Proposal: a Type Theory for Bend

Aaron Stump

July 10, 2025

1 What is a type theory?

A type theory is a statically typed programming language that can be understood as a logic. Programs are viewed as proofs, and the types of programs are viewed as the formulas they prove. This famous idea is called the Curry-Howard isomorphism. A very simple example is the program $\lambda\, x\, .\, \lambda\, y\, .\, x$, which takes in input x and then input y, and returns x. This program can be given the type $A \to B \to A$, for any types A and B. That type expresses that the program takes in an input of type A and then one of type B, and returns a result of type A. That indeed correctly describes the behavior of $\lambda\, x\, .\, \lambda\, y\, .\, x$. But it is also a valid logical formula, where the $\to$ operator is implication: A implies B implies A for the trivial reason that A implies A, and adding an extra assumption that B is true does not change that fact. To go beyond just propositional logic, type theories use more expressive types than just implications. We will see examples below.

To be interpreted as a logic, it is not enough to have a way to view the types of a programming language as formulas. We must ensure that it is not possible to prove false formulas. So the language must be logically sound. In type theory, an essential part of ensuring logical soundness is to guarantee that all programs terminate. The reason for this is that an infinite loop can be viewed, in most programming languages, as having any type one wants. So you can prove the formula False by writing a diverging program.

Much theoretical effort has been expended on techniques for proving logical soundness of type theories, by showing that all programs are guaranteed to terminate. Bend has a very interesting original approach to this problem, which we consider next.

2 Bend's approach to logical soundness

There are two current traditions for devising type theories, that should be mentioned for comparison with Bend'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

Variables	$x, y, z, \ldots$		
Machine integers	$i, j, \ldots$	$::=$	$0 \mid 1 \mid \cdots$
Arithmetic operators	o	$::=$	$+ \mid * \mid \cdots$
Terms	s, r, t	$::=$	$x \mid i \mid t\ o\ t' \mid \lambda\, x\,.\, s \mid t\ t' \mid \langle\rangle \mid \langle l, n, r\rangle \mid r\ \bullet\ t$

Figure 1: The syntax for Bend's programming language. In λ-abstractions, the variable x is allowed to occur at most once in the body s

> 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). The great computer scientist Jean-Louis Krivine puts it simply: "types can be thought of as properties of λ-terms" [1, page 43].

Bend'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.

As we begin our look at Bend, it is important to emphasize one further design goal for the language: minimality. We aim to have a core language with a small number of primitive operations, and similarly for the type system. This is both for ease of implementation, and reliability of the language design.

3 Bend's core language

The syntax of Bend's core programming language is shown in Figure 1. Programs, called terms, can be variables x, machine integers i, or infix applications $t\ o\ t'$ of arithmetic operators o to arguments t and t'. They can also be anonymous functions $\lambda\, x\,.\, s$, with the restriction that x may be used at most once in s. Some restriction is needed, or else it is very easy to write diverging λ-terms. Traditionally, type theories have restricted anonymous functions using types. With Bend's approach to logical consistency, we need a type-free way to enforce termination of λ-terms. One method is to restrict how often a variable may be used in an anonymous function. Bend requires λ-bound variables to be used at most once. Such λ-abstractions are called affine. It is well known that this restriction ensures termination. It does impose serious limits on programs written with anonymous functions, but we will see that the way recursion works expands the possibilities greatly (while preserving termination).

Returning to the syntax: we have applications $t\ t'$ of a term t being used as a function to term t' given as the argument to that function. We have a trivial piece of data $\langle\rangle$, which is useful as a placehold. We could use a machine integer i as a base case instead, but we will see that with typing, it is more convenient to have a separate trivial piece of data. That is $\langle\rangle$. We have a way to form structured data $\langle l, n, r\rangle$, and a term $r\ \bullet\ t$ for recursing over such data. These constructs constitue a version of what is known as W-types, and they will be presented in detail below.

3.1 Structured data

To implement data structures, every programming language needs some approach to creating and processing structured data. In Bend, the main form of structured data is the construction $\langle l, n, r\rangle$. Before we discuss this, though, it is very helpful to have a way of constructing pairs (t, t').

3.1.1 Pairs

There are several possibilities for how to include pairs in Bend. Here we propose to λ-encode them. So we take this definition:

$$(t, t') := \lambda\, c \,.\, c\ t\ t'$$

Syntactically, we propose to add pairs as primitive syntax, but the implementation of Bend will just treat them as defined. There are two benefits to defining pairs this way. First, we do not need to add another primitive construction for pairs to the semantics of the language: they are just defined using a λ-term. Second, we gain affine access to the components of a pair: we can make use of a pair just once and still obtain both its components. If instead we had primitive accessors like $p.1$ and $p.2$ for accessing the components of a pair p, we would not be able to write simple functions like the one that swaps the components of a pair, as an affine function. For such a function would be written as

$$\lambda\, p \,.\, (p.2, p.1)$$

where we can see that the λ-bound input variable is used twice. Instead, we define swapping of pair p as

$$p\ \lambda\, x \,.\, \lambda\, y \,.\, (y, x)$$

This looks a little peculiar, but recall that pairs are defined to be functions. So if p is $(1, 2)$, for example, then we will have this computational behavior:

$$(1,2)\ \lambda\, x \,.\, \lambda\, y \,.\, (y,x)\ =\ (\lambda\, c \,.\, c\ 1\ 2)\ \lambda\, x \,.\, \lambda\, y \,.\, (y,x)\ \rightsquigarrow\ (\lambda\, x \,.\, \lambda\, y \,.\, (y,x))\ 1\ 2 \rightsquigarrow^{*} (2,1)$$

The input variable c gets instantiated with $\lambda\, x \,.\, \lambda\, y \,.\, (y, x)$, giving that λ-term access to both components of the pair. These λ-abstractions are all linear: the input variables c, x, and y are used exactly once in the terms where they are λ-bound.

3.1.2 Structures

The construction $\langle l, n, f\rangle$ will be called a W-structure (or just structure, for short). It is the basic form in Bend for recursively structured data. The idea for structures comes from the type-theoretic construction known as W-types. Indeed, Bend's structures are just a version of those for W-types. Since Bend starts from a type-free programming language and then adds types, we start with W-structures, and define the W-types that describe them later. We also have a primitive feature $r\ \bullet\ t$ of the language, for recursively processing a structure.

A structure $\langle l, n, f\rangle$ consists of three parts:

- a *label* l, to describe what kind of structure this is
- nonrecursive subdata n, which will not be recursively processed by recursions $r\ \bullet\ t$
- a recursive subdata function f, which takes in an index x that specifies which piece of recursive subdata is desired, and returns it.

The index x given to the subdata function f is drawn from some finite set if the structure has only finitely many immediate subdata. For example, a list node has one piece of subdata, namely the tail, while a node of a binary tree has two pieces of subdata. So the indices would come from a one-element set and a two-element set, respectively. But the power of W-structures comes from the fact that i could also be from an infinite set,

$$\begin{array}{lcl} (\lambda\, x \,.\, s)\ t & \rightsquigarrow & [t/x]s \\ R\ \bullet\ \langle l, n, f\rangle & \rightsquigarrow & R\ l\ n\ f\ (\lambda\, x \,.\, R\ \bullet\ (f\ x)) \\ i\ o\ i' & \rightsquigarrow & j\ \text{ according to machine semantics for } o \end{array}$$

$$\frac{s \rightsquigarrow t}{[s/x]r \rightsquigarrow^* [t/x]r} \qquad \frac{r \rightsquigarrow^* s \quad s \rightsquigarrow^* t}{r \rightsquigarrow^* t}$$

Figure 2: Reduction semantics for Bend

in which case the structure can have infinitely many immediate subdata. There is still a restriction, though. While there may be infinitely many paths through a structure, each path is finite. This enables us to define $r\ \bullet\ t$ as a form of terminating recursion over structures, because we cannot recurse infinitely deeply into a structure.

For recursions $R \bullet d$ over a structure d, we write a function R which takes the structure's label, nonrecursive subdata, and subdata function. R also is provided 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 $R \bullet d$ uses this function R to recursively process a structure d. The reduction rule, which explains how recursions work computationally, is:

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

So someone using the recursor writes R, and then for each piece of data $\langle l, n, f\rangle$, that function R will be invoked with the label l, the nonrecursive subdata n, and the function f, which R can then call as needed to obtain subdata. The fourth 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 $f\ x$.

3.2 Reduction semantics

Figure 2 gives the complete reduction semantics for Bend. This semantics specifies the meaning of programs by saying how they compute. Calling it a "reduction" semantics indicates that we are not specifying a deterministic evaluation order for programs. There might be multiple choices of which part of a term to reduce next, and the rules does not specify which one should be chosen. So the reduction relation is nondeterministic. But we will show below (Section **??**) that all choices are guaranteed to lead to the same result. It could happen that some choices lead to that result more efficiently. But they all will, in principle, succeed.

Returning to Figure 2: there are reduction rules for reducing terms where an anonymous function $\lambda\, x \,.\, s$ is applied to an argument t. This is the well-known β-rule from lambda calculus. It uses the notation $[t/x]s$ to denote the result of substituting t for x in s. This should be done in a standard way to avoid capturing free variables of t as one passes under λ-abstractions in s. There is also the reduction rule for W-structures, mentioned above. The two rule inference rules at the bottom of the figure then define multi-step reduction $\rightsquigarrow^*$ in terms of single-step reduction $\rightsquigarrow$. The first rule says that if you can single-step reduce s to t, then you can do a multi-step reduction of a term containing s some finite number of times to one containing instead t. The term r mentioned in the rule might not contain x at all, in which case we get reflexivity of $\rightsquigarrow^*$, as the rule will say that $r \rightsquigarrow^* r$ in that situation.

The semantics as presented here does not specify the exact behavior of the machine operations on integers. Those can be specified in detail later as needed. It is noteworthy that the semantics is quite compact, in terms of numbers of rules needed to define it. This is in keeping with Bend's goal of minimality.

4 Examples

5 Metatheory of Bend's programming language

References

[1] Jean-Louis Krivine. *Lambda-calculus, types and models.* Ellis Horwood series in computers and their applications. Masson, 1993.