1-Cycle List Induction

Daniel Nikpayuk

September 5, 2020

Abstract

In this essay I give an informal walk-through in how to build a general operator which acts on lists, such that it can be specialized to better known list operators such as *map, fold, find.* Its only restriction is that it can cycle or parse through a given list at most once.

Such an operator is of theoretical importance as not only are there several 1-cycle algorithms (other than map, fold, find) which are defined as specific instances, but there are also many higher-cycle algorithms which can immediately be composed out of these 1-cycle specializations.

This operator is also of practical relevance as I have taken care to consider its optimizations, giving its implementation a modular design so as to make it realistic for use in many applications.

Philosophy

Given that the title of this essay is "1-Cycle List Induction" we should discuss this idea of *induction* here as it informs the major philosophy of design throughout.

For starters, I borrow this word from the Type Theory expressed in [1]. There, induction is also called *(dependent) elimination*, which if you're unfamiliar can be thought of as a rule that effectively tells us what kind of functions are even possible for a given *type* definition. In this sense then, another way of thinking about induction is to consider it as an operator which is used to define functions related to a specific type.

My claim then is that the induction operator which is to be presented here can in its potential be used to create *all* possible functions that iterate *once* over a list.[1]

Methodology

The methodology is this essay can be split into two parts:

1. Methods to model list induction.

2. Methods to model lists as a type.

[1]Such an operator can be generalized to sublists of a given list, which is why at times I am comfortable stating it is an operator that is of *at most* one cycle.

Function Induction

Our model for list induction comes from aspects of [2] as well as [3] which both provide modelling strategies for functions in general. In particular we are interested in what is called *grammatical path* modelling from [2]. From [3] we will want to make use of *register induction* and in particular its grammatical notations.

As I do not assume reader familiarity with these sources, I will offer a brief as well as *naive* introduction to them here.

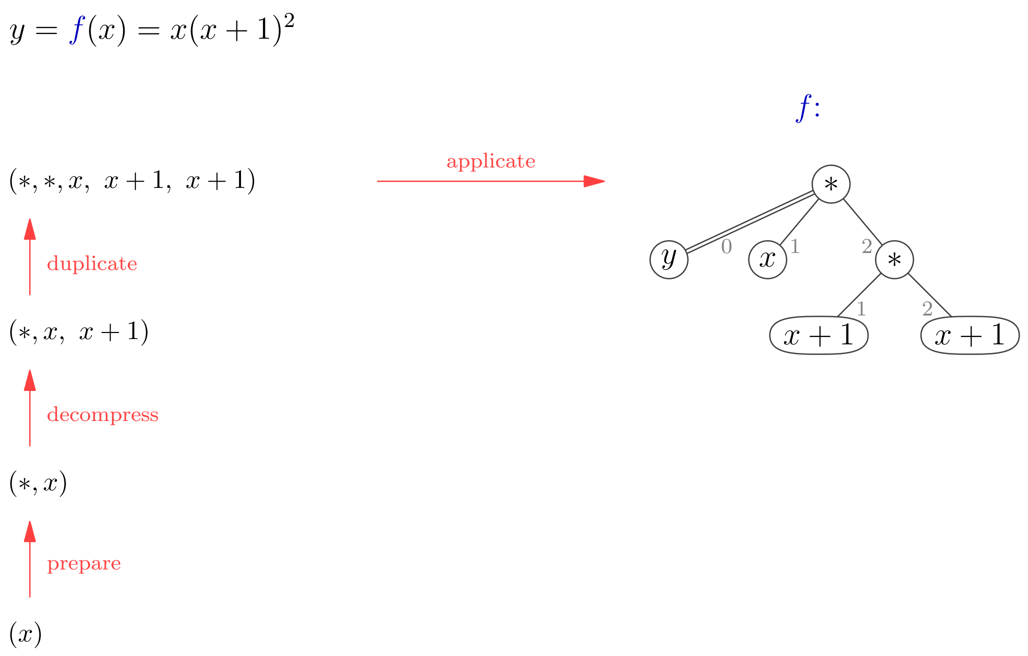
Grammatical Path Modelling

The basic idea behind grammatical path modelling is to view a function as having three components: A signature, a tree structure, and an evaluator (meta-operator) which first maps the content of the signature to the tree, and then applies accordingly to get the final return value.

Let's take the following function:

$$y = f(x) = x(x+1)^2$$

here expressed in regular *Eulerian* notation, as example. In grammatical path form this would be:



The idea is to start with the Eulerian function *specification* and make its evaluation process more explicit: In particular we begin with a bare minimum signature (x), expand it to include all required content, map that content to create to a proper evaluation ordering, and then evaluate.

The term "grammatical path" comes from the tree structure component of the model, where each branch is labelled by number. This effectively allows us to refer to any part by *pathname*, starting from the *root* operator. Such a convenience makes this notation useful for *algorithmic analysis*. Furthermore, as this notation is much more visually expansive than Eulerian notation it is also effective for top-down function design as it allows us to visualize the *whole* of the evaluation structure. Both of these reasons are why this modelling language was chosen to be used in this essay.

Register Induction Grammar

The register induction grammar of [3] is a grammar meant for building functions.

Getting right into an example of register induction, a *dual block* of such grammar appears as follows:

	composite x			**memory** y		
	open	**pass**	**call**	**closing**	**call**	**call**
0.	$policy?_{\mathbf{c},0}$	$next_{\mathbf{c},0}$	$f_{\mathbf{c},0}$	$policy?_{\mathbf{m},0}$	$break_{\mathbf{m},0}$	$f_{\mathbf{m},0}$
1.	$policy?_{\mathbf{c},1}$	$next_{\mathbf{c},1}$	$f_{\mathbf{c},1}$	$policy?_{\mathbf{m},1}$	$break_{\mathbf{m},1}$	$f_{\mathbf{m},1}$
2.	$policy?_{\mathbf{c},2}$	$next_{\mathbf{c},2}$	$f_{\mathbf{c},2}$	$policy?_{\mathbf{m},2}$	$break_{\mathbf{m},2}$	$f_{\mathbf{m},2}$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n.	$policy?_{\mathbf{c},n}$	$next_{\mathbf{c},n}$	$f_{\mathbf{c},n}$	$policy?_{\mathbf{m},n}$	$break_{\mathbf{m},n}$	$f_{\mathbf{m},n}$
	closed			**closed**		

There's a lot of information here, and there's potentially a lot going on with it, but the interpretation is rather straightforward.

First, it is split into two tables: *composite* and *memory.* The idea is each table has its own instruction set largely independent of the other, but from time to time these tables might still transfer content to each other. The composite table holds the function that is being built, while the memory table can be thought of as a stack. We read top down, starting with given input x for the composite table, and y for the memory table.

To explain how this grammar is evaluated, let's momentarily isolate the composite table as if it existed on its own: We would start the evaluation with the zeroth row, checking if its *policy* was true or false. If true, we would compose the input x with the leftside function (of the *pass* column) within the same row. In the above this function is named $next_{\mathbf{c},0}$. If false, we would instead compose the input x with the rightside function (of the *call* column) within the same row. This function in turn is named $f_{\mathbf{c},0}$.

This is the basic process, which is then conditionally repeated again and again until we evaluate the final row. In such a case we would potentially end up composing a chain of functions as our result, for example:

$$f_{\mathbf{c},n-1} \circ \ldots \circ f_{\mathbf{c},2} \circ f_{\mathbf{c},1} \circ x \circ next_{\mathbf{c},0} \circ next_{\mathbf{c},n}$$

Overall, that's the idea of this grammatical form. Also note, this chain may appear in an odd order, but is in fact valid—this will be explained shortly.

All that's left now is to explain the meaning of the column headers. In particular the policy columns get their names from mathematical interval terminology:

(a, b)	names an **open** interval.
$[\,a, b)$	names a **closing** interval.
$[\,a, b\,]$	names a **closed** interval
$(a, b\,]$	names an **opening** interval

The idea in terms of the above grammar is that each table corresponds with one of these intervals. In particular the *closing* header of the memory table is meant to suggest it corresponds with a leftside closed interval which in terms of its evaluation is to say it halts on its leftside. This closing table also corresponds with a rightside open interval and so its evaluation defaults to continuation on its rightside. In contrast to this, the composite table is *open* meaning it continues regardless of the policy value.

As for the **call** and **pass** headers, they tell us if we're composing left or composing right. To remember which is which think of it like this:

- If we were to *call* a function to be composed with (applied to) our subject of interest, the function would compose on the *left.*
- If we were to *pass* a function (as value) to be composed with our subject of interest, the function would compose on the *right.*

This is how we were able to achieve the unexpected ordering of functions in the above chain composition.

In anycase, the value of register induction and its grammar within this essay is that it offers us a uniform and Turing complete way to build functions. In practice we often don't even need the full dual block, as the single composite table is sufficient for a surprisingly large number of canonical computational functions.

Continuation Passing

Another way to express functions which we will need to model our 1-cycle inductor is to use what's called *continuation passing style*.

The intuitive inspiration of this concept is to take a function f, and redefine it to take an extra argument function c called the continuation, which is applied to the return value of the original f:

$$\begin{aligned} f &: \quad A \to B \\ \text{cont}(f) &: \quad A \to (B \to C) \to C \\ \text{cont}(f)(x, c) &:= \quad c \circ f(x) \end{aligned}$$

From here, we can generalize this concept using the theory of *monads*, to which we can then define a monadic composition operator ($\star$) here called an *endoposition* operator as follows:

$$f(x, c_1(y)) \star g(y, c_2(z)) \quad := \quad f(\, x,\ \lambda y.g(y, c_2(z))\,)$$

Keep in mind these generalized continuation passing functions need not have been derived from a standard function! As well, the value of this endoposition operator is that it behaves just the way we'd expect of composition operators, but is instead applied to non-standard functions. Thus we have parallel theories—different details, similar behaviours—which can even be used as alternatives to interpreting the previous register induction grammar (us having used only standard composition up until now).

Finally, in the case that we are working with continuation passing functions which were in fact derived from standard versions, our endoposition operator can be optimized as follows:

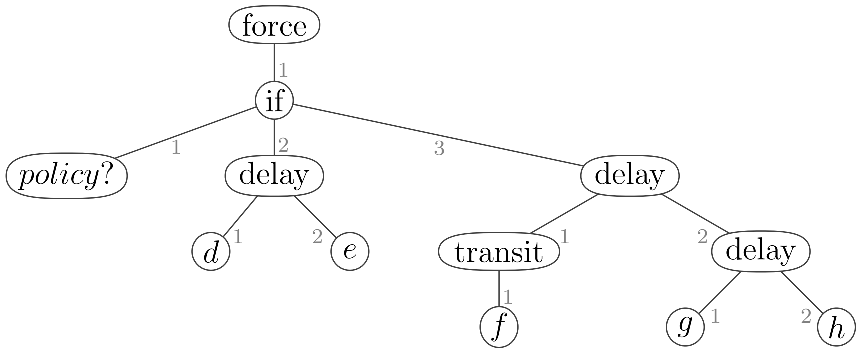
$$\text{cont}(f)(x, c_1(y)) \star \text{cont}(g)(y, c_2(z)) \quad = \quad \text{cont}(g \circ f)(\, x,\ c_2(z)\,)$$

Stem Operator

Register induction grammars have been previously implemented in [3] using what's called a *stem operator* and its corresponding *conditional endoposition operators*. Although these operators are indirectly relevant to us for this reason, they are also directly relevant as they are our means to modularizing the 1-cycle list operator.

For this essay then, I will summarize their definitions here, starting with stem:

$$\text{stem}(policy?, d, e, f, g, h):$$



I would think the general form here is fairly intuitive, with its notable component being the conditional operator **if**. Regardless, the interpretation is to compose operators d, e when the *policy*? value is true, and otherwise compose f, g, h when it is false. The **force**, **delay**, **transit** operators are there to prevent unnecessary calculations since we are working within a conditional: As we know we are branching, it would be wastefull to compute both $d \circ e$ and $f \circ g \circ h$. As for their operator definitions, they are as follows:

$$\begin{aligned} \text{delay}(d, e) &:= (d, e) \\ \text{force}((d, e)) &:= d \circ e \\ \text{transit}(f) &:= f \circ \text{force} \end{aligned}$$

Now, in terms of stem's conditional endoposition operators, they are defined in continuation passing style:

$$\begin{array}{llcll}
\langle \mathit{policy}?,\ \mathit{break},\ \mathit{next} \rangle & \underset{\text{call call}}{\star\{\text{closing}\}} & \mathit{cont} & := & \text{stem}(\mathit{policy}?,\ \mathit{break},\ -_{arg},\ \lambda y.\mathit{cont}(y, -_c),\ \mathit{next},\ -_{arg}) \\
\langle \mathit{policy}?,\ \mathit{break},\ x \rangle & \underset{\text{call pass}}{\star\{\text{closing}\}} & \mathit{cont} & := & \text{stem}(\mathit{policy}?,\ \mathit{break},\ -_{arg},\ \lambda y.\mathit{cont}(y, -_c),\ -_{arg},\ x) \\
\langle \mathit{policy}?,\ \mathit{break},\ \mathit{next},\ x \rangle & \underset{\text{call pose}}{\star\{\text{closing}\}} & \mathit{cont} & := & \text{stem}(\mathit{policy}?,\ \mathit{break},\ -_{w},\ \lambda y.\mathit{cont}(y, -_c),\ \mathit{next},\ x) \\
\langle \mathit{policy}?,\ w,\ \mathit{next} \rangle & \underset{\text{pass call}}{\star\{\text{closing}\}} & \mathit{cont} & := & \text{stem}(\mathit{policy}?,\ -_{arg},\ w,\ \lambda y.\mathit{cont}(y, -_c),\ \mathit{next},\ -_{arg}) \\
\langle \mathit{policy}?,\ w,\ x \rangle & \underset{\text{pass pass}}{\star\{\text{closing}\}} & \mathit{cont} & := & \text{stem}(\mathit{policy}?,\ -_{arg},\ w,\ \lambda y.\mathit{cont}(y, -_c),\ -_{arg},\ x) \\
\langle \mathit{policy}?,\ w,\ \mathit{next},\ x \rangle & \underset{\text{pass pose}}{\star\{\text{closing}\}} & \mathit{cont} & := & \text{stem}(\mathit{policy}?,\ -_{break},\ w,\ \lambda y.\mathit{cont}(y, -_c),\ \mathit{next},\ x) \\
\langle \mathit{policy}?,\ \mathit{break},\ w,\ \mathit{next} \rangle & \underset{\text{pose call}}{\star\{\text{closing}\}} & \mathit{cont} & := & \text{stem}(\mathit{policy}?,\ \mathit{break},\ w,\ \lambda y.\mathit{cont}(y, -_c),\ \mathit{next},\ -_{x}) \\
\langle \mathit{policy}?,\ \mathit{break},\ w,\ x \rangle & \underset{\text{pose pass}}{\star\{\text{closing}\}} & \mathit{cont} & := & \text{stem}(\mathit{policy}?,\ \mathit{break},\ w,\ \lambda y.\mathit{cont}(y, -_c),\ -_{next},\ x) \\
\langle \mathit{policy}?,\ \mathit{break},\ w,\ \mathit{next},\ x \rangle & \underset{\text{pose pose}}{\star\{\text{closing}\}} & \mathit{cont} & := & \text{stem}(\mathit{policy}?,\ \mathit{break},\ w,\ \lambda y.\mathit{cont}(y, -_c),\ \mathit{next},\ x)
\end{array}$$

This is a lot to take in, but its complexity is mostly representative of small variations.

In anycase, to read this ruleset the endoposition operator (signified in dark gray) first tells us how to compose `stem` operators:

$$\text{stem}(\mathit{policy}?,\ \mathit{break},\ \mathit{arg}_1,\ \mathit{cont},\ \mathit{next},\ \mathit{arg}_2)$$

Here though, we short form these operators into continuation passing style by refactoring $\mathit{arg}_1, \mathit{arg}_2$ and then partially applying $\mathit{policy}?,\ \mathit{break},\ \mathit{next}$ to derive alternative representations:

$$\langle \mathit{policy}?,\ \mathit{break},\ \mathit{next} \rangle(-_{arg},\ -_c) \quad := \quad \text{stem}(\mathit{policy}?,\ \mathit{break},\ -_{arg},\ -_c,\ \mathit{next},\ -_{arg})$$

Note that $\langle \mathit{policy}?,\ \mathit{break},\ \mathit{next} \rangle$ represents the function name. It's also worth noting that this function is in what I would call *conditional continuation passing style*. In particular, if we adhere to the rule that we don't apply or simplify the `stem` operator directly, then all the monadic rules of continuation passing hold. In that case, these rules are in the same format as the generic endoposition operator in the continuation passing section:

$$f(x, c_1(y)) \ \star\ g(y, c_2(z)) \quad := \quad f(\,x,\ \lambda y.g(y, c_2(z))\,)$$

The only difference being that the evaluation has been optimized with knowledge of `stem`'s definition, and the fact that the *right operand* (right argument) is short formed to *cont* to reflect this.

As an aside, I can now say at this point that the meaning of the phrase *conditional endoposition* is in the sense that once we allow ourselves to evaluate the internal `stem` operators, we then have to accept that any chain of conditionally endoposed functions may *break* before all of such functions are called.

Next, let's discuss the `pose` keyword expressed within the endopose modifiers: It is a header like `call` and `pass` which were discussed previously in terms of register induction grammar. Think of it as a convenience operator, where instead of taking its input from the evaluation process it has its own specialized and fixed input.

As for the number of definitions within this ruleset, the thing to note is that this list of nine conditional endoposition operators is as large as it is because of all the header variations seen within register induction tables. In truth we could rely on just a handful of these operators without any loss in *potential*, but we'd also lose much of the existing expressivity and convenience.

Finally, the `closing` nomenclature in the above definitions hint at the way in which `stem` and its **cposes** are used to implement register induction. The general idea then is to pair up these cposes to match the respective composite and memory tables of the previously discussed register induction construct. With that said, the matter is not actually this straightforward: For starters, there are the other table keywords, namely `opening`, and `open`: These respectively are presented in [3] as the `costem`, and `distem` functions, along with their own `cpose` operators. Beyond that, evaluating dual blocks only as dual cpose chains leaves no room for the blocks to communicate. In this case, the full solution is actually to modify the above definitions to accommodate for these interactions.

Type Theoretic Lists

With our list induction methodology explained, we now need to address the basic methods of lists as types.

In Type Theory a dependent or polymorphic list is defined recursively as:

$$\text{List}_A \ := \ \text{Nil} \mid A \times \text{List}_A$$

where A is the dependent type variable, and Nil as a type only has one element:

$$\text{null} : \text{Nil}$$

In practice, one can thus consider a list of this form to be the recursive nesting of pairs:

$$(a, b, c, \ldots, z) = (a, (b, (c, (\ldots, (z, \ \text{null}\,) \ldots))))$$

List Operator Primitives

Let's go over the basic inventory of primitive list operators needed to implement more general ones.

To start, it is safe to assume we have the *pair* constructors known as **cons** and **push**:

$$\begin{aligned} \text{cons}_A \quad &: \quad A \times \text{List}_A \to \text{List}_A \\ \text{push}_A \quad &: \quad A \times \text{List}_A \to \text{List}_A \end{aligned}$$

where `cons` *prepends* an element of type A to a given list of type List_A, while `push` *appends* an element of type A to a given list of type List_A.

We can also assume we have pair projections (also known as eliminators, or selectors):

$$\begin{aligned} \text{car}_A \quad &: \quad A \times \text{List}_A \to A \\ \text{cdr}_A \quad &: \quad A \times \text{List}_A \to \text{List}_A \end{aligned}$$

Here **car** projects the first element of the pair, while **cdr** projects the second element. In coding literature the returned second element is frequently referred to as the *rest* of the list.

The **cons**, **car**, **cdr** terminology I borrow from the LISP programming language. It should be noted that this language is an untyped lambda calculus while the functions in this context are intended to be typed. With that said, when the context is clear (or otherwise free of potential ambiguity), we will omit the type subscript A in the above notations as well as others yet to be introduced.

The next primitive operator we need is the conditional **if**, which we've already introduced as part of the stem operator. As we left that mentioned version untyped, let's now present it in its more technical form:

$$\text{if}_C \quad : \quad \text{Boolean} \times C_{true} \times C_{false} \to C_?$$

Here I'm abusing Type Theory notation: The intended meaning in the above is that the function **if** is defined on a family of types C where C is indexed by boolean values.[2] I've written $C_?$ at the end to indicate the type index (and thus the type itself) of the output value is determined by the boolean input value, meaning it is not known until this function is evaluated and is in fact determined as part of the evaluation. In type theory proper, they call this a *dependent function type.*

The final core operator we'll need for general lists is a test for equality:

$$\text{eq?}_B \quad : \quad B \times B \to \text{Boolean}$$

[2]In Set Theory notation this would be $\{\, C_\alpha \,\}_{\alpha \in \text{Boolean}}$

To reiterate, our inventory of list operator primitives is as follows:[3]

$$\{\text{ if, eq?, cons, car, cdr, push }\}$$

As an example of how these can be used to construct general list operators, we implement a basic "length" function in LISP style grammar with C style comments:

```
(define (length′ count list)               //  define length′
  (if (eq? list null)                      //  if list == null
   count                                   //  return count
   (length′ (+ count 1) (cdr list))        //  else (recursively) call
  )                                        //   (length′ (count + 1) rest)
)
```

If this version of the list *length* function is unfamiliar to you, the more intuitive interpretation is then specialized as:

$$\text{length}(\textit{list}) \;:=\; \text{length}'(0, \textit{list})$$

Modelling The 1-Cycle Loop

With the methodology behind us, we are now ready to start modelling our 1-cycle list operator.

With that said, at this stage we will be focusing on the operator's specification rather than any of its implementations. As for our spec, the key intuition needed in helping us to realize its form is that any general 1-cycle operator must *recursively* iterate over its given list. Thus, our goal is to focus on determining the recursive *loop* of this operator.

As for this spec's exposition: The general paradigm I will take here is *design through incremental attempt*, where each try is then framed casually as a puzzle to be pieced together.

The Initial Iteration (zeroth try)

We start with a naive zeroth try, where we reason about a single iteration of the loop, and in particular the initial iteration.

To this end, we will want to begin with an *initial list.* As we are iterating over the list's elements we will also want to split this initial list into two components:

$$\textit{initial list} \quad = \quad \{\ \textit{current}\ ,\ \textit{rest}\ \}$$

Following this, for our list operator to be meaningful we will then want to act on the list contents in some way. Since we only have direct access at the moment to the *current* component—*rest* being a sublist—we will now want to apply some function to it:

$$\textit{act}(\textit{current})$$

Lastly, we will want to pass the results of these steps to the next iteration of the loop:

$$\{\ \textit{act}(\textit{current})\ ,\ \textit{rest}\ \}$$

[3]It should be mentioned in the following should I use well known operators such as addition or multiplication within any examples I take them for granted as being outside the scope of this essay, and I am not explicitly counting them here as part of the inventory.

Loop Functions (first try)

The problem with the naive try is the input signature (the initial list) doesn't match the output signature:

$$(\, act(current),\ rest \,)$$

This is an issue since we are trying to create a recursive loop, meaning these signatures need to match. Even still, the zeroth attempt does provide us with valuable clues as to how we can move forward.

The next attempt starts by synchronizing the two previous signatures. As the *initial list* and the *rest* are expected to be lists, we can take a cover term for both, here named as the *precedent*. In that case then, the only difference between the signatures is that the output signature has one extra object $\{act(current)\}$ than the input signature. With that in mind, I declare our new signature as follows:

$$(\, resultant,\ precedent \,)$$

I've chosen the term *resultant* to be generic. Although we do have refined information to work with in the form of $act(current)$, I find it's a better practice not to overspecialize too soon—allowing us to maintain higher entropy or flexibility in our design until we have actual reason to restrict our definitions.

Okay, so let's go over our single loop iteration again!

We will want to separate *precedent* into two parts the way we did with the initial list. The intuitive choice here would be to use the `car` and `cdr` functions, but as we're designing for higher entropy right now we'll instead split our list with the following weakly specified functions:

$$value(precedent) \quad , \quad next(precedent)$$

From here, and as with our initial attempt, we will want to act on the list component we have access to, that being $value(precedent)$. In this case we can keep the same weakly specified function name:

$$act(value(precedent))$$

Now, we effectively have the same information as before:

$$\{\ act(value(precedent))\ ,\ next(precedent)\ \}$$

We pass this to the next iteration of the loop, but this time we have one extra piece of (unused) information:

$$resultant$$

In this case $act(value(precedent))$ is meant to coincide with the *resultant* of the next iteration, but we still have the *resultant* of this iteration. The way to resolve these distinct objects is to introduce a new function to "unite" them:

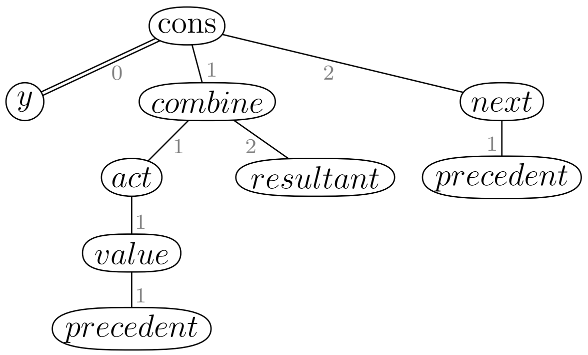
$$combine(\, act(value(precedent)),\ resultant \,)$$

Admittedly, the intuitive choice for *combine* would be `push` so as to build back up a new list to return, but we also don't want to restrict ourselves to this only. Certainly we will want this 1-cycle operator to be able to specialize as `map`, but we also want it to specialize as `fold`, in which case we should not be restricted to returning a list. Given this, let it be said that our *combine* operator could very well be a *monoid*, or something else still for that matter.

In anycase, we now have enough components to make the first real attempt at our loop model. I display it in grammatical path visual style as follows:

$$y = \text{loop}(\mathit{resultant}, \mathit{precedent}) :$$

$$\begin{tikzcd}
(\mathit{resultant}, \mathit{precedent}) \arrow[d, "\text{prepare}"] & \\
(\mathit{resultant}, \mathit{precedent}, \mathit{value}, \mathit{act}, \mathit{combine}, \mathit{next}, \text{cons}) \arrow[d, "\text{duplicate}"] & \\
(\mathit{precedent}, \mathit{resultant}, \mathit{precedent}, \mathit{value}, \mathit{act}, \mathit{combine}, \mathit{next}, \text{cons}) \arrow[r, bend right, "\text{applicate}"'] & \text{loop:}
\end{tikzcd}$$



To summarize, this *loop* iteration function:

$$\text{loop}(\,\mathit{resultant}, \mathit{precedent}\,)$$

takes us from one iteration state to the next, where we start with the *facade signature*

$$(\,\mathit{resultant}, \mathit{precedent}\,)$$

expand it, meta-map it to the grammatical tree form, then evaluate. In particular, we `prepare` the facade by adding to the signature the unique constant values that will be needed in the evaluation. Then we effectively `duplicate` this data to also be used in the later evaluation. Next, our `applicate` is the meta-map which bijectively positions the contents of the full signature into the tree to be evaluated. As for evaluation, it is simply the process already described leading up to this figure. Lastly, we repackage the output y using `cons` and return it along the 0-path, which is also signified by the parallel line segment.

To review grammatical path notation, the tree itself is numbered so we can reference its synactic locations, for example:

$$\text{loop}/1/1/1/ \quad = \quad \mathit{value}$$

which denotes the *value* function. The path starts from the *root operator*, here being `cons`. As another example, such paths represent both functions and objects:

$$\text{loop}/1/2/ \quad = \quad \mathit{resultant}$$

Here in particular it now denotes *resultant*.

Also, notice the difference between:

$$\text{loop}/1/1/1 \qquad \text{and} \qquad \text{loop}/1/1/1/$$

The former refers to the pathname expression itself, while the latter with the extra *slash* at the end (the far right side) refers to the object named in the model. In the language of semiotics the former would be the *signifier* while the latter would be the *signified*. With that said, as these small differences can be tedious to parse at the human level, I prefer to write:

$$\text{loop}/1/1/1 \qquad \text{as} \qquad \textit{value}$$

instead of

$$\text{loop}/1/1/1/ \qquad = \qquad \textit{value}$$

which is a convention I will maintain throughout.

Conditioning the Model (second try)

As is, our first try model covers much of what a 1-cycle list induction operator is expected to do as we can already specialize towards `map` and `fold`. The issue for us now is that our model still does not account for the operational component which would allow us to specialize toward the `find` operator. This then, is the inspiration for our latest attempt.

In particular, we would like to extend our existing model so that we could potentially *break* the loop each time a function is applied within a given iteration. More specifically, we would like to be able to test and possibly break *immediately before* as well as *immediately after* a given function application within the loop. This then would allow us to specialize toward our `find` operator.

With that being said, I would like to state that there are actually at least two additional reasons we might want the ability to break within a loop:[4]

- Loop breaking can act as an effective *short-circuiting* optimization. For example, if we are folding a list using multiplication we might at some point end up with the value *zero* as an intermediate result. In that case, assuming an appropriate context we might wish to reduce cycle costs by returning immediately rather than finishing the fold.

- If our list operator is untyped or even weakly typed, we might want to test the input and output before continuing the loop.[5]

By listing these here, we can now account for them within our coming design. Before continuing though, there is a practical concern we should also address.

With previous programming exprience I am willing to say that the looping grammars of modern programming languages such as `for`, `while`, or even *ranges* aren't designed to allow for *selective* branching overhead within a loop body. This is to say, we have to "hard-code" a break condition as part of the loop's definition, and although we can design for any given breaking test to be ignored (set the break condition to always be *false*), the overhead still exists. This is relevant to anyone who cares about performance as even overhead costs are proportional to the *length* of the input list within a loop.

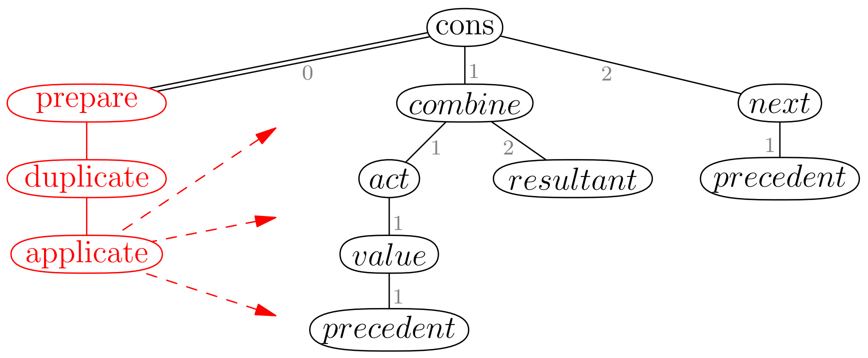
As an aside to this concern, I should also state that as far as I know modern compilers are often designed to optimize out overhead such as *unconditional* conditionals, but for the purpose of this essay I would rather not rely on external assumptions—I would rather design specifically to handle this ability ourselves.

Following all of these logics then, we are nearly ready to begin with only a bit of housekeeping still remaining. Notably, the previous loop modelling figure was quite large on the page, so we will want to condense it here as follows:

[4]One other reason not listed here is that as of yet we aren't actually able to break from the recursive loop in general. As it stands it's not a recursive loop, it's an infinite one.

[5]Ideally we would be working in a fully typed system, but in practice it's always possible to come across contexts in which such assumptions don't hold. For example part of my motivation for this essay comes from writing a C++ meta-programming library: C++ template programming is known to be Turing complete, but as it was never part of the intended design it does not provide us with full access the compiler's verification semantics nor debugging tools at that level.

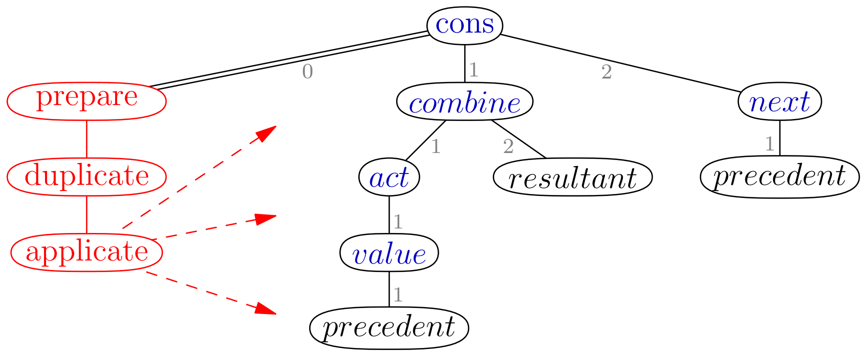
`loop`(*resultant*, *precedent*) :



In this figure, we still keep the signature expansion functions as in the previous, but since they are currently outside of our loop breaking concerns their details have been left hidden.

There is also a secondary motive for this housekeeping: Displaying the looping model this way helps us to analyze where and how to express our conditional tests. To that end, let's revise things a little further and highlight in blue (the former color of our `loop` name) the testing locations we're interested in:

`loop`(*resultant*, *precedent*) :



From here we can condense the relevant locator pathnames as follows:

`loop`	as	cons,
`loop`/1	as	*combine*,
`loop`/1/1	as	*act*,
`loop`/1/1/1	as	*value*,
`loop`/2	as	*next*

This is actually incomplete. As stated previously, we would like to test *before* and *after* each function and so our locator pathnames are more accurately described as:

`loop`/0	as	cons output,	`loop`/1/1/0	as	*act* output,
`loop`/1	as	cons left input,	`loop`/1/1/1	as	*act* input,
`loop`/2	as	cons right input,	`loop`/1/1/1/0	as	*value* output,
`loop`/1/0	as	*combine* output,	`loop`/1/1/1/1	as	*value* input,
`loop`/1/1	as	*combine* left input,	`loop`/2/0	as	*next* output,
`loop`/1/2	as	*combine* right input,	`loop`/2/1	as	*next* input

In such cases we take our original locator paths and by convention specify a function argument *pre-test* with the appropriate step location $/n$. We then specify a function image *post-test* using the zero step $/0$.

With this housekeeping out of the way, the objective for us now is to *modularize* loop breaking tests at the above stated locations, to which we should then be able link chosen modules together to otherwise reform our previous loop. Admittedly this may not be so straightforward on first thought, fortunately the solution was already given in the methodology: We will use the `stem` operator and its `cposes` to implement our loop.

Conditional Reduction (third try)

It's heartbreaking to say, but we're not yet done.

The current issue beyond our ability to break is less theoretical and more practical: We have twelve breakpoint locations in our baseline model, but can we do better? Can we reduce this? The short answer is yes, the long answer is that we need to find redundancy or general computations which aren't actually necessary with respect to these locations.

To start with this, the output of `cons` (with locator path /0) is intended to be a *container* meant to pass information to the next iteration of the loop (implemented as a signature pair). As such there's no point in testing this container, meaning we can safely ignore testing within `prepare`, `duplicate`, and `applicate`. As for the content of this container—being the *resultant* and *precedent* arguments (with respective locator paths $/1, /2$) of `cons`—it will be tested when we start the next loop iteration, meaning we can ignore testing them before they're passed to `cons`.

From here, we might be tempted to say the output of *combine* is redundant, because it becomes the new *resultant* in the next iteration, and it only occurs once in the looping diagram. This is an insightful logic, but if we're thinking in terms of future optimizations we may wish to break the loop there and then before we apply further computations. For example, let's say we left the testing until the next iteration and only then broke, we'd still have done the computations within the functions `prepare`, `duplicate`, `applicate`, and likely for nothing. I'm not saying a post-*combine* test will be done often, but we shouldn't deny its possibility altogether. We'll leave it in.

In anycase, the only other place we might be performing redundant calculations is the output of *next*, which is meant to become the new *precedent* in the loop iteration which follows. Yet, the same sort of logic we just used with the possibility of a post-*combine* test applies here as well, so we'll also leave this test in.

To summarize, we will from now on focus on the following reduced set of breakpoint locations:

$$\left\{\begin{array}{llllll}
\texttt{loop}/1/0 & \text{as} & \textit{combine}\text{ output,} & \texttt{loop}/1/1/1/0 & \text{as} & \textit{value}\text{ output,} \\
\texttt{loop}/1/1 & \text{as} & \textit{combine}\text{ left input,} & \texttt{loop}/1/1/1/1 & \text{as} & \textit{value}\text{ input,} \\
\texttt{loop}/1/2 & \text{as} & \textit{combine}\text{ right input,} & \texttt{loop}/2/0 & \text{as} & \textit{next}\text{ output,} \\
\texttt{loop}/1/1/0 & \text{as} & \textit{act}\text{ output,} & \texttt{loop}/2/1 & \text{as} & \textit{next}\text{ input} \\
\texttt{loop}/1/1/1 & \text{as} & \textit{act}\text{ input,} & & &
\end{array}\right\}$$

Conditional Ordering (fourth try)

Last issue: We may have our nine modular locations, but how do we order them?

When I say *order*, I don't just mean the locator pathnames in the above set, I mean what order do we evaluate our breakpoint tests? For composition chains such as: [6]

$$\left\{\begin{array}{r} \textit{value} \,.\, \textit{act} \,.\, \textit{combine} \,.\, \text{cons}, \\ \textit{next} \,.\, \text{cons} \end{array}\right\}$$

Testing necessarily proceeds in link order, that much is a given at least, but with separate chains which are otherwise independent of each other how do we sequence our evaluation order? [7]

This actually brings up the ideas of:

series evaluation vs **parallel evaluation**

[6]Here I am using the contravariant (opposite order) composition operator: $(f \,.\, g)(x) = g(f(x))$. I am also using this operator to express (in this limited scope) partial composition as there should be no confusion in general.

[7]It's worth adding here that one other use for grammatical path notation is it can help to clarify which aspects of a computation are independent, further aiding algorithmic analysis.

I don't want to get too much into this as it takes away from the purpose of the essay, but it would also be unfair to exclude the matter entirely. For the record, we will default to series evaluation, but the point worth bringing up as that the policy choice either way informs our model design from this point on.

As for parallel evaluation, it's worth stating that as long as we eventually receive all input arguments in terms of *combine* and `cons` respectively, I don't foresee[8] any issues regarding the order of these functions' evaluations. As for whether or not this operator is even worth splitting the workload in the first place, I suspect it is entirely dependent on the computational cost of the functions being used within the loop.

In terms of series evaluation, the assumption is we have a single processor and have to evaluate things one at a time, thus bringing us back to our order of evaluation issue. As to this, I would first mention that without further contextual information some aspects of the sequential ordering don't actually matter here. All the same, we still need to pick an order. To that end, rather than arbitrarily picking one, let's look to how optimization constraints might force a solution.

For me, the first example that comes mind is actually the breaking condition which checks if the *precedent* list is empty or not: It would be erroneous to call either `car` or `cdr` when the list was empty, and although we could run the same test before each of these functions were applied, in practice we tend to run this test only once so as to optimize out what is otherwise a redundant calculation. The problem with this is that the order of evaluation now matters: For example it's possible to place the test in the wrong location (between `car` and `cdr`) meaning we would at some point end up calling one of these two functions on an empty list.

In anycase, all of this brings up the point that there's no best order of evaluation in general, and that our intended implementation should design for this. Truth be told, setting up code to allow for such variations would complicate this essay beyond its intended *intuitive* explanation. As such, I instead choose here a default interpretation, known as the **applicative order** of evaluation.[9] Effectively it's the same left-to-right enumeration style seen when parsing over a tree diagram.

With the decision made, our loop model's fallback order of evaluation now becomes:

$$\left\{\begin{array}{lllllll}
\texttt{loop}/1/1/1/1 & \text{as} & \textit{value}\text{ input,} & \qquad & \texttt{loop}/1/2 & \text{as} & \textit{combine}\text{ right input,} \\
\texttt{loop}/1/1/1/0 & \text{as} & \textit{value}\text{ output,} & & \texttt{loop}/1/0 & \text{as} & \textit{combine}\text{ output,} \\
\texttt{loop}/1/1/1 & \text{as} & \textit{act}\text{ input,} & & \texttt{loop}/2/1 & \text{as} & \textit{next}\text{ input,} \\
\texttt{loop}/1/1/0 & \text{as} & \textit{act}\text{ output,} & & \texttt{loop}/2/0 & \text{as} & \textit{next}\text{ output} \\
\texttt{loop}/1/1 & \text{as} & \textit{combine}\text{ left input,} & & & &
\end{array}\right\}$$

This being read from top-to-bottom, left-to-right.

As an auxiliary note, and if it helps, this can also be (informally) written as:

$$\textit{value}\ .\ \textit{act} \quad\mapsto\quad \textit{combine}_{\text{left}} \quad\mapsto\quad \textit{combine}_{\text{right}} \quad\mapsto\quad \text{cons}_{\text{left}} \quad\mapsto\quad \textit{next} \quad\mapsto\quad \text{cons}_{\text{right}}$$

Finally, I want to make it clear before finishing this section that it is not my intention to lessen my design with this overspecialized order of evaluation. Rather, I would make the claim that no generality will be lost due to the modular nature of any implementation using the `stem` operators and their `cposes`. In these designs evaluative rearrangements of order may not be automated, but it is only a little extra work to extend such support.

Implementing 1-Cycle Induction

As our specification is now complete, we are ready to implement the general 1-cycle operator using register induction grammar from the methodology section. In this case, we don't actually need a dual block, and can rely simply on the composite table, or an equivalent thereof.

Our 1-cycle list induction operator is defined as follows:

[8]Knock on wood as they say.

[9]This order of evaluation is often used as default by interpreters and compilers when they're evaluating the source code functions given to them.

```
**induct** 1-cycle_operator 1-$cycle\_name$  $value$  $act$  $combine$  $next$

        test_$value$_input?          $policy_0$  $break_0$    test_$combine$_right_input?  $policy_5$  $break_5$
        test_$value$_output?         $policy_1$  $break_1$    test_$combine$_output?       $policy_6$  $break_6$
        test_$act$_input?            $policy_2$  $break_2$    test_$next$_input?           $policy_7$  $break_7$
        test_$act$_output?           $policy_3$  $break_3$    test_$next$_output?          $policy_8$  $break_8$
        test_$combine$_left_input?   $policy_4$  $break_4$

    **define** 1-$cycle\_name$  $resultant$  $precedent$

        **repeat** $\infty$ ( $resultant$, $precedent$ )

        alias ( $local\_resultant$, $local\_precedent$ )

        **open**[$\star$] $local\_precedent$      **pass**                                      **pass**

        test_$value$_input?            $\langle$ $policy_0$, $break_0$, $value$ $\rangle$              cont($value$)
        test_$value$_output?           $\langle$ $policy_1$, $break_1$, id $\rangle$                   id
        test_$act$_input?              $\langle$ $policy_2$, $break_2$, $act$ $\rangle$                cont($act$)
        test_$act$_output?             $\langle$ $policy_3$, $break_3$, id $\rangle$                   id
        test_$combine$_left_input?     $\langle$ $policy_4$, $break_4$, $combine$ $\rangle$            cont($combine$)

        **closed**

        assign $curried\_combine$

        **open**[$\star$] $local\_resultant$      **pass**                                      **pass**

        test_$combine$_right_input?    $\langle$ $policy_5$, $break_5$, $curried\_combine$ $\rangle$   cont($curried\_combine$)
        test_$combine$_output?         $\langle$ $policy_6$, $break_6$, id $\rangle$                   id

        **closed**

        assign $curried\_cons$  $cons$

        **open**[$\star$] $local\_precedent$      **pass**                                      **pass**

        test_$next$_input?             $\langle$ $policy_7$, $break_7$, $next$ $\rangle$               cont($next$)
        test_$next$_output?            $\langle$ $policy_8$, $break_8$, id $\rangle$                   id

        **closed**

        $curried\_cons$
        **halt**
```

I would hope that much of this induction is sufficiently clear and intuitive—or at least its meaning can be guessed at with some accuracy—but I shouldn't assume either.

The first thing to note is the **induct** keyword, which tells us we're defining a family of functions. The scope of induction operators ends with the keyword **halt**. The first parameter of this induction is `1-cycle_operator` which is the name of this particular operator. Its second parameter is 1-*cycle_name* which is the name of the specific function we're defining—this is given that we're working within a family of such *1-cycle operators*. Following that we have the parameters for the various *value*, *act*, *combine*, *next* functions of our model. After that we then have a whole lot of parameters which specify the breakpoints we might want to `test`. These parameters can be grouped as triples:

test? *policy* *break*

Here we would tell our inductor whether or not we want a breakpoint `test` at the given location, and if so we then specify which *policy* and *break* functions to partially apply to the *conditional continuation passing* function we're (potentially) building within the chain. In the case we don't want a break point, we switch to an alternative continuation passing function, which is made possible by using an open table, but we'll discuss that detail soon enough. Also, if we don't want a break point at all, the choices of *policy* and *break* then become moot, for which default values such as $false$ and id can be provided.

Next, we have the **define** keyword where we are declaring that our 1-*cycle_name* function is being defined. As it is a function it is expected to have its own arguments, which here are the *resultant* and *precedent* values of our model. As for the body of the function, it starts with a **repeat** keyword previously introduced in [3]. This keyword says that the code which follows is to be repeated infinitely many times until its halting conditions are met. In many cases, the ∞ modifier could be replaced with the more precise length(*precedent*) which would guarantee it to halt, but for greater generality it's better to leave it as is.[10]

The first grammatical construct in the function body following repeat is the alias keyword which lets us *pattern match* and locally name the input. It might seem redundant, but given this block of code will likely repeat itself, the values of any intermediate output may end up differing from the initial *resultant* and *precedent* input (which are assumed immutable).

With the setup out of the way, we finally arrive at the core behaviour of the model. We've established the order of evaluation for breakpoints, but when translated into register induction grammar it becomes split into three blocks of open tables. The initial value at the beginning of each block can be translated into a continuation passing constant function to start the composition process.[11] Each of these tables create an endoposition of functions, where the block decidedly ends with a curried function, to which we then assign a name for later reference. This much it should be pointed out is actually a convenience to make the instruction set more readable.

Beyond that, we also need to discuss the modified form of open:

$$\textbf{open}[\star]$$

What this keyword means is that instead of using the baseline composition operator $[\circ]$ to connect the respective column functions, we are now using the standard continuation passing endoposition. Monadic composition parallels its baseline, and even though we introduced register induction grammars as being implemented with stem functions and their cposes, they can also readily be extended to endoposition style register induction tables by defining "higher composition" stem extensions as well.

The final thing to consider in this implementation is the use of identity functions. For starters, they are not labelled, and it is assumed that their respective *domains* and *codomains* can be inferred. In particular, unless specified the identity functions in the above algorithm also tend to be in continuation passing style. Beyond this, there are performance considerations: When we build a chain of endoposed functions, are we saying that identity functions are in the chain? Will they be called and applied unnecessarily? That generally seems wasteful, and weakens the appeal of this style of coding.

Fortunately for us, this particular performance issue has been dealt with in [3]: Instead of using stem (or in this case distem) directly, we switch to an alternative within a known lattice, allowing the induction process to optimize out unnecessary identity compositions. Although the details won't be provided here, the lattice is as follows:

$$\begin{tikzcd}
 & & \text{distem}_{(3,3)} \arrow[dl] \arrow[dr] & & \\
 & \text{stem}_{(2,3)} \arrow[dl] \arrow[dr] & & \text{costem}_{(3,2)} \arrow[dl] \arrow[dr] & \\
\text{pend}_{(1,3)} \arrow[dr] & & \text{dihold}_{(2,2)} \arrow[dl] \arrow[dr] & & \text{copend}_{(3,1)} \arrow[dl] \\
 & \text{hold}_{(1,2)} \arrow[dr] & & \text{cohold}_{(2,1)} \arrow[dl] & \\
 & & \text{if}_{(1,1)} & &
\end{tikzcd}$$

[10]If you'd like the option to specialize you could always add this as its own variable in a modified induction definition.

[11]This is made possible using the *unit* component of the underlying continuation passing monad.

Specializations

Let's build some 1-cycle list functions. We start with `map`:

1-cycle_operator map

$value$ = car act $combine$ = push $next$ = cdr

test_*value*_input?	$= true$	$policy_0 =$ isNull? $local\_precedent$	$break_0 = local\_resultant$
test_*value*_output?	$= false$	$policy_1 = false$	$break_1 =$ id
test_*act*_input?	$= false$	$policy_2 = false$	$break_2 =$ id
test_*act*_output?	$= false$	$policy_3 = false$	$break_3 =$ id
test_*combine*_left_input?	$= false$	$policy_4 = false$	$break_4 =$ id
test_*combine*_right_input?	$= false$	$policy_5 = false$	$break_5 =$ id
test_*combine*_output?	$= false$	$policy_6 = false$	$break_6 =$ id
test_*next*_input?	$= false$	$policy_7 = false$	$break_7 =$ id
test_*next*_output?	$= false$	$policy_8 = false$	$break_8 =$ id

This is followed by `fold`:

1-cycle_operator fold

$value$ = car act = id $combine$ $next$ = cdr

test_*value*_input?	$= true$	$policy_0 =$ isNull? $local\_precedent$	$break_0 = local\_resultant$
test_*value*_output?	$= false$	$policy_1 = false$	$break_1 =$ id
test_*act*_input?	$= false$	$policy_2 = false$	$break_2 =$ id
test_*act*_output?	$= false$	$policy_3 = false$	$break_3 =$ id
test_*combine*_left_input?	$= false$	$policy_4 = false$	$break_4 =$ id
test_*combine*_right_input?	$= false$	$policy_5 = false$	$break_5 =$ id
test_*combine*_output?	$= false$	$policy_6 = false$	$break_6 =$ id
test_*next*_input?	$= false$	$policy_7 = false$	$break_7 =$ id
test_*next*_output?	$= false$	$policy_8 = false$	$break_8 =$ id

And finally, we have `find`:

1-cycle_operator find

$value$ = car act = id $combine$ = push $next$ = cdr

test_*value*_input?	$= true$	$policy_0 =$ isNull? $local\_precedent$	$break_0 = (local\_resultant, local\_precedent)$
test_*value*_output?	$= true$	$policy_1 =$ match? arg	$break_1 = (local\_resultant, local\_precedent)$
test_*act*_input?	$= false$	$policy_2 = false$	$break_2 =$ id
test_*act*_output?	$= false$	$policy_3 = false$	$break_3 =$ id
test_*combine*_left_input?	$= false$	$policy_4 = false$	$break_4 =$ id
test_*combine*_right_input?	$= false$	$policy_5 = false$	$break_5 =$ id
test_*combine*_output?	$= false$	$policy_6 = false$	$break_6 =$ id
test_*next*_input?	$= false$	$policy_7 = false$	$break_7 =$ id
test_*next*_output?	$= false$	$policy_8 = false$	$break_8 =$ id

I would hope by now such detailed figures are largely straightforward, and so I won't explain how these specializations are read, but there is also a potential issue we need to discuss before concluding.

In particular, if you're concerned about the scope of the variables $local\_resultant$, $local\_precedent$, and arg (and the fact that we're breaking the *abstraction* principle of design), you'd be right in your concern: These variable names are meant to be restricted to the body of the function definition, not as part of specializations declared outside the original. And yet, as this is still only an essay I am allowing these ill-formed expressions for the sake of conceptual clarity, and the fact that the broader issue can be expressed in terms of what's known as the *signature problem.*

The signature problem is actually straightforward to solve using register induction grammar. The explanation of the problem and its solution are given in [3]. Truth be told, I did not implement this solution in the above 1-cycle definition as again the goal of this essay is aimed more toward conceptual clarity, but once you're comfortable enough with how the algorithm works it's not difficult to extend it to safely designate bound variables which are otherwise outside of their respective scopes.

In fact the signature problem also applies to the situation where we might want to designate the *act* and *combine* function variables (of our looping model) differently: Notably, one might want to declare such variables as part of a 1-cycle function definition rather than as part of the inductor definition as we've currently done with our existing implementation. For example, there are contexts in implementing the `map` operator in which declaring *act* as a function argument makes more sense. At the same time, there are also contexts in implementing the `fold` operator in which declaring *combine* as a function argument then makes more sense. In anycase, the signature solution would allow us a modified 1-cycle inductor to do these things as well.[12]

With that said, register induction—the grammar that allows for the solution to the signature problem—is actually based off of a theory of computation coinciding with what I am calling *concept theory*, which is a language describing concepts and associated terms such as *specification; weak, refined, resolved specification.* In terms of practical programming, a compiler based off of concept theory would allow us to declare a variable as a weak specification, where we could then resolve its *spec* as being a compile time or run time variable accordingly, which is to say *when needed.*[13]

Conclusion

I normally have more to say in my essay conclusions, but for this one I've pretty much already said everything I wanted to say, with the exception of: Thank you, I hope this text was sufficiently clear, and that it was helpful to you. Take care.

Pijariiqpunga.

References

[1] Homotopy Type Theory: Univalent Foundations of Mathematics. The Univalent Foundations Program (2013).

[2] D. Nikpayuk. Toward the Semantic Reconstruction of Mathematical Functions (2020). https://github.com/Daniel-Nikpayuk/Mathematics/blob/main/Essays/Function%20Semantics/Version-Two/semantics.pdf

[3] D. Nikpayuk. Grammatical Elements of Function Induction (2020). https://github.com/Daniel-Nikpayuk/Mathematics/blob/main/Essays/Function%20Induction/Version-Two/induction.pdf

[12] Too long don't read: We define an induction operator for the 1-cycle induction operator (viewed as a function).

[13] Although such flexibility in a compiler is technically required to solve the signature problem, there are still workarounds in other genres of programming languages, to which the 1-cycle implementation offered in this essay is still a valuable design.