Type in Type and Schematic Affine Recursion

Aaron Stump and Victor Taelin

August 26, 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, found also in [2]. 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 "$\mathit{Type} : \mathit{Type}$" 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.

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

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

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

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

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

where app_t is schematic in meta-variable t. Then we have this reduction sequence, for any term t:

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

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

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

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 It, define

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

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

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

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:

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

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

Figure 3: Syntax of SAR

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:

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

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 **??** 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.

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

$$\frac{\Delta; \bar{u} \vdash Closed\ t}{\Delta \vdash Wf\ \ \mathbf{f}[\bar{u}] = t} \qquad \frac{u \in \{\bar{u}\}}{\Delta; \bar{u} \vdash Closed\ u}$$

Figure 4: Well-formedness of lists of definitions

4 Typing without termination

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