Proposal for Basing Bend2 on W-Types

Aaron Stump

July 1, 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 [1, 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 "@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:

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

So for example, the term *Succ* (*Succ 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 @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 @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.

5 Adding types

References

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