Proposal for Basing Bend2 on W-Types

Aaron Stump

July 5, 2025

1 Bend2's Philosophy

Bend2 is based on a novel proposal (as far as I know) for how to design a type theory, which is a form of pure functional programming language that can also be viewed as a logic. To be sound as a logic, the language should not allow one to prove False.

There are two current traditions for devising type theories, that should be mentioned for comparison with Bend2's approach:

1. **Church-style** type theory builds up a notion of typed terms (programs), where the types are inherent to those terms. By a difficult argument, one shows that all well-typed programs terminate. So the type system is enforcing termination, in addition to other properties usually enforced by static typing. From termination, it is then easy to argue that the system is logically sound. This is because it is relatively easy to show that values, which are the final results of computation, cannot have type False.
2. **Curry-style** type theory starts with a notion of type-free program, and then adds types to describe properties of the behavior of programs. For example, the identity function can be described as having type $X \to X$ for any type X, as it is guaranteed to take an input of type X and return an output of type X (namely, the input it was given). A difficult argument is still required to show that typing enforces termination. But the language design is made quite a bit easier by not having types be inherent parts of programs. This is because in reasoning about programs, one does not then have to reason about types inside them. Programs are type-free, and typing comes second. In fact, the slogan I propose for this style of type theory is "Computation First" (because we first explain what type-free programs are and how they execute, and only afterwards use types to describe their properties).

Bend2's philosophy can be viewed as a strengthened form of Curry-style type theory, with the modified slogan: "Terminating Computation First". The idea, proposed by Victor Taelin, is similar to Curry-style type theory, where one first defines type-free programs, and how they compute. But differently, these programs are designed so that they are guaranteed to terminate, without reference to any notion of typing. Just the structure of the programs and the rules for how the execute are sufficient to establish that all programs terminate. Giving a detailed proof of that fact is still not trivial, but expected to be much simpler than the approaches based on typing. And then one has a lot of freedom to design a type system on top of the terminating type-free language. Now the only requirement is that the language should have the usual type-safety property that ones expects of any statically typed programming language. This is vastly easier to achieve than crafting a type system that enforces termination.

2 Bend2's notion of terminating type-free programs

As it exists currently, Bend2's terminating language is based on a notion of sizes of values, where values are built from tuples (including the empty tuple, which is just the unit value familiar from other functional programming) using labels. A list, for example, is just a nested tuple with labels through the nesting that

indicate whether the tuple represents an empty list (and hence has no more data), or adding an element to the head of another list (which is then contained inside this tuple). Terminating recursions operating on such data just need to be structurally recursive: all recursive calls should happen on subdata of the input. Anonymous functions are restricted to being *affine*, which means they may use their inputs either one time or not at all. These restrictions individually are well-known to enforce termination of type-free programs, and in combination they should continue to enforce termination (this may be known, but I am not aware of a proof).

This representation of data as nested tuples is appealingly simple. It does have the drawback that data need to have a finite size. In type theory, however, it is common to work with data structures that are infinitely branching, but where each branch is finite. Such structures do not have a finite size. One can extend finite sizes to the ordinals, and then cover such cases. In fact, ordinals are a classic (if theoretically motivated) example of such a datatype:

```
data Ord = Zero | Succ | Lim (Nat -> Ord)
```

The `Lim` constructor has an infinite number of subtrees, one for each natural number. This infinite set of subtrees is represented by having `Lim` accept a function (call it `f`) from `Nat` to `Ord`. Given natural number `i`, the function `f` returns the `i`'th subtree.

Basing a test for structural decrease of programs using ordinals might be possible, but I do not know of an example of this. It would require defining some suitable class of ordinals, and then testing functions for decrease with respect to that class. I am not sure how one would be able to do that in a type-free way, as I am not sure how one would check, just by seeing how code is manipulating part of a data structure, that the ordinal measure of the entire data structure is being decreased.

3 W-types for general structurally terminating recursion

An alternative – which might possibly be viewed as the correct way of achieving some kind of implicit form of ordinal decrease as just described – is to use so-called W-types, proposed by the great type theorist Per Martin-Löf (see [2, Chapter 15] for an introduction). Of course, with Bend2's strengthened Curry-style philosophy, we begin not with the types of this proposal, but just the type-free programs. There are two constructs, one for constructing values, and the other for recursing over them.

3.1 Construction and recursion

A construction consists of a *label* l and then a function f producing the subdata. Each input to f is like a name or index for a piece of subdata, and given that name, f returns the corresponding subdata. So for the example above of `Lim` for ordinals, the name would be a natural number i, and the subdata would be the i'th ordinal contained in that limit ordinal. Let us write $\langle l, f \rangle$ for this construction with label l and function f. So it is effectively a pair, but the second argument is always a function.

To recurse over such a value, we write a function R which takes the label l, the function f, and also a function r that returns recursive results. R then returns a result, generally by calling r. For each possible input to f, the function r returns the result of recursing on the subdata $f\ i$; that is, the i'th piece of subdata. The recursor itself takes in R and a piece d of data, decomposes that data into l and f, and then invokes r. In more detail, let us write $R \bullet d$ for this. Then the computation rule is:

$$R \bullet \langle l, f \rangle \quad = \quad R\ l\ f\ (\lambda x\,.\,R \bullet (f\ x))$$

So someone using the recursor writes R, and then for each piece of data $\langle l, f \rangle$, that function R will be invoked with the label l, and the function f, which R can then call as needed to obtain subdata. The third argument to R is the value that R will use for r, namely $\lambda x\,.\,R \bullet (f\ x)$. That function takes in an index x, and

recursively invokes the recursor on the x'th piece of subdata, given by $f\ x$. We can see this in action through an example.

4 Examples

Here are a couple examples of standard datatypes, represented using this approach. The examples presuppose a set of labels of the form "$\mathsf{@name}$", together with a basic (termination-preserving) function $\{\mathsf{@name}_1 \mapsto t_1\ ;\ \cdots \mathsf{@name}_k \mapsto t_k\}$, which maps input label $\mathsf{@name}_i$ to t_i, and is undefined if applied to any label not listed. Later, the type system can make sure such functions are always called with a label from the list. Also, to serve as a base case for constructions, assuming we have a unit value $\langle\rangle$.

4.1 Natural numbers

We can encode zero and successor as follows:

$$\begin{array}{rcl} \mathit{Zero} & := & \langle \mathsf{@zero}, \lambda\, x\,.\, \langle\rangle \rangle \\ \mathit{Succ} & := & \lambda\, n\,.\, \langle \mathsf{@succ}, \lambda\, x\,.\, n \rangle \end{array}$$

So for example, the term $\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})$ normalizes to

$$\langle \mathsf{@succ}, \lambda\, x\,.\, \langle \mathsf{@succ}, \lambda\, x\,.\, \langle \mathsf{@zero}, \lambda\, x\,.\, \langle\rangle \rangle \rangle \rangle$$

This is not very different from an encoding using just labels and tuples, which would be

$$\langle \mathsf{@succ}, \langle \mathsf{@succ}, \langle \mathsf{@zero}, \langle\rangle \rangle \rangle \rangle$$

The encoding with W-types has additional lambda abstractions throughout the value, which the encoding with labels and tuples lack.

Now to recurse on natural numbers, we could either directly use the recursor for W-types defined above, or we can derive a recursor R_{Nat} specifically for natural numbers using the recursor for W-types:

$$\mathsf{R}_{Nat} := \lambda\, s\,.\, \lambda\, z\,.\, \lambda\, n\,.\, \{\mathsf{@zero} \mapsto \lambda\, p\,.\, \lambda\, r\,.\, z\ ;\ \mathsf{@succ} \mapsto \lambda\, p\,.\, \lambda\, r\,.\, s\ (p\ \langle\rangle)\ (r\ \langle\rangle)\}\ \bullet\ n$$

Given a function s to apply when the input n is a successor, and a value z to return if n is zero, this term applies the W-type recursor on the value n, with a function that takes action based on the label. In both cases, the function takes inputs p and r, where p is a function returning the predecessor number if the number is non-zero, and r is a function returning the result of recursion on that value (again, if the number is non-zero).

- If the label is $\mathsf{@zero}$, then the subdata function and recursion function are both unused, because *Zero* has no subdata. The function just returns z in this case.
- If the label is $\mathsf{@succ}$, then the function calls s on the predecessor and the result of recursion for the predecessor. Those values are obtained by calling p and r, respectively, with $\langle\rangle$ as input. Actually, in the type-free setting, it does not matter what input is used, as p and r are both constant functions, which discard their inputs.

Using R_{Nat}, one may then define the usual arithmetic functions in a standard way. For example, addition may be defined as:

$$\mathit{add}\ :=\ \lambda\, x\,.\, \lambda\, y\,.\, \mathsf{R}_{Nat}\ (\lambda\, p\,.\, \mathit{Succ})\ y\ x$$

This definition uses R_{Nat} to iterate the successor function – well, actually a function which first discards the predecessor, which R_{Nat} always supplies when invoking the step case of the recursion – starting with y. And since we are writing recursors here instead of iterators, we can write a constant-time predecessor function:

$$\mathit{pred}\ :=\ \mathsf{R}_{Nat}\ (\lambda\, p\,.\, \lambda\, r\,.\, p)\ \mathit{Zero}$$

The predecessor p is given to the step case of the recursion, which returns it.

4.2 Ordinals

The example of ordinals given above can be implemented using W-types. Here are definitions of the three constructors:

$$\begin{array}{lcl} \mathit{Zero} & := & \langle @\mathsf{zero}, \lambda\, x\,.\, \langle\rangle\rangle \\ \mathit{Succ} & := & \lambda\, n\,.\, \langle @\mathsf{succ}, \lambda\, x\,.\, n\rangle \\ \mathit{Lim} & := & \lambda\, f\,.\, \langle @\mathsf{lim}, f\rangle \end{array}$$

While we are still type-free here, intuitively, *Lim* is taking in a function from *Nat* to *Ord*, and installing that function as the second component of the construction labeled @lim. This means that we can obtain the i'th smaller ordinal from a limit ordinal just by calling that function f with i.

Using this definition, we can define ordinal addition:

$$\begin{array}{lcl} \mathit{add} & := & \lambda\, n\,.\, \lambda\, m\,. \\ & & \quad \{@\mathsf{zero} \mapsto \lambda\, p\,.\, \lambda\, r\,.\, n; \\ & & \quad\ @\mathsf{succ} \mapsto \lambda\, p\,.\, \lambda\, r\,.\, \mathit{Succ}\ (r\ \langle\rangle); \\ & & \quad\ @\mathsf{lim} \mapsto \lambda\, p\,.\, \lambda\, r\,.\, \mathit{Lim}\ r\}\ \bullet\ m \end{array}$$

This definition works, because in the @lim case, the function r will add the i'th sub-ordinal of the limit, to m. But this is exactly what we want to take the limit of, and so we just write *Lim r*. So W-types can support ordinal arithmetic. Proving theorems about such operations is involved, however, and there is quite a bit that can be studied about constructive theories of ordinals [1]. Here, the point is just to demonstrate terminating recursion with a generalized inductive datatype (one where values might not have finite sizes).

4.3 Pairs

There are several ways one could implement construction of pairs (x, y). In the context of affine typing, it might be best just to lambda-encode these:

$$(x, y) \quad := \quad \lambda\, c\,.\, c\ x\ y$$

This would allow an affine way to decompose the pair. One could also add pairs with pattern-matching *let* as primitive. Finally, one could define pairs with W-types this way:

$$\begin{array}{lcl} (x, y) & := & \langle\langle\rangle, \{@\mathsf{fst} \mapsto x;\ @\mathsf{snd} \mapsto y\}\rangle \\ \mathit{fst} & := & \lambda\, p\,.\, (\lambda\, l\,.\, \lambda\, f\,.\, f\ @\mathsf{fst})\ \bullet\ p \\ \mathit{snd} & := & \lambda\, p\,.\, (\lambda\, l\,.\, \lambda\, f\,.\, f\ @\mathsf{snd})\ \bullet\ p \end{array}$$

To extract the first value from a pair $\langle\langle\rangle, f\rangle$, you call f @fst, and similarly f @snd for the second value. But then you cannot get both components of the pair in an affine way.

For examples below, I will assume we have a function *open* with the following computation rule:

$$\mathit{open}\ (x, y)\ f = f\ x\ y$$

It is trivial to implement *open* using lambda-encoded pairs, as I am proposing.

4.4 Lists and non-recursive subdata

The datatype of lists is famously similar to *Nat*. Both have a non-recursive constructor, namely *Nil* for *List* and *Zero* for *Nat*; as well as a recursive constructor, namely *Cons* for *List* and *Succ* for *Nat*. The sole difference is that for lists, the recursive constructor takes extra data of some type A. This results in a type *List A* of lists storing values of type A.

The standard representation of lists with W-types is slightly annoying, because the non-recursive data is included in the label of the construction. To prepend head h to tail t, the code is:

$$\mathit{Cons} \;:=\; \lambda\, h\,.\,\lambda\, t\,.\,\langle(@\mathsf{cons}, h), \lambda\, x\,.\,t\rangle$$

So the head h of the list is stored in the first component of the W-type construction, along with the @cons label. For uniformity (so that given a list of unknown form, we can find the label telling whether it is a @cons or a @nil), one then needs the label to be a pair also in the nil case:

$$\mathit{Nil} \;:=\; \langle(@\mathsf{nil}, \langle\rangle), \lambda\, x\,.\,\langle\rangle\rangle$$

We just store some trivial data $\langle\rangle$ in the label, along with @nil. This is a little clunky, because then a function like append needs to dig in a level to the label part of the construction, to find out which case should be applied:

$$\mathit{append} \;:=\; \lambda\, x\,.\,\lambda\, y\,.\,(\lambda\, q\,.\,\lambda\, p\,.\,\lambda\, r\,.\,\mathit{open}\ q\ (\lambda\, l\,.\,\lambda\, d\,.\,\{@\mathsf{cons} \mapsto \mathit{Cons}\ d\ (r\ \langle\rangle); @\mathsf{nil} \mapsto y\}\ l))\ \bullet\ x$$

This function takes in two lists x and y, and then uses the W-type recursor to recurse on x. The first part (to the left of the $\bullet$ symbol) of the W-type recursor takes in the label q, subdata function p, and recursion function r, and then opens the label q to see what kind of list we have. When we open q, we will get a label l (either @cons or @nil), and a piece of non-recursive data d. In the @cons case, this d is the head of the input list, which we use to rebuild the list around the result $r\ \langle\rangle$ of recursing on the tail of the input list. In the @nil case, of course, we just return the second list y.

4.5 Factoring out nonrecursive data

It would be reasonable to change the format of W-types so that the construction explicitly has a place for nonrecursive data. So we would have

$$\langle l, n, f\rangle$$

where n is just the nonrecursive data associated with the given label, and f remains the function which returns subdata. The syntax of the W-type recursor would remain the same, but its computation rule would change as follows:

$$R\ \bullet\ \langle l, n, f\rangle \;\;=\;\; R\ l\ n\ f\ (\lambda\, x\,.\,R\ \bullet\ (f\ x))$$

So the function R which is being folded over the constructor value would be given the nonrecursive subdata n explicitly, as an extra argument. Everything else is the same as before. With this simple modification, lists could be written more naturally as follows:

$$\begin{array}{lcl}
\mathit{Cons} & := & \lambda\, h\,.\,\lambda\, t\,.\,\langle@\mathsf{cons}, h, \lambda\, x\,.\,t\rangle \\
\mathit{Nil} & := & \langle@\mathsf{nil}, \langle\rangle, \lambda\, x\,.\,\langle\rangle\rangle \\
\mathit{append} & := & \lambda\, x\,.\,\lambda\, y\,.\,(\lambda\, l\,.\,\lambda\, h\,.\,\lambda\, p\,.\,\lambda\, r\,.\,\{@\mathsf{cons} \mapsto \mathit{Cons}\ d\ (r\ \langle\rangle); @\mathsf{nil} \mapsto y\}\ l)\ \bullet\ x
\end{array}$$

Now *append* is more natural, because the function we are folding over the first list x takes in the label l (which is either @cons or @nil), and then the nonrecursive data h. If l is @cons, then h is the head of the list, which we use as before. The code no longer needs to open the first component of the W-type construction to see what kind of list this is.

5 Adding types

Now that we have considered the type-free definitions of construction and recursion for W-types, let us consider how to type them. We will use the version of W-types factoring out nonrecursive data, as described in the previous section. We do not need to be too specific about the other typing features available, but they should include at least function types and a kind $\star$ to be the classifier for types (so $\mathit{Bool} : \star$, for example).

Also, there should be enumeration types $\{\mathsf{l}_1, \cdots, \mathsf{l}_k\}$ describing set of labels. We should have a unit type $\langle\rangle$ for the $\langle\rangle$ value (so with typing $\langle\rangle : \langle\rangle$). Finally, I write $\bot$ for an empty type (with no inhabitant $t : \bot$).

First, we need to add a new primitive type form for W-types, to describe constructions $\langle l, n, r\rangle$, where (to recall) l is a label indicating which constructor is used, n is the nonrecursive data, and r is a function returning subdata. To type such a construction, we clearly need:

- a type L for the label l
- a type constructor N where $N\ l$ is the type for the nonrecursive data
- a type constructor R where $R\ l$ is the type for inputs to the function r

N and R are type constructors (that is, functions returning types), because we need to describe the types for each possible label l. So given a label l, we need to know what the type will be for nonrecursive data of a construction labeled l, for example. In the case of the recursive subdata, only the input type of r needs to be specified, because the output type is determined already: given an index (like the natural number i selecting the i'th smaller ordinal for a *Limit*), r returns a piece of subdata, which recursively has the W-type again. I propose this syntax:

$$\langle L, N, R\rangle$$

Following the above discussion, this has kind $\star$ (so it is a type) assuming

- L has kind $\star$
- N and R both have kind $L \to \star$

If one wanted to ensure that the top-level syntactic form of an expression determined whether that expression is a term, type, or kind, then it would be necessary to pick a slightly different syntax, maybe $\ll L, N, R \gg$ or (if one avoids Unicode) $\langle|L, N, R|\rangle$. In my opinion, it is rather nicer just to reuse the term syntax, if possible, the way Haskell does for tuples as values and tuple types (for example, $(\mathit{True}, \mathit{"hi"}) : (\mathit{Bool}, \mathit{String})$).

The typing rule for W-types as just described would then be:

$$\frac{L : \star \quad N : L \to \star \quad R : L \to \star}{\langle L, N, R\rangle : \star}$$

This records exactly the same information as in the informal description: the three premises just say what we wrote above, that L has kind $\star$ ($L : \star$), and so forth. Usually typing rules prove statements of the form $\Gamma \vdash t : T$, where Γ is a set of assumptions about the types or kinds of free variables. But for W-types we will not need to modify the typing context, so it can just be elided throughout the rules. This will make them easier to read.

5.1 Example: the type for lists

The type of lists storing data of type A would be defined as

$$\langle\{@\mathsf{cons}, @\mathsf{nil}\}, \{@\mathsf{nil} \mapsto \langle\rangle; @\mathsf{cons} \mapsto A\}, \{@\mathsf{nil} \mapsto \bot; @\mathsf{cons} \mapsto \langle\rangle\}\rangle$$

Here, the type L of labels is $\{@\mathsf{cons}, @\mathsf{nil}\}$. The type constructor N describing the nonrecursive data is

$$\{@\mathsf{nil} \mapsto \langle\rangle; @\mathsf{cons} \mapsto A\}$$

This is indeed a function from the type L to $\star$. N returns $\langle\rangle$ for $@\mathsf{nil}$, because $@\mathsf{nil}$ lists have no nonrecursive data; and N returns A for $@\mathsf{cons}$, for the type of the head of the list.

Finally, the type constructor R is

$$\{@\mathsf{nil} \mapsto \bot; @\mathsf{cons} \mapsto \langle\rangle\}$$

Recall that R is supposed to tell you the type for indices to the function r in a construction $\langle l, n, r\rangle$. Empty lists have no subdata, so one should not be able to call r at all if l is `@nil`. That is why the input type for r is $\bot$. From a typing perspective, calling r must return some subdata, of type *List A* (which is defined to be this W-type) again. Since a `@nil` list truly has no subdata, we must make sure it is impossible to call r, which we do by making r require an input that cannot possibly exist (unless in code which is already itself taking in an input of type $\bot$, but then that code cannot be called either). R returns $\langle\rangle$ if the label is `@cons`, because there is just one piece of subdata for a non-empty list, namely the tail. So given $\langle\rangle$ as an input, r will return the tail. Notice that it would not be correct to use a type with more inhabitants than $\langle\rangle$ as the value for R `@cons`, because then the *List A* type would also abstract values that held multiple pieces of subdata for `@cons` lists. But lists are supposed to be linear data structures, so there should just be one piece of subdata for `@cons` lists.

References

[1] Nicolai Kraus, Fredrik Nordvall Forsberg, and Chuangjie Xu. Type-theoretic approaches to ordinals. *Theor. Comput. Sci.*, 957:113843, 2023.

[2] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Lo¨f's type theory: an introduction*. Clarendon Press, USA, 1990.