Type in Type and Schematic Affine Recursion

Aaron Stump and Victor Taelin

September 27, 2025

1 Terminating Computation First

Constructive type theories based on the Curry-Howard isomorphism enforce logical soundness by ensuring that all programs are uniformly terminating. Proofs are identified with programs, and diverging programs would thus prove arbitrary propositions. So these must be ruled out statically. The approach adopted in systems like Coq, Agda, and Lean is to enforce termination through a combination of typing and syntactic checks for structural decrease at recursive calls.

This paper proposes an alternative, where termination is enforced through syntactic checks alone, prior to typing. The approach has the drawback that programs must be significantly restricted in order to guarantee termination without any reference to types. There is a notable benefit, however: the type system is now no longer required to enforce termination. This greatly increases the options for the typing relation, which now needs only to satisfy type safety, as required in Programming Languages.

We present a language SAR, which combines an untyped affine lambda calculus with a form of structural recursion. With no further restriction, this language allows diverging terms. So we impose what Alves et al. call the "closed-by-construction" restriction on structural recursion [1], found also in [3]. This requires that the functions to be iterated when recursing are closed. But this rules out most of the usual higher-order functions like `map` on lists, where the function to iterate calls a function f that is given as a variable bound outside the recursion. We address this problem by proposing a language of schematic terms, where such variables f are not λ-bound, but treated schematically. This allows generic definition of functions like `map`, without losing termination.

With termination of SAR established prior to typing, we are free to adopt a more exotic type system than possible in other constructive type theories, where the burden of termination falls on typing. To demonstrate this, we consider a dependent type system called Typed SAR, with the $\star : \star$ principle.

2 Previous work on termination with affine recursion

Affine lambda calculus restricts λ-abstractions so that the λ-bound variable may occur at most once in the body of the abstraction. With this restriction, β-reduction is easily seen to be terminating, as the number of applications decreases by at least one with every β-step.

$$\textit{Terms} \quad t \quad ::= \quad x \mid \lambda\, x\,.\, t \mid t\ t' \mid \mathsf{Nil} \mid \mathsf{Cons}\ t_1\ t_2 \mid \mathsf{R}\ t_1\ t_2\ t_3$$

Figure 1: The syntax of untyped terms

$$\begin{array}{lcl}
(\lambda\, x\,.\, t)\ t' & \leadsto & [t'/x]t \\
\mathsf{R}\ \mathsf{Nil}\ t_1\ t_2 & \leadsto & t_2 \\
\mathsf{R}\ (\mathsf{Cons}\ t_a\ t_b)\ t_1\ t_2 & \leadsto & t_1\ t_a\ t_b\ (\mathsf{R}\ t_b\ t_1\ t_2)
\end{array}$$

Figure 2: Reduction semantics

But affine lambda calculus seems too restrictive for regular programming. For example, Church-encoded data such as natural numbers and lists are not affine in general. So we consider expanding the language with some form of inductive datatype, and its structural recursor. For simplicity, in this paper we just consider a datatype of lists, with its recursor. The syntax is shown in Figure 1, and its reduction semantics in Figure 2.

Without some further restriction, this language is easily seen to allow diverging terms. Let us see an example, adapted from [1]. Define

$$\begin{array}{rcl}
\mathtt{app}_t & = & \lambda\, y\,.\, y\ t \\
\mathtt{apply} & = & \lambda\, a\,.\, \lambda\, b\,.\, a\ b
\end{array}$$

where $\mathtt{app}_t$ is schematic in meta-variable t. Then we have this reduction sequence, for any term t:

$$\begin{array}{ll}
\mathtt{app}_t\ (\mathtt{app}_t\ \mathtt{apply}) & \leadsto^2 \\
\mathtt{app}_t\ (\lambda\, b\,.\, t\ b) & \leadsto^2 \\
t\ t &
\end{array}$$

The example is easier to complete with an iterator, defined as

$$\mathtt{It} = \lambda\, n\,.\, \lambda\, f\,.\, \lambda\, x\,.\, \mathsf{R}\ n\ (\lambda\, q\,.\, \lambda\, r\,.\, f)\ x$$

This uses R to repeat the function f, but dropping the head and tail of the list that R supplies to its second argument in case the first is a Cons. Using $\mathtt{It}$, define

$$\begin{array}{rcl}
\mathtt{u} & = & \lambda\, x\,.\, x \\
\mathtt{T} & = & \mathsf{Cons}\ u\ (\mathsf{Cons}\ u\ \mathsf{Nil}) \\
\delta & = & \lambda\, x\,.\, \mathtt{It}\ \mathtt{T}\ \mathtt{app}_x\ \mathtt{apply}
\end{array}$$

$\mathtt{T}$ is a list of length two, and applying $\mathtt{It}$ with it will lead to two nested calls. So we have

$$\begin{array}{ll}
\mathtt{It}\ s\ \mathtt{app}_x\ \mathtt{apply} & \leadsto^+ \\
\mathtt{app}_x\ (\mathtt{app}_x\ \mathtt{apply}) & \leadsto^+ \\
x\ x &
\end{array}$$

So the term $\Omega = \delta\ \delta$ is not normalizing, because δ reduces to $\lambda\, x\,.\, x\ x$.

This example is not so surprising, since the reduction rule for R is (at the meta-level) not affine. What is more surprising is that the example still is diverging if one adopts what Alves et al. call the "closed at reduction" restriction on the reduction rules for R:

$$\begin{array}{lcll}
\mathsf{R}\ \mathsf{Nil}\ t_1\ t_2 & \leadsto & t_2, & \text{if } FV(t_1) = \emptyset \\
\mathsf{R}\ (\mathsf{Cons}\ t_a\ t_b)\ t_1\ t_2 & \leadsto & t_1\ t_a\ t_b\ (\mathsf{R}\ t_b\ t_1\ t_2), & \text{if } FV(t_1) = \emptyset
\end{array}$$

With the "closed at reduction" restriction, the former reduction sequence is not available. But we still have this one:

$$\begin{array}{ll}
\delta\ \delta & \leadsto \\
\mathtt{It}\ \mathtt{T}\ \mathtt{app}_\delta\ \mathtt{apply} & \leadsto^+ \\
\mathtt{app}_\delta\ (\mathtt{app}_\delta\ \mathtt{apply}) & \leadsto^+ \\
\delta\ \delta &
\end{array}$$

showing that Ω diverges.

Alves et al. observe that if one goes even further and restricts the syntax so that $\mathsf{R}\ t\ t_1\ t_2$ is only syntactically allowed if t_1 is closed, then the language is indeed terminating. They call this the "closed at construction" restriction. It significantly reduces the computational power of the language: only the primitive recursive functions are definable [1].

While limiting oneself to primitive recursive functions may be a concern for theoretical applications, it is not a concern for practical programming, where the primitive recursive functions already encompass computations far beyond the feasible. But "closed at construction" does have a serious practical drawback: it prevents the usual higher-order combinators one expects in functional programming. For example, using the unrestricted syntax, we may define a `map` function on lists:

$$\texttt{map} = \lambda\, f\,.\,\lambda\, x\,.\,\mathsf{R}\ x\ (\lambda\, h\,.\,\lambda\, t\,.\,\lambda\, r\,.\,\mathsf{Cons}\ (f\ h)\ r)\ \mathsf{Nil}$$

Here, R is used to recurse through list x, applying f to the head h of each sublist. But the term performing that application has f free. So this program, along with many others from usual functional programming practice, will be disallowed by affine lambda calculus with "closed at construction" recursion.

3 Schematic affine recursion

We can address this difficulty by applying a basic idea that one finds in both Logic and Programming Languages, namely the use of schematic constructions. For the definition of Peano Arithmetic in first-order logic, one postulates a *scheme* of induction. This is a meta-level formulation representing an infinite set of axioms, each one expressing the induction principle for proving $\forall\, x\,.\,\phi$ from base and step cases. This is in contrast to second-order logic, where a single axiom suffices, by quantifying, within the logic, over the formula ϕ. In contrast, the first-order scheme of induction is a meta-level quantification, outside the logic, over ϕ.

We can use this same method here to recover generic programming while respecting the "closed at construction" requirement. We extend the syntax of Figure 1 to allow top-level schematic definitions of terms. This is shown in Figure 3. Schematic terms have the same syntax as terms above, except for the addition of the construct $\mathbf{f}[\bar{t}]$. Here, $\bar{t}$ is a (finite) vector of schematic terms, and $\mathbf{f}$ is a defined symbol. Figure 4 defines a relation Wf on lists of definitions, imposing these requirements:

- The defined symbols occurring on the right-hand side of a definition are defined earlier in the list, with the same number of schematic variables $\bar{u}$ as the arguments $\bar{t}$ where that symbol is applied.
- In applications $\mathbf{f}[\bar{t}]$, the terms $\bar{t}$ are all required to be closed, which means that they have no free variables except for schematic ones. Those will only be instantiated by closed terms, so they may stand for such in applications of defined symbols.
- The term t_1 in a recursion $\mathsf{R}\ t_1\ t_2\ t_3$ is required to be closed.

These requirements are imposed using the following relations:

- $\mathit{Wf}\,\Delta$ expresses that the list Δ of schematic definitions is well-formed, in the sense that each definition in Δ is well-formed with respect to the definitions to the left of it in Δ.
- The relation $\Delta \vdash \mathit{Wf}\ D$ expresses that definition D is well-formed with respect to Δ. This requires, for definition $\mathbf{f}[\bar{u}] = t$, that the body t of the definition is closed with respect to the parameters $\bar{u}$, and (for simplicity) that $\mathbf{f}$ is not declared already in Δ.
- The relation $\Delta; \bar{u} \mid \bar{x} \vdash \mathit{Closed}\, t$ means that term t is closed with respect to the definitions in Δ (which are just used to check that defined symbols $\mathbf{f}$ are applied to the correct number of arguments), and the parameters $\bar{u}$. The parameters are considered closed themselves, as are the locally bound variables

$$\begin{array}{lccl}
\textit{Defined symbols} & \mathbf{f} & & \\
\textit{Schematic variables} & u & & \\
\textit{Schematic terms} & t & ::= & x \mid u \mid \mathbf{f}[\bar{t}] \mid \lambda\, x\,.\, t \mid t\ t' \mid \\
 & & & \mathsf{Nil} \mid \mathsf{Cons} \mid \mathsf{R} \\
\textit{Definitions} & D & ::= & \mathbf{f}[\bar{u}] = t \\
\textit{Lists of definitions} & \Delta & ::= & \cdot \mid \Delta, D
\end{array}$$

Figure 3: Syntax of SAR

$$\frac{}{\textit{Wf}\ \cdot} \qquad \frac{\textit{Wf}\ \Delta \quad \Delta \vdash \textit{Wf}\ D}{\textit{Wf}\ (\Delta,\ D)}$$

$$\frac{\begin{array}{c}\Delta; \bar{u} \mid \cdot \vdash \textit{Closed}\ t \\ \forall\, \bar{u}'\,.\, \forall\, t'\,.\, (\mathbf{f}[\bar{u}'] = t') \notin \Delta\end{array}}{\Delta \vdash \textit{Wf}\ \ \mathbf{f}[\bar{u}] = t} \qquad \frac{u \in \{\bar{u}\}}{\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ u} \qquad \frac{x \in \{\bar{x}\}}{\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ x}$$

$$\frac{\Delta; \bar{u} \mid \bar{x}, y \vdash \textit{Closed}\ t}{\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ \lambda\, y\,.\, t} \qquad \frac{\begin{array}{c}\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ t_1 \\ \Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ t_2\end{array}}{\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ (t_1\ t_2)} \qquad \frac{}{\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ \mathsf{Nil}}$$

$$\frac{\begin{array}{c}\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ t_1 \\ \Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ t_2\end{array}}{\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ \mathsf{Cons}\ t_1\ t_2} \qquad \frac{\begin{array}{c}\Delta; \bar{u} \mid \cdot \vdash \textit{Closed}\ t_1 \\ \Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ t_2 \\ \Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ t\end{array}}{\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ \mathsf{R}\ t_1\ t_2\ t} \qquad \frac{\begin{array}{l}(\mathbf{f}[\bar{u}] = t) \in \Delta \\ |\bar{u}| = |\bar{t}| \\ \Delta; \bar{u} \mid \cdot \vdash \textit{Closed}\ \bar{t}\end{array}}{\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ \mathbf{f}[\bar{t}]}$$

$$\frac{}{\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ \cdot} \qquad \frac{\begin{array}{c}\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ t \\ \Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ \bar{t}\end{array}}{\Delta; \bar{u} \mid \bar{x} \vdash \textit{Closed}\ (t, \bar{t})}$$

Figure 4: Well-formedness of definitions and lists of definitions; closed schematic terms and lists of such terms

$\bar{x}$. Those variables are dropped, however, in premises (of the rules of Figure 4) where terms must not contain any free local variables. Dropping them enforces that the terms in question cannot contain free local variables. This can be seen where $\bar{x}$ is replaced with $\cdot$ in premises to the rules expressing closedness of R-terms and applications $\mathbf{f}[\bar{t}]$ of defined symbols.

3.1 Example

We may define higher-order functions like `map` using a schematic definition:

$$\texttt{map}[f] = \lambda\, x\,.\, \mathsf{R}\ x\ (\lambda\, h\,.\, \lambda\, t\,.\, \lambda\, r\,.\, \mathsf{Cons}\ (f\ h)\ r)\ \mathsf{Nil}$$

Call this definition $D_{\texttt{map}}$. We have $\textit{Wf}\ D_{\texttt{map}}$, because the body of the definition, namely

$$\mathsf{R}\ x\ (\lambda\, h\,.\, \lambda\, t\,.\, \lambda\, r\,.\, \mathsf{Cons}\ (f\ h)\ r)\ \mathsf{Nil}$$

is closed with respect the parameter f. Being closed means that the term iterated by R should have no free variables. Here, this is the term beginning $\lambda\, h$. It may, however, use the parameter f, as indeed it does.

$$\begin{array}{lcll}
(\lambda\, x\, .\, t)\ t' & \rightsquigarrow_\Delta & [t'/x]t & \\
\mathsf{R}\ \mathsf{Nil}\ t_1\ t_2 & \rightsquigarrow_\Delta & t_2 & \\
\mathsf{R}\ (\mathsf{Cons}\ t_a\ t_b)\ t_1\ t_2 & \rightsquigarrow_\Delta & t_1\ t_a\ t_b\ (\mathsf{R}\ t_b\ t_1\ t_2) & \\
\mathsf{f}[\bar{t}] & \rightsquigarrow_\Delta & [\bar{t}/\bar{u}]t & \text{if } (\mathsf{f}[\bar{u}] = t) \in \Delta \text{ and } |\bar{u}| = |\bar{t}|
\end{array}$$

Figure 5: Reduction with respect to definitions

It is possible to define the problematic term δ from Section 2 above, as a schematic term. We first must define **It** schematically:

$$\mathbf{It}[f] = \lambda\, n\, .\, \lambda\, x\, .\, \mathsf{R}\ n\ (\lambda\, q\, .\, \lambda\, r\, .\, f)\ x$$

This is necessary so that the second argument to R will be closed except for parameters (here, f). Note that this definition reverses the order of the first two arguments to **It**, compared to the definition in Section 2. Now using **It**, we can define δ schematically, where **T** is, as above, a list of length two:

$$\begin{array}{lcl}
\mathbf{u} & = & \lambda\, x\, .\, x \\
\mathbf{T} & = & \mathsf{Cons}\ \mathbf{u}\ (\mathsf{Cons}\ \mathbf{u}\ \mathsf{Nil}) \\
\delta[u] & = & \mathbf{It}[\mathsf{app}_u]\ \mathbf{T}\ \mathtt{apply}
\end{array}$$

This is a well-formed definition, because the argument supplied for the parameter of **It** is closed with respect to parameter u.

So we could define **It** and δ. What breaks down is the definition of Ω. Here, as δ is a schematic term, it must be supplied with a closed term. We cannot write $\delta[\delta]$ because δ cannot be used as an argument: it is not a closed term, but requires a closed term as an argument for its parameter. This shows that restricting terms to be closed with respect to schematic definitions blocks the counterexample above to termination.

3.2 Reduction

Figure 5 defines a reduction semantics for terms with respect to a list Δ of definitions. The notation $t \rightsquigarrow_\Delta t'$ means that t reduces to t' possibly using the definitions in Δ. The rules are the same as in Figure 2 above, except that we add the last rule for looking up a definition from Δ.

Definition 1. *A term t is a value with respect to a list Δ of definitions iff t contains no subterms of any of these forms:*

- $(\lambda\, x\, .\, t)\ t'$
- $\mathsf{R}\ \mathsf{Nil}\ t_1\ t_2$
- $\mathsf{R}\ (\mathsf{Cons}\ t_a\ t_b)\ t_1\ t_2$
- $\mathsf{f}[\bar{t}]$

The following should be proved in detail to establish the intended consequences of the above definitions.

Proposition 2 (Safety)**.** *If Wf Δ and $\Delta; \cdot \mid \cdot$ Closed t, then either there exists t' such that $t \rightsquigarrow_\Delta t'$, or t is a value.*

Proposition 3 (Normalization)**.** *If Wf Δ and $\Delta; \cdot \mid \cdot$ Closed t, then there exists value t' such that $t \rightsquigarrow^*_\Delta t'$.*

3.3 Questions

1. Could SAR use λ-encodings instead of a built-in recursor? The usual λ-encodings like the Church and Parigot encodings represent data using non-linear λ-terms in general. The Scott encoding is affine, and even one can derive recursion on it in untyped λ-calculus. But that derivation, which goes through the exotic Lepigre-Rafalli encoding, is nonlinear [2, Section 7]. So it seems that the only way to use λ-encodings (and gain their benefits for keeping the language design concise) would be to impose some additional discipline that would allow non-affine terms but restrict them in some different way to prevent divergence. But it is unclear how that would work.

4 Typing without termination

In this section, we define Typed SAR, which is a typed version of the SAR language just presented. In Typed SAR, we write terms with some type annotations. This makes it possible to compute types for such terms. Erasing these annotations results in a term of SAR. We may then require both that a term is typable and that its erasure is closed, in the sense of the previous section. This ensures that the term is typable and normalizing. A simple argument then shows that there is an uninhabited type, thus proving the theory consistent. We begin with the syntax of annotated terms, including types.

4.1 Annotated terms

The syntax of annotated terms for Typed SAR is shown in Figure 6. We add several groups of new notations.

- Constructs from pure dependent type theory:
  - $\star$, the type for types;
  - dependent function types $\Pi\, x : M \,.\, M'$;
  - erased dependent function types $\forall\, x : M \,.\, M'$, which introduce a variable $x : M$ that may then be used in the body M' without corresponding to an input to a function; and
  - erased applications M -N, giving an instantiation to a $\forall$-bound variable.
- Constructs for typed extensional equality:
  - $\epsilon\ M$ a proof of reflexivity
  - $\rho\,[x\,.\,T]\ M - M'$ a substitution principle, allowing to replace a term by an equal one in a type
  - $\xi\, M$ an extensionality principle
  - $T\{M = M'\}$ expressing that M and M' are equal at type T
- Additionally we have:
  - annotations on λ-bound variables;
  - annotations on the parameters to $\bar{u}$ in definitions; and
  - the type List for lists.

4.2 Typing rules

The typing rules for Typed SAR are listed in several figures: rules for pure dependent type theory with $\star : \star$ in Figure 7, types for list constants in Figure 8, and rules for equality in Figure 9. The conversion rule, labeled (c) in Figure 7, references the erasure $|M|$ of terms M. Erasure is defined in Figure 10. Its use in the conversion rule is what makes this a Curry-style dependent type theory, because terms are considered equal regardless of differences in their annotations. Erasure drops the equality proof from uses of substitution (ρ-terms). It also drops implicit arguments; i.e., N in M -N. Annotations on bound variables are also dropped. It otherwise proceeds homomorphically throughout terms. Note that the erasure of a term might not be a schematic affine term in the sense of Section 3, both because the type system does not enforce the

$$
\begin{array}{llll}
\textit{Constants} & C & ::= & \mathsf{Nil} \mid \mathsf{Cons} \mid \mathsf{R} \mid \mathsf{List} \\
\textit{Annotated terms} & M, N & ::= & x \mid u \mid C \mid \star \mid \mathbf{f}[\bar{M}] \mid \\
 & & & \lambda\, x : M \,.\, N \mid \Lambda\, x : M \,.\, N \mid M\ N \mid M\ \text{-}N \\
 & & & \Pi\, x : M \,.\, N \mid \forall\, x : M \,.\, N \mid \\
 & & & \epsilon\ M \mid \rho\,[x \,.\, T]\ M - M' \mid \xi\, M \mid T\{M = M'\} \\
\textit{Annotated definitions} & D & ::= & \mathbf{f}[u_1 : M_1, \cdots, u_k : M_k] = t \\
\textit{Lists of definitions} & \Delta & ::= & \cdot \mid \Delta, D
\end{array}
$$

Figure 6: Annotated terms of Typed SAR

$$
\frac{\Gamma(x) = M}{\Gamma \vdash x : M}
\qquad
\frac{\Gamma, x : M \vdash N : N'}{\Gamma \vdash \lambda\, x : M \,.\, N : \Pi\, x : M \,.\, N'}
$$

$$
\frac{\Gamma \vdash M : \Pi\, x : N' \,.\, M' \quad \Gamma \vdash N : N'}{\Gamma \vdash M\ N : [N/x]M'}
\qquad
\frac{\Gamma, x : M \vdash N : N' \quad x \notin FV(|N|)}{\Gamma \vdash \Lambda\, x : M \,.\, N : \forall\, x : M \,.\, N'}
$$

$$
\frac{\Gamma \vdash M : \forall\, x : N' \,.\, M' \quad \Gamma \vdash N : N'}{\Gamma \vdash M\ \text{-}N : [N/x]M'}
\qquad
\frac{\Gamma \vdash M : N' \quad |N| =_{\beta,\eta} |N'| \quad \Gamma \vdash N : \star}{\Gamma \vdash M : N}\ (c)
$$

$$
\frac{}{\Gamma \vdash \star : \star}
$$

Figure 7: Rules from pure dependent theory

affine property, and also because erasure might not remove some constructs like Π-types from terms. While Π-types used as annotations would all be removed, such constructs could also appear as arguments in terms. Those occurrences would not be removed by erasure. If one wanted to enforce that conversion is terminating, this point would have to be addressed, either by relaxing the notion of affine to permit extra occurrences in constructs like Π-types, or strengthening the notion of erasure to remove all such constructs. For now, we simply do not concern ourselves with termination of conversion (as we also do not concern ourselves with feasibility of conversion: terminating terms of SAR could still be infeasible to normalize).

Let us consider the constructs for equality, with typing rules in Figure 9, in more detail.

- The $\rho\,[x\,.\,T]\ M - M'$ construct is used with a proof M that two terms M_1 and M_2 are equal at some type T'. The $x.T$ part of the construct shows (by occurrence of bound variable x) which occurrences of M_1 are to be replaced with M_2 in the type of M'. The rule requires that T is typable assuming $x : T'$. This is to prevent applying an equality at type T' to replace an occurrence of M_1 where that occurrence is being used at a different type (not T'). Since this is Curry-style type theory, the same term could be used at different types, because identity of terms depends on their erasures only.
- The $\xi\, M$ construct is for extensionality. There are three different typing rules, for applying extensionality at Π-type, $\forall$-type, or the equality type itself.

4.3 Examples

In this section, we show examples that are permitted by the type system and whose erasures satisfy the requirements of schematic affine recursion.

$$\begin{array}{lll}
\mathsf{Nil} & : & \forall\, A : \star \,.\, \mathsf{List}\ A \\
\mathsf{Cons} & : & \forall\, A : \star \,.\, A \to \mathsf{List}\ A \to \mathsf{List}\ A \\
\mathsf{List} & : & \star \to \star \\
\mathsf{R} & : & \forall\, A : \star . \forall\, C : \mathsf{List}\ A \to \star\,. \\
 & & (\Pi\, h : A \,.\, \Pi\, t : \mathsf{List}\ A \,.\, C\ t \to C\ (\mathsf{Cons}\ \text{-}A\ h\ t)) \to \\
 & & C\ (\mathsf{Nil}\ \text{-}A) \to \\
 & & \Pi\, x : \mathsf{List}\ A \,.\, C\ x
\end{array}$$

Figure 8: Types for list constants

$$\frac{\begin{array}{c}\Gamma \vdash M : \star \\ \Gamma \vdash N_1 : M \\ \Gamma \vdash N_2 : M\end{array}}{\Gamma \vdash M\{N_1 = N_2\} : \star}
\qquad
\frac{\begin{array}{c}\Gamma \vdash M : T'\{M_1 = M_2\} \\ \Gamma, x : T' \vdash T : \hat{T} \\ \Gamma \vdash M' : [M_1/x]T\end{array}}{\Gamma \vdash \rho\,[x \,.\, T]\ M - M' : [M_2/x]T}$$

$$\frac{\Gamma \vdash M : \Pi\, x : T \,.\, (T'\{M_1\ x = M_2\ x\})}{\Gamma \vdash \xi\, M : (\Pi\, x : T \,.\, T')\{M_1 = M_2\}}
\qquad
\frac{\begin{array}{c}\Gamma \vdash M_1 : T\{M_a = M_b\} \\ \Gamma \vdash M_2 : T\{M_a = M_b\}\end{array}}{\Gamma \vdash \xi\, M : (T\{M_a = M_b\})\{M_1 = M_2\}}$$

$$\frac{\Gamma \vdash M : \forall\, x : T \,.\, (T'\{M_1\ x = M_2\ x\})}{\Gamma \vdash \xi\, M : (\forall\, x : T \,.\, T')\{M_1 = M_2\}}
\qquad
\frac{\Gamma \vdash M : T}{\Gamma \vdash \epsilon\ M : T\{M = M\}}$$

Figure 9: Rules for equality

$$\begin{array}{lcl}
|C| & = & C \\
|\star| & = & \star \\
|\Pi\, x : M \,.\, M'| & = & \Pi\, x : |M| \,.\, |M'| \\
|\forall\, x : M \,.\, M'| & = & \forall\, x : |M| \,.\, |M'| \\
|x| & = & x \\
|M\ \text{-}N| & = & |M| \\
|M\ N| & = & |M|\ |N| \\
|\lambda\, x : M \,.\, M'| & = & \lambda\, x \,.\, |M'| \\
|\Lambda\, x : M \,.\, M'| & = & |M'| \\
|M\{M_1 = M_2\}| & = & |M|\{|M_1| = |M_2|\} \\
|\epsilon\ M| & = & \epsilon\ |M| \\
|\rho\,[x \,.\, T]\ M - M'| & = & |M'| \\
|\xi\, M| & = & \xi\, |M|
\end{array}$$

Figure 10: Erasing annotations

4.3.1 Basic datatypes

Let us make the following simple definitions, to have some basic datatypes to use:

$$\begin{array}{lcll}\mathbf{False} & : & \star & := & \forall\, X : \star\,.\, X \\ \mathbf{Unit} & : & \star & := & \mathsf{List}\ \mathbf{False} \\ \mathbf{Nat} & : & \star & := & \mathsf{List}\ \mathbf{Unit}\end{array}$$

Now we may define some terms of these types. Notice that there is just one inhabitant of the **Unit** type in a consistent context, because we cannot apply Cons **-False** to build any other inhabitant than just Nil **-False**:

$$\begin{array}{lcll}\mathbf{unit} & : & \mathbf{Unit} & := & \mathsf{Nil}\ \text{-}\mathbf{False} \\ \mathbf{zero} & : & \mathbf{Nat} & := & \mathsf{Nil}\ \text{-}\mathbf{Unit} \\ \mathbf{succ} & : & \mathbf{Nat} \to \mathbf{Nat} & := & \mathsf{Cons}\ \text{-}\mathbf{Unit}\ \mathbf{unit}\end{array}$$

4.3.2 Type-level data structures

Thanks to the $\star : \star$ axiom (Figure 7), we can form lists of types; for example:

$$\mathsf{Cons}\ \text{-}\star\ \mathbf{Unit}\ (\mathsf{Cons}\ \text{-}\star\ \mathbf{Nat}\ (\mathsf{Nil}\ \text{-}\star))$$

4.3.3 A congruence principle

As an example of equational reasoning, let us may derive this congruence principle:

$$\begin{array}{lcl}\mathbf{cong} & : & \forall\, X : \star\,.\forall\, Y : \star\,.\forall\, f : X \to Y\,.\forall\, x : X\,.\forall\, x' : X\,. \\ & & \quad X\{x = x'\} \to \\ & & \quad Y\{f\ x = f\ x'\}\end{array}$$

This states that if x equals x', then $f\ x$ equals $f\ x'$. The proof is

$$\begin{array}{l}\Lambda\, X : \star\,.\,\Lambda\, Y : \star\,.\,\Lambda\, f : X \to Y\,.\,\Lambda\, x : X\,.\,\Lambda\, x' : X\,. \\ \quad \lambda\, p : X\{x = x'\}\,.\,\rho\,[y\,.\,Y\{f\ x = f\ y\}]\ p - \epsilon\ (f\ x)\end{array}$$

This works because:

- $\epsilon\ (f\ x)$ has type $Y\{f\ x = f\ x\}$
- then the ρ-term is saying that because p proves $X\{x = x'\}$, we may replace the right occurrence of x in $Y\{f\ x = f\ x\}$ (as derived by ϵ) with x'.

4.3.4 Inductive reasoning

Consider these definitions:

$$\begin{array}{lclcl}\mathbf{id} & : & \forall\, X : \star.\ \mathsf{List}\ X \to \mathsf{List}\ X & := & \Lambda\, X : \star\,.\,\lambda\, x : X\,.\, x \\ \mathbf{rebuild} & : & \forall\, X : \star.\ \mathsf{List}\ X \to \mathsf{List}\ X & := & \Lambda\, X : \star\,.\ \mathsf{R}\ \text{-}X\ \text{-}\lambda\, x : \mathsf{List}\ X\,.\,\mathsf{List}\ X\ (\mathsf{Cons}\ \text{-}X)\ \mathsf{Nil}\ \text{-}X\end{array}$$

The **rebuild** function recursively applies Cons to a list, starting from Nil. Notice that the instantiation of the variable C in the type of R is the constant constructor

$$\lambda\, x : \mathsf{List}\ X\,.\,\mathsf{List}\ X$$

We are using R for (non-dependently typed) computation, and are just trying to compute a $\mathsf{List}\ X$.

We can state that **rebuild** equals **id**, using our extensional equality:

$$(\forall\, X : \star.\ \mathsf{List}\ X \to \mathsf{List}\ X)\{\mathbf{rebuild} = \mathbf{id}\}$$

The proof of this is $\xi\,\xi\,\mathbf{pf}$ where we have this type and definition for the inductive **pf**, and helper predicate, and writing α as a label, for subsequent reference:

$$\begin{array}{lll}
\mathbf{pred} & : & \forall\, X : \star\,.\, \mathsf{List}\ X \to \star = \\
& & \Lambda\, X : \star\,.\, \lambda\, x : \mathsf{List}\ X\,.\, (\mathsf{List}\ X)\{\mathbf{rebuild}\ \text{-}X\ x = \mathbf{id}\ \text{-}X\ x\} \\
\\
\mathbf{pf} & : & \forall\, X : \star\,.\, \Pi\, x : X\,.\, \mathbf{pred}\ \text{-}X\ x = \\
& & \Lambda\, X : \star\,.\, \lambda\, x : X\,. \\
& & \quad \mathsf{R}\ \text{-}X\ \text{-}(\mathbf{pred}\ \text{-}X\ x) \\
& & \qquad {}^{\alpha}(\lambda\, h : X\,.\, \lambda\, t : \mathsf{List}\ X\,.\, \lambda\, ih : \mathbf{pred}\ \text{-}X\ t\,. \\
& & \qquad\quad \mathbf{cong}\ \text{-}\mathsf{List}\ X\ \text{-}\mathsf{List}\ X\ (\mathsf{Cons}\ \text{-}X\ h)\ ih) \\
& & \qquad \epsilon\ (\mathsf{Nil}\ \text{-}X)
\end{array}$$

The step case of the proof, labeled α, uses congruence to go from a proof (namely ih) of $\mathsf{List}\ X\{\mathbf{rebuild}\ \text{-}X\ t = \mathbf{id}\ \text{-}X\ t\}$, to a proof of the equality of those same terms except with $\mathsf{Cons}\ \text{-}X\ h$ applied to each. Those terms are equal to the desired ones, where **rebuild** and **id** are applied outside Cons instead of inside, by conversion.

References

[1] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. Gödel's system T revisited. *Theoretical Computer Science*, 411(11):1484–1500, 2010.

[2] Christopher Jenkins and Aaron Stump. Monotone recursive types and recursive data representations in cedille. *Math. Struct. Comput. Sci.*, 31(6):682–745, 2021.

[3] Ugo Dal Lago. The geometry of linear higher-order recursion. *ACM Trans. Comput. Log.*, 10(2):8:1–8:38, 2009.