12. von Thun, M., Thomas, R.: Joy: Forth's functional cousin. In: Proceedings of the 17th EuroForth Conference (2001)

Screenshots and outline

This appendix provides tool screenshots, identifies various features, and notes points of behaviour that are incorporated in the demonstration.

Overall view of web evaluator



The top row includes, from left to right:

1. A text box for entering a Kihi program to evaluate;
2. A button that executes the program;
3. A button that translates the given program into core operations, expanding binds and named terms;
4. A button that shrinks excess terms inside abstractions.

The second row provides buttons for the operator symbols. The third is the output area, showing the final result value of the program and any values emitted by the evaluation.

The check boxes, from left to right:

1. Select the leftmost available reduction (checked) or the rightmost available reduction (unchecked);
2. Enable evaluation of the program without clicking "Run";
3. Remove values reaching the left-hand edge and move them to the output area;
4. Render operators as names (e.g. `right`) instead of symbols (→);
5. Sugar outputted Church numerals into textual numbers;
6. Render operators using substitute ASCII symbols (e.g. `:`) instead of mathematical symbols (×). This mode suits some limited browsers and systems.

The step limit determines the maximum number of reduction steps the evaluator will take before stopping. It also stops if a program becomes too long. These stopping points are to preserve browser resources. In particular, programs with nested binds (as in the provided factorial example program) can expand to

many thousands of terms of core Kihi, and creating the list of steps performs very poorly. This is a limitation of the web evaluator.

The fourth row allows manual stepping through the evaluation: jumping to a specific numbered step (left), dragging the slider through steps (middle), or automatically replaying and pausing evaluation (right).

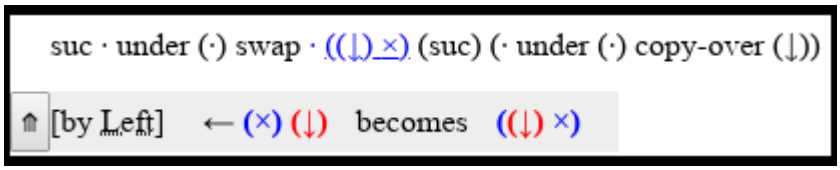
The black box shows the current program being evaluated at this step, depicted and described in more detail in the next section.

The "Steps" heading shows the total number of steps, and acts to hide or restore the complete list of reduction steps below. The filter text box permits showing only a subset of steps: for example, entering "left" will make only "left" reduction steps appear.

The list of steps shows the program as it is at each step, underlining any new terms introduced at that step and marking the rule used to obtain them. Hovering the mouse over the rule will show a detailed display of the specific reduction. The terms to be reduced next are highlighted in blue; it is possible for portions of the program to be both new (underlined) and to-be-reduced (blue) at once. Clicking on a step jumps the display above to that step of the program.

A labelled list of sample programs is below, any of which can be loaded and evaluated by clicking the title.

Single-step display of web evaluator



The complete program at this step is displayed at the top, with the rule that produced it displayed below. The underlined text in the program is that on the right-hand side of the rule display, and blue text is the next to be expanded as before. The rule display highlights different elements of the rule (for example, arguments) and matches corresponding elements on each side with the same highlighting.