One Weird Trick to Untie Landin's Knot

PAULETTE KORONKEVICH, University of British Columbia, Canada
WILLIAM J. BOWMAN, University of British Columbia, Canada

In this work, we explore Landin's Knot, which is understood as a pattern for encoding general recursion, including non-termination, that is possible after adding higher-order references to an otherwise terminating language. We observe that this isn't always true—higher-order references, by themselves, don't lead to non-termination. The key insight is that Landin's Knot relies not primarily on references storing functions, but on unrestricted quantification over a function's environment. We show this through a closure converted language, in which the function's environment is made explicit and hides the type of the environment through impredicative quantification. Once references are added, this impredicative quantification can be exploited to encode recursion. We conjecture that by restricting the quantification over the environment, higher-order references can be safely added to terminating languages, without resorting to more complex type systems such as linearity, and without restricting references from storing functions.

ACM Reference Format:
Paulette Koronkevich and William J. Bowman. 2023. One Weird Trick to Untie Landin's Knot. In *Workshop on Higher-order Programming with Effects*. ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

How do we add *higher-order references* to an otherwise pure functional language? The problem is that higher-order references (that is, mutable references that can store higher-order functions) lead to Landin's Knot, which breaks strong normalization. While many languages value the expressivity of higher-order references, some languages, such as dependently-typed languages, value strong normalization as well. Existing languages often recover strong normalization by using *linear types*, which carefully track the usage of references in the type system [1, 6]. However, linear types are typically difficult to integrate into existing type systems, in particular, dependent types [9]. Quantitative Type Theory (QTT) [3] is one language that succeeds at integrating linear and dependent types, but it aims to provide a framework for reasoning about resources in general, which seems a heavyweight solution to ruling out non-termination.

To explore alternative methods for adding higher-order references to pure functional languages, let's study Landin's Knot in detail. Landin's Knot [7] refers to encoding recursion through backpatching—updating a mutable reference to create a cyclic data structure. In this case, the cyclic data structure happens to be a closure, and a reference is updated to contain the closure itself, enabling recursion. This idea is illustrated through the following program in the simply typed lambda calculus (STLC) with references, which diverges.

$$\begin{aligned}
&id : \mathbf{Nat} \to \mathbf{Nat} \\
&id = (\boldsymbol{\lambda} x.x) \\
&r : \mathbf{Ref}\ (\mathbf{Nat} \to \mathbf{Nat}) \\
&r = \mathbf{new}\ id \\
&f : \mathbf{Nat} \to \mathbf{Nat} \\
&f = (\boldsymbol{\lambda} x.((\mathbf{deref}\ r)\ x)) \\
&r := f; f\ 0
\end{aligned}$$

The function f closes over the reference r, and calls whatever function is stored in r. Initially, r contains an arbitrary function of the right type, but is later updated to contain f itself. After the update, f diverges when called.

The literature often attributes Landin's Knot merely to be due to references *storing* functions.

HOPE'24, September 4–9, 2022, Seattle, Washington, USA
2023.

> "... recursion can be encoded using function storage, as noted by Landin (folklore)." [8]
>
> "... in languages with higher-order store, it is usually possible to write recursion operators by backpatching function pointers." [6]

However, the actual cause of the recursion is due to the update to the *function's environment* through the store, not by storing the function in a reference. Even more interestingly, this update to the function's environment actually requires *impredicativity* to be well typed, but this impredicativity is implicit and hidden in the usual function type. This impredicativity exists in function types even if not explicit in the type system. We conjecture that restricting this impredicativity is one way to recover strong normalization, without the use of linear types.

We can make this observation more precise by closure converting the example. Closure conversion transforms functions into explicit closures, that is, pairs of closed procedure code with its environment. To type closures, we use the usual typing from Minamide et al. [10], where closures are typed as existential pairs, abstracting the type of the environment. The environment is a tuple of the free variables from the procedure code.

$$
\begin{array}{l}
id : \exists\, \alpha.\langle(\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha\rangle \\
id = \mathbf{pack}\langle\langle\rangle, \langle(\boldsymbol{\lambda} x.\boldsymbol{\lambda} env.x), \langle\rangle\rangle\rangle \\
r : \mathbf{Ref}\ (\exists\, \alpha.\langle(\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha\rangle) \qquad \texttt{-- abbreviated } \mathbf{Ref}_{\mathbf{type}} \texttt{ below} \\
r = \mathbf{new}\ id \\
f : \exists\, \alpha.\langle(\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha\rangle \\
f = \mathbf{pack}\langle\langle\mathbf{Ref}_{\mathbf{type}}\rangle, \langle(\boldsymbol{\lambda} x.\boldsymbol{\lambda} env : \langle\mathbf{Ref}_{\mathbf{type}}\rangle.\mathbf{let}\ r = \mathbf{proj_1}\ env\ \mathbf{in}\ (\mathbf{deref}\ r)\ x), \langle r\rangle\rangle\rangle \\
r := f; (\mathbf{proj_1}\ f)\ 0\ (\mathbf{proj_2}\ f)
\end{array}
$$

The functions f and id have the same type as closures, since the type of the environment is abstract. The r reference is then updated with a closure that contains r in its environment, and is well typed precisely because of the (implicit) impredicativity of the existential pair. The sort of universe of α is unrestricted, as is normal in System F, meaning the existentially quantification variable α can include the existential itself, or a reference that contains the existential.

2 BACKGROUND AND RELATED WORK

Related work modelling references often divide references into three categories. *Ground* references store only base types, *full-ground* [11] (or sometimes called recursive) references store base types and other references, and higher-order references store unrestricted higher-order functions. With the closure-converted Landin's Knot example, we see a middle ground between a terminating language with a full-ground store and a non-terminating language with a higher-order store.

The impredicativity requirement to type Landin's Knot has been recognized, but unexplored, in past work modeling languages with higher-order references. Levy [8] describes a possible world semantics to model languages with these three different kinds of references, and modeling functions in the store requires recursive domain equations. Kammar et al. [5] model full-ground references, but observe that in order to model higher-order references, recursive domain equations or step indexing is required. Ahmed [2] notes the circularity of modeling a store as mapping a location to a type, and that types are modelled as a predicates on stores, explicitly noting that the impredicative quantification of existential types is related to this circularity, but uses step-indexing to side-step this circularity. All this prior work describe possible models for higher-order references, but they do not investigate the alternative to impredicative quantification over a function's environment. Thus, our proposal is: design a language with higher-order references that is terminating by requiring predicativity with respect to environments and by interpreting the store inductively. The store at one level will depend only on the interpretation of terms at a lower universe, with a ground store as the base case.

3 PROPOSAL FOR LANGUAGE DESIGN

The closure-converted Landin's Knot shows us that the non-termination is due to updates to the function's environment. This guides us in our language design proposal; there needs to be some restriction on the types of the free variables a function closes over. We use the :: annotation to indicate the sort of a type.

To see how we might restrict the environment, consider the (perhaps overly) restrictive approach where we only allow full-ground references to appear in the environment. Below are a possible typing rule for closures and the (back-translated) equivalent rule for source language functions.

$$\frac{\Gamma \vdash \tau :: \textbf{full-ground} \qquad \Gamma \vdash e : \tau_1[\tau/\alpha]}{\Gamma \vdash \textbf{pack}\langle \tau, e \rangle : \exists\, \alpha.\tau_1} \qquad \frac{\Gamma \vdash FV(e) : \tau :: \textbf{full-ground} \ \cdots \qquad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \boldsymbol{\lambda} x : \tau_1.e : \tau_1 \rightarrow \tau_2}$$

In both rules, we use the sort to restrict the type of the environment. Either the type of the environment τ is restricted, or the types of free variables in the body of a function are restricted to be of a full-ground sort. Since closures are not ground types, Landin's Knot cannot be well typed. References can store closures, but the closures *themselves* are restricted. In a source language with references, this restricts the free variables in the function body to be full-ground. The downside of this approach is that higher-order closures would not be possible to encode, assuming that all closures must be allocated in mutable cells (heap allocated).

An alternative approach is to restrict the quantification over environments in closures. Each environment is typed at a the highest universe level of the types in the environment.

$$\frac{\Gamma \vdash \tau :: \textbf{Type}_j \qquad \Gamma \vdash e : \tau_1[\tau/\alpha]}{\Gamma \vdash \textbf{pack}\langle \tau, e \rangle : \exists\, \alpha : \textbf{Type}_j.\tau_1} \qquad \frac{\Gamma, \alpha : \textbf{Type}_j \vdash \tau_1 :: \textbf{Type}_k}{\Gamma \vdash \exists\, \alpha : \textbf{Type}_j.\tau_1 :: \textbf{Type}_k}$$

$$\frac{\Gamma \vdash FV(e) : \tau :: \textbf{Type}_i \cdots \qquad \textbf{max}(\ldots i \ldots) = j \qquad \Gamma, x : \tau_1 \vdash e : \tau_2 :: \textbf{Type}_k}{\Gamma \vdash \boldsymbol{\lambda} x : \tau_1.e : \tau_1 \rightarrow \tau_2 :: \textbf{Type}_j}$$

Existential types in the target language are standard—impredicative, which is necessary to type higher-order closures, but with an explicit sort annotation on the existentially bound variable. Note that for closures in particular, because the environment shows up in type $\tau_1 = C \times \alpha$, a closure with environment $\alpha : \textbf{Type}_j$ must live in $\textbf{Type}_j$ as well. The means closures can capture other closures, including closures of the same type, as is expected by closure conversion.

Functions are impredicative in their parameter, but predicative in the types of their environment variables; we've combined the typing and sort of functions above to express this. This makes the sort of the environment explicit in the source language typing rule, and forces a function's sort to be the same as its environment, as in the closure converted language.

Finally, we add the following sort rule for references, which breaks the circularity when a reference contains a closure, preventing an environment from containing a reference that contains its own type.

$$\frac{\Gamma \vdash A :: \textbf{Type}_i}{\Gamma \vdash \textbf{Ref}\ A :: \textbf{Type}_{i+1}}$$

Base types are at level 0, but references bump up the level of the stored type by 1. A reference cannot be updated to contain a closure that contains itself, since the type level of the environments would be necessarily lower than the type of the reference that must contain the environment. A closure can capture its own type in its environment, but it *cannot* capture a **Ref** of its own type.[1]

[1]We give a full derivation in Appendix A.

Typing our earlier example with these rules, the environment of the closure f contains a reference, so its environment type is forced to be of sort $\mathbf{Type}_1$, while *id*'s environment type is $\mathbf{Type}_0$. This approach is similar to the step-indexing model developed by Ahmed [2], but makes type levels explicit in the type system, rather than in the meta-theory.

We have yet to prove any of these proposed languages terminating, and are still open to exploring other designs as well. But the intuition is that the semantics of stores will be inductive on the universe level, with each level able to contain only functions that close over the previous level of stores. At level 0 the semantics is equivalent to a ground store, and so the store must contain only terminating functions. Our eventual goal is to extend this language design not only to a simply typed language, but also to a dependently typed language like the Calculus of Constructions (CC). This could serve as an IL for a type-preserving compiler from CC to a low-level language like C, enabling a compiler with a type-check-then-link approach [4].

REFERENCES

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L3: A Linear Language with Locations. *Fundam. Inf.* 77, 4 (2007). https://doi.org/10.1007/11417170_22

[2] Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State.* Ph. D. Dissertation. http://www.ccs.neu.edu/home/amal/ahmedthesis.pdf

[3] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Symposium on Logic in Computer Science (LICS)*. https://doi.org/10.1145/3209108.3209189

[4] William J. Bowman. 2018. *Compiling with dependent types.* Ph. D. Dissertation. https://doi.org/10.17760/d20316239

[5] Ohad Kammar, Paul B. Levy, Sean K. Moss, and Sam Staton. 2017. A Monad for Full Ground Reference Cells. In *Symposium on Logic in Computer Science (LICS)*. https://doi.org/10.48550/arXiv.1702.04908

[6] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2676726.2676969

[7] P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* (1964). https://doi.org/10.1093/comjnl/6.4.308

[8] Paul Blain Levy. 2002. Possible World Semantics for General Storage in Call-By-Value. In *International Workshop on Computer Science Logic (CSL)*. https://doi.org/10.1007/3-540-45793-3_16

[9] Conor McBride. 2016. I Got Plenty o' Nuttin'. https://personal.cis.strath.ac.uk/conor.mcbride/PlentyO-CR.pdf [Online; accessed 24-May-2023].

[10] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed closure conversion. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/237721.237791

[11] Andrzej S. Murawski and Nikos Tzevelekos. 2018. Algorithmic Games for Full Ground References. *Formal Methods in Systems Design* (2018). https://doi.org/10.1007/s10703-017-0292-9

A APPENDIX

Below we have the tree for deriving that *id* has type $\exists\ \alpha : \mathbf{Type}_0.(\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha$, using our proposed typing rules.

$$\dfrac{\Gamma \vdash \langle\rangle :: \mathbf{Type}_0 \qquad \dfrac{\Gamma \vdash (\boldsymbol{\lambda} x.\boldsymbol{\lambda} env.x) : \mathbf{Nat} \to \langle\rangle \to \mathbf{Nat} :: \mathbf{Type}_0 \qquad \Gamma \vdash \langle\rangle : \langle\rangle :: \mathbf{Type}_0}{\Gamma \vdash \langle(\boldsymbol{\lambda} x.\boldsymbol{\lambda} env.x), \langle\rangle\rangle : (\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha[\langle\rangle/\alpha]}}{\Gamma \vdash \mathbf{pack}\langle\langle\rangle, \langle(\boldsymbol{\lambda} x.\boldsymbol{\lambda} env.x), \langle\rangle\rangle\rangle : \exists\ \alpha : \mathbf{Type}_0.(\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha :: \mathbf{Type}_0}$$

Below we have the tree for deriving that f has type $\exists\ \alpha : \mathbf{Type}_1.(\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha$, using our proposed typing rules. The function body has been abbreviated to e. $\mathbf{Ref}_{\mathbf{type}}$ is $\mathbf{Ref}\ (\exists\ \alpha : \mathbf{Type}_0.(\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha)$, which we expand when we derive its sort, but abbreviate otherwise. We omit the subtree of deriving the type for the function body e, as it is similar to the *id* case and not interesting. The proposed rule for **Ref** bumps the sort level of the environment of f, which in turn bumps the sort level for the pair of code and environment, since the environment contains a reference.

$$\dfrac{\dfrac{\texttt{by derivation } D_1}{\Gamma \vdash \langle\mathbf{Ref}_{\mathbf{type}}\rangle :: \mathbf{Type}_1} \qquad \dfrac{\Gamma \vdash (\boldsymbol{\lambda} x.\boldsymbol{\lambda} env : \langle\mathbf{Ref}_{\mathbf{type}}\rangle.e) : \mathbf{Nat} \to \langle\mathbf{Ref}_{\mathbf{type}}\rangle \to \mathbf{Nat} :: \mathbf{Type}_1 \qquad \dfrac{\texttt{by derivation } D_1}{\Gamma \vdash \langle\mathbf{Ref}_{\mathbf{type}}\rangle :: \mathbf{Type}_1}}{\Gamma \vdash \langle(\boldsymbol{\lambda} x.\boldsymbol{\lambda} env : \langle\mathbf{Ref}_{\mathbf{type}}\rangle.e), \langle r\rangle\rangle : (\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha[\langle\mathbf{Ref}_{\mathbf{type}}\rangle/\alpha] :: \mathbf{Type}_1}}{\Gamma \vdash \mathbf{pack}\langle\langle\mathbf{Ref}_{\mathbf{type}}\rangle, \langle(\boldsymbol{\lambda} x.\boldsymbol{\lambda} env : \langle\mathbf{Ref}_{\mathbf{type}}\rangle.e), \langle r\rangle\rangle\rangle : \exists\ \alpha : \mathbf{Type}_1.\langle(\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha\rangle :: \mathbf{Type}_1}$$

where D_1 is

$$\dfrac{\text{Ref}\ \dfrac{\dfrac{\Gamma \vdash \mathbf{Type}_0 :: \mathbf{Type}_1 \qquad \Gamma, \alpha :: \boxed{\mathbf{Type}_0} \vdash (\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha :: \boxed{\mathbf{Type}_0}}{\Gamma \vdash \exists\ \alpha : \mathbf{Type}_0.(\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha :: \boxed{\mathbf{Type}_0}}}{\Gamma \vdash \mathbf{Ref}\ (\exists\ \alpha : \mathbf{Type}_0.(\mathbf{Nat} \to \alpha \to \mathbf{Nat}) \times \alpha) :: \boxed{\mathbf{Type}_1}}}{\Gamma \vdash \langle\mathbf{Ref}_{\mathbf{type}}\rangle :: \mathbf{Type}_1}$$

Because of the differing sort annotations, the update to the reference r cannot be well typed. This prevents the non-termination caused by backpatching.

More generally, the derivation of D_1 shows how the type of the **Ref** must be one greater than the environment of the closure it contains, so the **Ref** cannot appear in the environment of a closure it contains.