Type in Type and Schematic Affine Recursion

Aaron Stump and Victor Taelin

August 31, 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

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

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:

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

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

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

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

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

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:

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

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

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

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:

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

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:

- *Wf* Δ 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

$$
\begin{array}{llll}
\textit{Defined symbols} & \mathbf{f} & & \\
\textit{Schematic variables} & u & & \\
\textit{Definitions} & D & ::= & \mathbf{f}[\bar{u}] = t \\
\textit{Schematic terms} & t & ::= & x \mid u \mid \mathbf{f}[\bar{t}] \mid \lambda\, x\,.\, t \mid t\ t' \mid \\
 & & & \mathsf{Nil} \mid \mathsf{Cons}\ t_1\ t_2 \mid \mathsf{R}\ t_1\ t_2\ t_3 \\
\textit{Lists of definitions} & \Delta & ::= & \cdot \mid \Delta, D
\end{array}
$$

Figure 3: Syntax of SAR

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

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

$$
\frac{\Delta;\bar{u} \mid \bar{x}, y \vdash \mathit{Closed}\,t}{\Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,\lambda\,y\,.\,t} \qquad \frac{\Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,t_1 \quad \Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,t_2}{\Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,(t_1\ t_2)} \qquad \frac{}{\Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,\mathsf{Nil}}
$$

$$
\frac{\Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,t_1 \quad \Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,t_2}{\Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,\mathsf{Cons}\ t_1\ t_2} \qquad \frac{\Delta;\bar{u} \mid \cdot \vdash \mathit{Closed}\,t_1 \quad \Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,t_2 \quad \Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,t}{\Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,\mathsf{R}\ t_1\ t_2\ t} \qquad \frac{(\mathbf{f}[\bar{u}] = t) \in \Delta \quad |\bar{u}| = |\bar{t}| \quad \Delta;\bar{u} \mid \cdot \vdash \mathit{Closed}\,\bar{t}}{\Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,\mathbf{f}[\bar{t}]}
$$

$$
\frac{}{\Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,\cdot} \qquad \frac{\Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,t \quad \Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,\bar{t}}{\Delta;\bar{u} \mid \bar{x} \vdash \mathit{Closed}\,(t, \bar{t})}
$$

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:

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

Call this definition $D_{\mathtt{map}}$. We have $\mathit{Wf}\,D_{\mathtt{map}}$, 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.

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

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, 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:

$$\delta[u] = \mathbf{It}[\mathtt{app}_u]\ T\ \mathtt{apply}$$

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 $t \rightsquigarrow_\Delta^* t'$.*

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

$$\textit{Annotated terms} \quad M, N \quad ::= \quad x \mid u \mid \mathbf{f}[\bar{M}] \mid \lambda\, x : M \,.\, N \mid \Lambda\, x : M \,.\, N \mid M\ N \mid \mathsf{Nil} \mid \mathsf{Cons}\ M\ N \mid \mathsf{R}\ M_1\ M_2\ N \mid \mathsf{List}\ M \mid \star$$

Figure 6: Annotated terms of Typed SAR

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. Somewhat surprisingly, the only additions needed for the syntax are:

- an annotation on λ-bound variables;
- an erased abstraction $\Lambda\, x : M \,.\, N$, used to introduce a specification variable x, which is computationally irrelevant;
- the type List for lists; and
- sort $\star$.

We will view types as abstractions of terms, and hence reuse λ-abstraction to represent an abstraction of a function, as opposed to introducing a new binder Π for this. The idea of using just one binder for functions and their abstractions has been studied previously by Kamareddine [2]. The typing relation should be viewed as an abstraction relation: the proposition $M : N$ will mean that N is an abstraction of M. It has discarded some information from M. Annotations on λ-bound variables are there to show which abstraction N should be computed from M.

To develop an example, let us first recall that we can view natural numbers as lists with trivial data. So we may define:

$$\begin{array}{lcl} \mathtt{Zero} & := & \mathsf{Nil} \\ \mathtt{Suc} & := & \mathsf{Cons}\ (\lambda\, x \,.\, x) \end{array}$$

Here, we use the identity function as the trivial data stored with each Cons node. Then we may define the length function on lists as a function which simply discards any data in the list:

$$\mathtt{length} := \mathtt{map}\ (\lambda\, h \,.\, \lambda\, x \,.\, x)$$

The type we intend to compute for an annotated version of `length` is the following, where we may define the type `Nat` of natural numbers to be $\mathsf{List}\ \lambda\, x \,.\, x$:

$$\Lambda\, A : \star \,.\, \lambda\, x : \mathsf{List}\ A \,.\, \mathtt{Nat}$$

An erased abstraction is used to introduce the type A into the rest of the expression. The expression can be understood as a function taking in an x of type $\mathsf{List}\ A$ and returning `Nat`, for any type A. Such a function is abstract, as the return value is always just `Nat` for every input, rather than some particular (represented) natural number. If we write $A \to B$ instead of $\lambda\, x : A \,.\, B$ when $x \notin FV(B)$, then the type we will give to `length` is:

$$\Lambda\, A : \star \,.\, \mathsf{List}\ A \to \mathtt{Nat}$$

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] Fairouz Kamareddine. Typed lambda-calculi with one binder. *J. Funct. Program.*, 15(5):771–796, 2005.

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