Proposal: a Type Theory for Bend

Aaron Stump

July 23, 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

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 labels l. Labels will be used to distinguish between different kinds of data, and they could be implemented just by machine integers as well. There are also infix applications $t\ o\ t'$ of arithmetic operators o to arguments t and t'.

Terms 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 constitute a version of what is known as W-types, and they will be presented in detail below. Finally, there is a label-matching function $\{l_1 \mapsto t_1\ ;\ \ldots\ ;\ l_k \mapsto t_k\}$, which will return term t_i if applied to label l_i.

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').

Variables	$x, y, z, \ldots$		
Labels	l		
Machine integers	$i, j, \ldots$	$::=$	$0 \mid 1 \mid \cdots$
Arithmetic operators	o	$::=$	$+ \mid * \mid \cdots$
Terms	s, r, t	$::=$	$x \mid i \mid l \mid t\ o\ t' \mid \lambda\, x\,.\, s \mid t\ t' \mid \langle\rangle \mid \langle l, n, r\rangle \mid r \bullet t$
			$\{l_1 \mapsto t_1\ ;\ \ldots\ ;\ l_k \mapsto t_k\}$

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

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 W-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 *selector* s, which indicates 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, 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. Note that for representing particular datatypes below, the selectors will always just be labels, but the definition does not require this.

For recursions $r \bullet t$ over a structure t, we write a function r which takes the structure's selector, nonrecursive subdata, and subdata function. r also is provided a function q that returns recursive results. Then r returns a result, generally by calling q. For each possible input to f, the function q returns the result of recursing on the subdata $f\ i$; that is, the i'th piece of subdata. The recursor $r \bullet t$ uses this function r to recursively process a structure t. 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 $\langle s, n, f \rangle$, that function r will be invoked with the selector s, 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 q, 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 6) 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 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. Finally, there is a reduction rule for applying a label-matching function $\{l_1 \mapsto t_1\ ;\ \ldots\ ;\ l_k \mapsto t_k\}$ to a label l_i, where $i \in \{1, \ldots, k\}$. The term reduces to t_k in that case. Applying a label-matching function to any other argument does not reduce. It is a stuck term, and we will impose typing (Section 4 below) to prevent such terms from arising.

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. It is noteworthy that the semantics overall is quite compact, in terms of numbers of rules needed to define it. This is in keeping with Bend's goal of minimality.

$$
\begin{array}{lcl}
(\lambda\, x\,.\, s)\ t & \rightsquigarrow & [t/x]s \\
r\ \bullet\ \langle s, n, f\rangle & \rightsquigarrow & R\ s\ n\ f\ (\lambda\, x\,.\, r\ \bullet\ (f\ x)) \\
i\ o\ i' & \rightsquigarrow & j \ \text{ according to machine semantics for } o \\
\{l_1 \mapsto t_1\ ;\ \ldots\ ;\ l_k \mapsto t_k\}\ l_i & \rightsquigarrow & t_i \ \text{ if } i \in [1,k]
\end{array}
$$

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

Figure 2: Reduction semantics for Bend

$$
\begin{array}{llll}
\mathit{TyVar} & \ni & X, Y, Z & \\
\mathit{TmVar} & \ni & x, y, z & \\
\\
\mathit{Knd} & \ni & \kappa & ::= \ \star \mid \Pi\, x : T\,.\, \kappa \mid \Pi\, x : \kappa\,.\, \kappa' \\
\mathit{Ty} & \ni & R, S, T & ::= \ X \mid \Pi\, x : T\,.\, T' \mid \Pi\, X : \kappa\,.\, T \mid \lambda\, x : T\,.\, T' \mid \lambda\, X : \kappa\,.\, T \mid T\ t \mid T\ T' \\
\\
\mathit{Tm} & \ni & r, s, t & ::= \ x \mid \lambda\, x : T\,.\, t \mid \lambda\, X : \kappa\,.\, t \mid t\ t' \mid t\ T
\end{array}
$$

Figure 3: Syntax of CC

4 Typing

Having proposed a syntax and reduction semantics for Bend, let us now consider typing. We can describe Bend's proposed type system very succinctly: it is the (Curry-style) Calculus of Constructions plus a simple variant of W-types, and including also a primitive typed extensional equality type. In this section, we will elaborate on this short description, one ingredient at a time.

One overarching point: the type system is Curry-style, meaning that typing rules should be viewed as defining a typing relation on the untyped terms whose syntax is given above in Figure 1. But in this section, we will present the rules using annotated terms, which contain enough information to confirm a typing for a term. An example is the annotated term $\lambda\, x : A\,.\, t$, with a type for the bound variable. So the typing rules look like Church-style rules, computing types for terms that have various type annotations in them. But there is a critical difference. With Curry-style typing, we are entitled to make use of an erasure function $|e|$ to drop all annotations from the term parts of expressions e. This function is used in the conversion rule, allowing us to change the type A of a term to a type B, if $|A|$ and $|B|$ are convertible. Using erasure at conversion allows us to equate many more terms than with Church-style typing, and eliminates many difficult technical problems reasoning about annotated terms, which are encountered in Church-style type theory.

4.1 The Calculus of Constructions

The syntax for the Curry-style Calculus of Constructions (CC) is listed in Figure 3. The three main syntactic categories defined there are annotated terms (Tm), types (Ty, and kinds (Knd). Types abstract terms, and kinds abstract types. For example, every natural number is a term, abstracted by the type Nat. And every type (like Nat) is abstracted by kind $\star$. We also use meta-variable e for any form of expression, whether term, type, or kind. The typing rules are given in Figure 4.

Erasure, used in the conversion rules CONV and T-CONV, is defined in Figure 5. Erasure leaves the structure of types and kinds unchanged, but drops typing annotations from terms. The conversion rules also reference a β-equality relation on kinds and types, respectively. This relation is defined in Figure 6, which also defines β-reduction for the two possible kinds of type-level β-redexes of CC (one where the argument is a term, the other where it is a type).

These typing rules for CC do not enforce that λ-abstractions are affine. This requirement is to be imposed on the erasures on terms, and it is checked syntactically, without reference to typing. This is so that the termination of the system does not depend on typing.

4.2 Enumerated types

We extend the syntax of CC defined in Figure 3 above, with the following additions:

$$\begin{array}{llll} \mathit{Labels} & \ni & l & \\ \mathit{Ty} & \ni & R, S, T & ::= \quad \ldots \mid \{l_1, \ldots, l_k\} \mid \{l_1 \mapsto T_1 \ ; \ \ldots \ ; \ l_k \mapsto T_k\} \\ \mathit{Tm} & \ni & r, s, t & ::= \quad \ldots \mid l \mid \{l_1 \mapsto t_1 \ ; \ \ldots \ ; \ l_k \mapsto t_k\} \end{array}$$

Then type-level reduction is extended:

$$\{l_1 \mapsto T_1 \ ; \ \ldots \ ; \ l_k \mapsto T_k\} \ l_i \ \rightsquigarrow \ T_i \ \text{ if } i \in [1, k]$$

Typing and kinding are extended in Figure 7. Label-matching functions at the term level are dependently typed: in the MATCH rule, the return type of the function is allowed to depend on the input label x. We do not have a similar dependency for label-matching functions at the type level, because CC does not have kind-level λ-abstractions.

4.3 W-types

To give types to the W-structures described in Section 3.1.2 above, Bend uses a form of W-types, introduced by Martin-Löf [2] (see also the exposition in [3]). Our syntax is further extended as so:

$$\mathit{Ty} \quad \ni \quad R, S, T \quad ::= \quad \cdots \mid \langle L, N, R \rangle$$

This new primitive type form is meant to abstract W-structures $\langle s, n, f \rangle$, where (to recall) s is a selector indicating which constructor is used, n is the nonrecursive data, and f is a function returning subdata. To type such a construction, we clearly need:

- a type S for the selector s
- a type constructor N where $N \ s$ is the type for the nonrecursive data
- a type constructor R where $R \ s$ is the type for inputs to the function f

N and R are type constructors (that is, functions returning types), because we need to describe the types for each possible selector s. So given s, we need to know what the type will be for nonrecursive data. In the case of the recursive subdata, only the input type of f 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*), f returns a piece of subdata, which recursively has the W-type again. The proposed syntax is $\langle S, N, R \rangle$.

Following the above discussion, $\langle S, N, R \rangle$ should have kind $\star$ (so it is a type) assuming

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

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 $\ll S, N, R \gg$ or (if one avoids Unicode) $\langle |S, N, R| \rangle$. 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:

$$\frac{\Gamma \vdash S : \star \quad \Gamma \vdash N : S \to \star \quad \Gamma \vdash R : S \to \star}{\Gamma \vdash \langle S, N, R \rangle : \star}$$

$$\text{TYPE}\ \frac{}{\Gamma \vdash \star} \qquad \text{TK-PI}\ \frac{\Gamma \vdash T : \star \qquad \Gamma, x : T \vdash \kappa}{\Gamma \vdash \Pi\, x : T\,.\,\kappa} \qquad \text{KK-PI}\ \frac{\Gamma \vdash \kappa \qquad \Gamma, X : \kappa \vdash \kappa'}{\Gamma \vdash \Pi\, X : \kappa\,.\,\kappa'}$$

(a) Classifiable kinds

$$\text{T-VAR}\ \frac{X : \kappa \in \Gamma}{\Gamma \vdash X : \kappa} \qquad \text{T-CONV}\ \frac{\Gamma \vdash T : \kappa \qquad |\kappa| =_\beta |\kappa'|}{\Gamma \vdash T : \kappa'}$$

$$\text{TT-PI}\ \frac{\Gamma \vdash T : \star \qquad \Gamma, x : T \vdash T' : \star}{\Gamma \vdash \Pi\, x : T\,.\,T' : \star} \qquad \text{KT-PI}\ \frac{\Gamma \vdash \kappa \qquad \Gamma, X : \kappa \vdash T : \star}{\Gamma \vdash \Pi\, X : \kappa\,.\,T : \star}$$

$$\text{TK-ABS}\ \frac{\Gamma \vdash \Pi\, x : T\,.\,\kappa \qquad \Gamma, x : T \vdash T' : \kappa}{\Gamma \vdash \lambda\, x : T\,.\,T' : \Pi\, x : T\,.\,\kappa} \qquad \text{KK-ABS}\ \frac{\Gamma \vdash \Pi\, X : \kappa\,.\,\kappa' \qquad \Gamma, X : \kappa \vdash T : \kappa'}{\Gamma \vdash \lambda\, X : \kappa\,.\,T : \Pi\, X : \kappa\,.\,\kappa'}$$

$$\text{TK-APP}\ \frac{\Gamma \vdash T : \Pi\, x : R\,.\,\kappa \qquad \Gamma \vdash t : R}{\Gamma \vdash T\ t : [t/x]\kappa} \qquad \text{KK-APP}\ \frac{\Gamma \vdash T : \Pi\, X : \kappa\,.\,\kappa' \qquad \Gamma \vdash T' : \kappa}{\Gamma \vdash T\ T' : [T'/X]\kappa'}$$

(b) Kinding

$$\text{VAR}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{CONV}\ \frac{\Gamma \vdash t : T \qquad |T| =_\beta |T'|}{\Gamma \vdash t : T'}$$

$$\text{TT-ABS}\ \frac{\Gamma \vdash \Pi\, x : T\,.\,T' : \star \qquad \Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda\, x : T\,.\,t : \Pi\, x : T\,.\,T'} \qquad \text{KT-ABS}\ \frac{\Gamma \vdash \Pi\, X : \kappa\,.\,T : \star \qquad \Gamma, X : \kappa \vdash t : T}{\Gamma \vdash \lambda\, X : \kappa\,.\,t : \Pi\, X : \kappa\,.\,T}$$

$$\text{APP}\ \frac{\Gamma \vdash t : \Pi\, x : T\,.\,T' \qquad \Gamma \vdash t' : T}{\Gamma \vdash t\ t' : [t'/x]T'} \qquad \text{T-APP}\ \frac{\Gamma \vdash t : \Pi\, x : \kappa\,.\,T' \qquad \Gamma \vdash T : \kappa}{\Gamma \vdash t\ T : [T/x]T'}$$

(c) Typing

Figure 4: Typing Derivations for CC

$$
\begin{array}{lcl}
|\star| & = & \star \\
|\Pi\, x : T\,.\,\kappa| & = & \Pi\, x : |T|\,.\,|\kappa| \\
|\Pi\, x : \kappa\,.\,\kappa'| & = & \Pi\, x : |\kappa|\,.\,|\kappa'| \\
|X| & = & X \\
|\Pi\, x : T\,.\,T'| & = & \Pi\, x : |T|\,.\,|T' \\
|\lambda\, x : T\,.\,T'| & = & \lambda\, x : |T|\,.\,|T'| \\
|T\ t| & = & |T|\ |t| \\
|T\ T'| & = & |T|\ |T'| \\
|t \simeq t'| & = & |t| \simeq |t'| \\
|x| & = & x \\
|\lambda\, x : T\,.\,t| & = & \lambda\, x\,.\,|t| \\
|\lambda\, X : \kappa\,.\,t| & = & |t| \\
|t\ t'| & = & |t|\ |t'| \\
|t\ T| & = & |t|
\end{array}
$$

Figure 5: Erasing annotations from terms, types, and kinds of CC

CONV-TTBETA

$$\frac{}{(\lambda\, x : T\,.\,T')\ t \rightsquigarrow^* [t/x]T'}$$

CONV-KTBETA

$$\frac{}{(\lambda\, X : \kappa\,.\,T')\ T \rightsquigarrow^* [T/X]T'}$$

CONV-RED

$$\frac{e \rightsquigarrow^* e'}{e =_\beta e'}$$

CONV-SYM

$$\frac{e =_\beta e'}{e' =_\beta e}$$

CONV-SUBST

$$\frac{e_1 =_\beta e_2}{[e_1/x]e =_\beta [e_2/x]e}$$

Figure 6: Reduction for type-level β-redexes, and conversion

ENUM

$$\frac{}{\Gamma \vdash \{l_1, \ldots, l_k\} : \star}$$

T-MATCH

$$\frac{\forall\, i \in [1,k]\,.\,\Gamma \vdash T_i : \kappa}{\{l_1 \mapsto T_1\ ;\ \ldots\ ;\ l_k \mapsto T_k\} : \{l_1, \ldots, l_k\} \to \kappa}$$

MATCH

$$\frac{\forall\, i \in [1,k]\,.\,\Gamma \vdash t_i : T\ l_i}{\{l_1 \mapsto t_1\ ;\ \ldots\ ;\ l_k \mapsto t_k\} : \Pi\, x : \{l_1, \ldots, l_k\}\,.\,T\ x}$$

Figure 7: Typing and kinding for enumerated types

This records exactly the same information as in the informal description: the three premises just say what we wrote above, that S has kind $\star$ ($S : \star$), and so forth.

4.3.1 Typing rule for W-structures

Now let us see the typing rule for constructions $\langle s, n, f \rangle$. For this to have a W-type of the form $\langle S, N, R \rangle$ we were just discussing, what has to be true?

- The selector s certainly has to have the type S.
- The nonrecursive data n should have the type $N\ s$ that the type constructor says should be there if the selector is s.
- The function f needs to take in a value of type $R\ s$, because this is the type for indices into the collection of subdata, and such an index is what f expects as input. It should return a result of the W-type, namely the subdata for the given index.

Formalizing these ideas, we get the following typing rule:

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash n : N\ s \quad \Gamma \vdash f : R\ s \to \langle S, N, R \rangle}{\Gamma \vdash \langle s, n, f \rangle : \langle S, N, R \rangle}$$

4.3.2 Typing rule for recursion

A recursion $r \bullet t$ takes in a function R and a value t of the W-type. Recalling the computation rule will help us understand what the typing of a recursion should be:

$$r \bullet \langle s, n, f \rangle \quad = \quad R\ s\ n\ f\ (\lambda x\,.\,r \bullet (f\ x))$$

We know from the typing rule for constructions that for some S, N, and R, we will have

$$\begin{array}{lcl} s & : & S \\ n & : & N\ s \\ f & : & R\ s \to \langle S, N, R \rangle \end{array}$$

Since R takes in those inputs on the right-hand side of the computation rule, we know that the type of R must look like:

$$\Pi\, s : S\,.\,\Pi\, n : N\ s\,.\,\Pi\, f : R\ s \to \langle S, N, R \rangle\,.\,\cdots$$

We just need to fill in the missing part of this type. We expect that the recursion is going to compute a value of some type constructor C which might be a function of the input value $\langle s, n, f \rangle$. So C will have kind

$$\langle S, N, R \rangle \to \star$$

This is similar to how a recursion on natural numbers n can compute a value of type $P\ n$, which is used, for example, when reasoning by induction on n. Now what type will the final (fourth) input of R on the right-hand side of the computation rule have? That input is

$$\lambda x\,.\,r \bullet (f\ x)$$

It is taking in a value of type $R\ s$, just as the subdata function f does. And it returns a value of type C (since it is recursing). C depends on the input to the recursion, so in this case, since $f\ x$ is the input, the result of the recursion will have type $C\ (f\ x)$. This means that the type for $\lambda x\,.\,r \bullet (f\ x)$ is

$$\Pi\, x : R\ s\,.\,C\ (f\ x)$$

Putting all this together, the type of R should be:

$$\begin{array}{l}\Pi\, s : S\,. \\ \Pi\, n : N\ s\,. \\ \Pi\, f : R\ s \to \langle S, N, R\rangle\,. \\ (\Pi\, x : R\ s\,.\, C\ (f\ x)) \to \\ C\ \langle s, n, f\rangle\end{array}$$

Finally, the type of $r \bullet d$ is $C\ d$. So if we think of $r \bullet d$ as a proof by induction, it is saying that to prove a property C of some value d of type $\langle S, N, R\rangle$, it is sufficent to assume

- $s : S$
- $n : N\ s$
- $f : R\ s \to \langle S, N, R\rangle$

and show $C\ \langle s, n, f\rangle$, assuming for induction hypothesis

$$\Pi\, x : R\ s\,.\, C\ (f\ x)$$

That type expresses that C holds for all the subdata that f can produce.

Putting this all into one rather indigestible rule, we have:

$$\frac{\begin{array}{ll}\Gamma \vdash S : & \star \\ \Gamma \vdash N : & S \to \star \\ \Gamma \vdash R : & R \to \star \\ \Gamma \vdash r : & \Pi\, s : S\,. \\ & \Pi\, n : N\ s\,. \\ & \Pi\, f : R\ s \to \langle S, N, R\rangle\,. \\ & (\Pi\, x : R\ s\,.\, C\ (f\ x)) \to \\ & C\ \langle s, n, f\rangle \\ \Gamma \vdash d : & \langle S, R, R\rangle\end{array}}{\Gamma \vdash r \bullet d : C\ d}$$

4.4 An extensional equality type

We continue our extension of the syntax of types and type-annotated terms with the following additions:

$$\begin{array}{lcllll} Ty & \ni & R, S, T & ::= & \cdots \mid T\{t == t'\} \\ Tm & \ni & r, s, t & ::= & \cdots \mid \epsilon\ t \mid \rho\,[x\,.\,T]\ t - t' \mid \xi\ t \end{array}$$

The type $T\{t == t'\}$ expresses extensional equality of terms t and t' at type T. Syntactically, it binds more tightly than Π, so an expression like $\Pi\, x : T\,.\, T'\{s == t\}$ should be parsed as $\Pi\, x : T\,.\, (T'\{s == t\})$. The new term constructs are for inference rules about this equality: reflexivity ($\epsilon\ t$), congruence ($\rho\,[x\,.\,T]\ t - t'$), and extensionality ($\xi\ t$).

The typing and kinding rules are shown in Figure 8. The first rule states that $\epsilon\ t$ proves that t equals itself at its type. The second rule, for typing terms of the form $\rho\,[x\,.\,T]\ s - t$, allows us to change the type of some term t by replacing chosen occurrences of t_1 with t_2, when we have a proof s that t_1 and t_2 are equal at type T'. The occurrences to be replaced are indicated using a variable x to show where in some type T the occurrences of t_1 are to be exchanged for t_2. That variable x has to have the same type T' for typing T, otherwise T might be using t_1 and t_2 at different types than the ones for which they are proved equal. For example, if we prove that t_1 equals t_2 at type $\mathit{False} \to T$, where *False* is uninhabited, then we are not allowed to replace t_1 except at that same type. Otherwise we could unsoundly change t_1 to t_2 that is not equivalent at the type where it is used.

$$\frac{\Gamma \vdash s : T \quad \Gamma \vdash t : T}{\Gamma \vdash T\{s == t\} : \star} \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash \epsilon\ t : T\{t == t\}}$$

$$\frac{\Gamma \vdash s : T'\{t_1 == t_2\} \quad \Gamma \vdash t : [t_1/x]T \quad \Gamma, x : T' \vdash T : \star}{\Gamma \vdash \rho\,[x\,.\,T]\ s - t : [t_2/x]T} \qquad \frac{\Gamma \vdash t : \Pi\, x : T'\,.\,T\{t_1\ x == t_2\ x\}}{\Gamma \vdash \xi\, t : (\Pi\, x : T'\,.\,T)\{t_1 == t_2\}}$$

$$\frac{\Gamma \vdash t : \Pi\, X : \kappa\,.\,T\{t_1 == t_2\}}{\Gamma \vdash \xi\, t : (\Pi\, X : \kappa\,.\,T)\{t_1 == t_2\}} \qquad \frac{\Gamma \vdash t_1 : T\{r == s\} \quad \Gamma \vdash t_2 : T\{r == s\}}{\Gamma \vdash \xi\, t : (T\{r == s\})\{t_1 == t_2\}}$$

Figure 8: Typing and kinding rules for equality

The last three rules are for typing uses $\xi\, t$ of extensionality. Two of them express the idea that if functions are equal for all inputs, then they are equal at function type. There is one rule for equality of term inputs, and another for equality of type inputs. Since we work in a Curry-style theory, the premise of the rule for equality of type inputs just requires that t_1 equals t_2, because in Curry-style type theory, we do not apply terms to type arguments. Church-style type theory would have had $\Gamma \vdash t : \Pi\, X : \kappa\,.\,T\{t_1\ X == t_2\ X\}$ instead. The very last rule of the figure expresses extensionality for equality types themselves, by stating that all proofs of equality are themselves equal: t_1 and t_2 are both proofs of some equation $T\{r == s\}$, then they are equal at that type. It should be noted that this principle is incompatible with Homotopy Type Theory, which leaves open the possibility of other proofs of equalities besides just reflexivity [4]. If desired, the axiom could be omitted.

Finally, we extend the erasure function to the new term constructs for equality, as follows:

$$\begin{array}{lcl} |\epsilon\ t| & = & \lambda\, x\,.\,x \\ |\rho\,[x\,.\,T]\ s - t| & = & |t| \\ |\xi\, t| & = & |t| \end{array}$$

The critical point here is that when equality proofs are used to change the type of a term, with the $\rho\,[x\,.\,T]\ s{-}t$ construct, the proof s of the equality is discarded by erasure. This makes a huge difference in working with terms where equality proofs are used to change the types of subterms, because in Church-style type theory, one can end up with stuck coercions: an equality proof is being used to change the type of some subterm, but it ends up blocking reduction or reasoning about the term. This can happen if the equality proof either is an assumption (i.e., a variable) or derived from one. One could have a stuck coercion around a λ-abstraction, for example, that is being applied to an argument. The coercion would block the β-reduction step, because it is in between the function and the argument. With the proposed approach, however, the equality proofs are all erased, and such stuck terms do not arise.

5 Examples

5.1 Symmetry and transitivity of equality

6 Metatheory of Bend's programming language

7 Metatheory of Bend's type system

References

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

[2] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

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

[4] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.