Solving the Funarg Problem with Static Types

Caleb Helbling
caleb@purdue.edu
Purdue University
West Lafayette, Indiana, USA

Fırat Aksoy
firat.aksoy@ogr.iu.edu.tr
Istanbul University
Istanbul, Turkey

ABSTRACT

The difficulty associated with storing closures in a stack-based environment is known as the funarg problem. The funarg problem was first identified with the development of Lisp in the 1970s and hasn't received much attention since then. The modern solution taken by most languages is to allocate closures on the heap, or to apply static analysis to determine when closures can be stack allocated. This is not a problem for most computing systems as there is an abundance of memory. However, embedded systems often have limited memory resources where heap allocation may cause memory fragmentation. We present a simple extension to the prenex fragment of System F that allows closures to be stack-allocated. We demonstrate a concrete implementation of this system in the Juniper functional reactive programming language, which is designed to run on extremely resource limited Arduino devices. We also discuss other solutions present in other programming languages that solve the funarg problem but haven't been formally discussed in the literature.

CCS CONCEPTS

• **Software and its engineering → Procedures, functions and subroutines**; *Functional languages.*

KEYWORDS

funarg, closure, stack, heap, memory, functional, embedded

ACM Reference Format:
Caleb Helbling and Fırat Aksoy. 2021. Solving the Funarg Problem with Static Types. In *IFL '21: ACM Symposium on Implementation and Application of Functional Languages, September 01–03, 2021, Online.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The funarg problem refers to the difficulty associated with compilation of first-class functions in programming languages for stack-based environments. The problem arises in the bodies of nested functions where the nested function refers to identifiers defined in the parent function's lexical scope which are not available in it's own lexical scope. The standard solution is to allocate closures on the heap. There are two variants of this problem: upwards and downwards. The upwards funarg problem refers to the management of the stack when a function returns another function (that may potentially have a non-empty closure), whereas the downwards funarg problem refers to the management of the stack when a function is passed to another function (that may also have a non-empty closure).

Solving the upwards funarg problem is considered to be much more difficult than the downwards funarg problem. The downwards funarg problem is easily solved by copying the closure's lexical scope (captured variables) to the top of the program's stack. For the upwards funarg problem, an issue arises with deallocation. A returning function may capture local variables, making the size of the closure dynamic. Due to this behavior, the closure itself cannot be copied down the stack without causing stack corruption. We must keep in mind that, as soon as the function returns, any local variables will be lost when they are popped off the stack.

In 1968, Weizenbaum [12] made an analysis of function closures using lambda calculus, and showed that function closures in lambda calculus cannot have a stack based data structure and must in fact consist of a tree. Moses [9] further analyzed the problem for the Algol programming language. Early attempts to solve the funarg problem [10] appear to be hampered by a number of factors, including dynamic scoping, lack of static types and mutable variables.

Despite the titles of these early papers, we contend that the funarg problem (in particular, the upwards funarg problem) has not been adequately solved. In our approach, we solve the upwards funarg problem by statically determining a closure's size at compile time. This allows the closure from the upwards funarg problem to be copied into the calling function, as if it were any other statically sized piece of data. Since the type of a closure's lexical scope can be viewed as a structural record type, we can make our language polymorphic over closures and their lexical scopes. This allows higher order functions such as map (downwards funarg problem) and compose (upwards and downwards funarg problem) to be written very concisely and naturally.

Analyzing closures from a type theoretic perspective is not a new exercise [1, 6–8]. These papers prefer to treat closures as explicit environments, whose type is determined by an existential. This is not helpful to solving the funarg problem, since existentials are typically treated as boxed values and must be allocated on the heap. Therefore our contribution is unique in the sense that we seek to solve the funarg problem rather than analyzing closures from a purely type theoretic environmental perspective.

1.1 Contributions

We describe the following contributions:

- We demonstrate how the prenex fragment of System F can be extended to enable stack allocated closures.
- We discuss a concrete implementation of this approach in the Juniper programming language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IFL '21, September 01–03, 2021, Online
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Expressions
$e := x \mid \lambda x : A.e \mid e_1\, e_2 \mid \Lambda\alpha.e \mid e\, A$

Type Schemes
$\sigma := A \mid \forall a.\sigma$

Types
$A, B, C := \alpha \mid A - C \to B \mid \delta$

Lexical Scope Types
$\delta := \{x_0 : A_0, ..., x_n : A_n\}$

Values
$v := (\lambda x.e)\ [\Delta] \mid \Lambda.e$

Contexts
$\Gamma ::= \emptyset \mid \Gamma, \alpha \mid \Gamma, x : A$

Environments
$\Delta := \{x_0 \mapsto v_0, ..., x_n \mapsto v_n\}$

Table 1: Grammar

$$\frac{v = \Delta(x)}{\Delta \vdash x \Downarrow v}\ [\text{EVAL-VAR}]$$

$$\frac{}{\Delta \vdash \lambda x : A.e \Downarrow (\lambda x.e)[\{y \mapsto \Delta(y) \mid y \in \mathbf{fv}(\lambda x : A.e)\}]}\ [\text{EVAL-ABS}]$$

$$\frac{\Delta_1 \vdash e_1 \Downarrow (\lambda x.e_3)[\Delta_2] \quad \Delta_1 \vdash e_2 \Downarrow v' \quad \Delta_2, x \mapsto v' \vdash e_3 \Downarrow v}{\Delta_1 \vdash e_1\, e_2 \Downarrow v}\ [\text{EVAL-APP}]$$

$$\frac{}{\Delta \vdash \Lambda\alpha.e \Downarrow \Lambda.e}\ [\text{EVAL-TABS}]$$

$$\frac{\Delta \vdash e_1 \Downarrow \Lambda.e_2 \qquad \Delta \vdash e_2 \Downarrow v}{\Delta \vdash e_1\, A \Downarrow v}\ [\text{EVAL-TAPP}]$$

Figure 1: Big step semantics

- We analyze the limitations of the extension and identify the connection between types and closures.
- We make an analysis of other approaches found in existing programming languages, but haven't been discussed formally in the literature.

2 EXTENDING SYSTEM F

In this section, we restrict and extend System F to enable stack allocated closures. We restrict System F to it's prenex fragment which separates regular System F types into type schemes and types for practical reasons that will be explained later. This is a necessary restriction to prevent the emergence of existential types in the type system which prevents determining the sizes of types at compile-time. The primary addition is that of the closure types, which are defined as function types with their lexical scopes attached to them. The lexical scopes δ are either type variables or records of captured variables and their respective types. These lexical scope types are are considered equivalent if both the field names and the types of those fields are identical. These additions make it possible for closures to be polymorphic and enable stack allocation of the closure since the size of the closure is known statically at compile-time. Table 1 defines the grammar of our language.

2.1 Big-step semantics

Stack allocation of closures require the lambda values and their lexical scopes to be propagated and used in pairs. For this reason, we combine the lambda values with their environments into a single value representation. When a lambda is applied, a flat representation of the lambda's lexical scope is constructed and used as the environment along with the lambda's argument for further evaluation. Figure 1 presents the big-step semantics for our language.

2.2 Typing

Figure 2 presents the typing rules for our language. There are several changes made to the type rules for the prenex fragment of System F. Of primary interest is the T-ABS rule which also constructs the type of the lexical scope. The construction of the lexical scope itself is performed by T-DELTA, which finds all free variables within the lambda, looks up their type in Γ and returns the corresponding lexical scope type. The rule T-TABS is also changed to restrict type abstractions to contain no free variables themselves which simplifies the type system and assists in compilation to a stack based environment.

2.3 Examples

In this section, we will look at two examples that demonstrate that our solution for the funarg problem works for accurately typing two of the most commonly used functions in functional programming: **map** and **compose**. For the map example, we will assume that the type system is enriched with a **list** type constructor. The type of these functions are considerably more noisy than the standard System F types. Fortunately, most of the noise can be elided through type inference.

$$map : \forall\alpha.\forall\beta.\forall\delta.(\alpha{-}\delta \to \beta){-}\{\} \to \alpha\,\mathbf{list}{-}\{f : \alpha{-}\delta \to \beta\} \to \beta\,\mathbf{list}$$

For the compose example we give the definition of **compose** as well as its type in our System F extension.

$$\begin{aligned} compose := {} & \Lambda\alpha.\Lambda\beta.\Lambda\gamma.\Lambda\delta_1.\Lambda\delta_2. \\ & \lambda f : (\beta - \delta_1 \to \gamma).\lambda g : (\alpha - \delta_2 \to \beta).\lambda x : \alpha.f\ (g\, x) \end{aligned}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \quad [\text{T-VAR}]$$

$$\frac{\Gamma, x : A \vdash e : B \qquad \Gamma \vdash \lambda x : A.e \rightsquigarrow \delta}{\Gamma \vdash \lambda x : A.e : A - \delta \to B} \quad [\text{T-ABS}]$$

$$\frac{\Gamma \vdash e_1 : A - \delta \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B} \quad [\text{T-APP}]$$

$$\frac{\Gamma, \alpha \vdash e : \sigma \qquad \mathbf{fv}(e) = \emptyset}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\sigma} \quad [\text{T-TABS}]$$

$$\frac{\Gamma \vdash e : \forall\alpha.\sigma}{\Gamma \vdash e\, A : \sigma[\alpha := A]} \quad [\text{T-TAPP}]$$

$$\frac{y_i \in \mathbf{fv}(\lambda x : A.e) \qquad \Gamma \vdash y_0 : C_0, ..., \Gamma \vdash y_n : C_n}{\Gamma \vdash \lambda x : A.e \rightsquigarrow \{y_0 : C_0, ..., y_n : C_n\}} \quad [\text{T-DELTA}]$$

Figure 2: Typing rules.

$$\begin{aligned} &\mathit{compose} : \forall\alpha.\forall\beta.\forall\gamma.\forall\delta_1.\forall\delta_2. \\ &(\beta - \delta_1 \to \gamma) - \{\} \to (\alpha - \delta_2 \to \beta) - \{f : \beta - \delta_1 \to \gamma\} \to \alpha \\ &\qquad - \{f : \beta - \delta_1 \to \gamma, g : \alpha - \delta_2 \to \beta\} \to \gamma \end{aligned}$$

3 LIMITATIONS AND EXTENSIONS

The most obvious limitation of the extension presented in this paper is that all branches in a program that return a closure must have the exact same lexical scope. One would think that this would be a problem since in most languages, branches of conditionals may return lambdas with different lexical scopes, as long as the types are equivalent. A close analysis of the **cond** function in System F when combined with Church-encoded Boolean values **true** and **false** reveals a surprising result, this problem simply does not appear in the extension described in this paper.

To further elucidate our point, we define **cond** operator as a function which takes in a true branch and a false branch and a Church-encoded Boolean value (as described in equations 3, 4, 5, and 6) and returns the respective branch in relation to it's Boolean input.

$$\begin{aligned} &\text{cond} := \Lambda\alpha.\Lambda\beta.\Lambda\gamma.\Lambda\delta \\ &\qquad \lambda t : \alpha.\lambda f : \beta.\lambda c : (\alpha - \{\} \to \beta - \delta \to \gamma).c\, t\, f \end{aligned} \tag{1}$$

$$\begin{aligned} &\text{cond} : \forall\alpha.\forall\beta.\forall\gamma.\forall\delta. \\ &\alpha - \{\} \to \beta - \{t : \alpha\} \to (\alpha - \{\} \to \beta - \delta \to \gamma) - \{t : \alpha, f : \beta\} \to \gamma \end{aligned} \tag{2}$$

$$\text{true} := \Lambda\alpha.\Lambda\beta.\lambda t : \alpha.\lambda f : \beta.t \tag{3}$$

$$\text{true} : \forall\alpha.\forall\beta.\alpha - \{\} \to \beta - \{t : \alpha\} \to \alpha \tag{4}$$

$$\text{false} := \Lambda\alpha.\Lambda\beta.\lambda t : \alpha.\lambda f : \beta.f \tag{5}$$

$$\text{false} : \forall\alpha.\forall\beta.\alpha - \{\} \to \beta - \{\} \to \beta \tag{6}$$

Applying the **cond** operator with **true** and **false** in equations 7 and 8 reveals why Church-encoding does not suffer from the branching problem. In System F, the branch chosen by **cond** is already present in the type system by selecting the type for γ. Fundamentally, this occurs because **true** and **false** have different type signatures, that also determine the branch taken.

$$((\text{cond}\, \alpha\, \beta\, \alpha\, \{t : a\})\, t\, f\, (\text{true}\, \alpha\, \beta)) : \alpha \tag{7}$$

$$((\text{cond}\, \alpha\, \beta\, \beta\, \{\})\, t\, f\, (\text{false}\, \alpha\, \beta)) : \beta \tag{8}$$

However, in a language with sum types and pattern matching (as Boolean types are in most languages), the branching problem with closures will be present. Whether or not this is a major issue is debatable, however, we can say for certain that this removes some generality from our approach. An interesting approach to recovering this generality is a paper on open closures by Scherer and Hoffmann [11], however their approach only applies to simply typed lambda calculus and not for System F.

The second limitation appears under the presence of mutually recursive functions. When compiling top level functions, we can consider their closure to be empty since we can refer to them using a simple function pointer. For inner functions that are often mutually recursive, the situation is more complicated. The lexical scope of all functions declared in a **let-rec** block can be constructed by taking the union of all of the lexical scopes of all the functions declared in the block. This constructs a common lexical scope that can be passed to all the related functions. Therefore, all functions declared in the **let-rec** block will have identical lexical scope types.

When one function in the let-rec block calls another function in the same let-rec block, it simply passes the lexical scope that it was passed. Within the lexical scope type itself, we place all the functions within the let-rec block with the non-closure function type $A \to B$. This indicates that this function should be passed the lexical scope that this function is contained within.

Extending the system presented in this paper to the impredicative System F at first glance seems feasible. We believe that the funarg problem is orthogonal issue to that of first-class polymorphism and stack allocated existential types. However we have yet to make a full analysis of how the lexical environments presented here interact with System F. One possible issue that may crop up is subtle interactions with existential types, which can be encoded in System F [2].

4 IMPLEMENTATION

We have implemented the closure system given here in Juniper [3], a functional reactive programming language for the Arduino. The Juniper language transpiles all of its code to C++. The analysis of C++ in section 5 reveals the issues facing the use of lambdas as a transpile target. To work around the issues presented in that section,

we implement our own **function** class. This class is constructed by passing in a lexical scope, along with a function pointer. In C++, lambda types that do not capture anything can be casted to type safe function pointers. We provide our own stack allocated closure support by passing in the lexical scope as the first argument to the underlying lambda function. Within this lambda, the compiler inserts variable assignments which extracts the contents of the lexical scope into local variables. Lambdas that do not capture anything are not passed a lexical scope.

```
template <typename ClosureType, typename Result, typename
        ...Args>
class function;

template <typename Result, typename ...Args>
class function<void, Result(Args...)> {
private:
    Result(*F)(Args...);

public:
    function(Result(*f)(Args...)) : F(f) {}

    Result operator()(Args... args) {
        return F(args...);
    }
};

template <typename ClosureType, typename Result, typename
        ...Args>
class function<ClosureType, Result(Args...)> {
private:
    ClosureType Closure;
    Result(*F)(ClosureType&, Args...);

public:
    function(ClosureType closure, Result(*f)(ClosureType
        &, Args...)) : Closure(closure), F(f) {}

    Result operator()(Args... args) {
        return F(Closure, args...);
    }
};
```

A lexical scope struct is defined for every combination of environment names discovered in the program. The types of the variables are made generic, so that these lexical scope structs can be declared before any of the other types in the program. Below is an example of the lexical scope struct generated by the Juniper compiler for the **compose** function. These structs are created and passed to the **function** constructor the moment the lambda declared. This ensures that the closure contains a snapshot of the current environment. Importantly, if a mutation of any local variables occurs after a closure has been constructed, the closure will not be updated (since it essentially takes a snapshot of the local environment the moment it is created). For purely functional languages this is not an issue since variables cannot be mutated.

```
// Compose lexical scope
template<typename T1,typename T2>
struct closuret_0 {
    T1 f;
    T2 g;

    closuret_0(T1 init_f, T2 init_g) :
        f(init_f), g(init_g) {}
};
```

5 EXISTING APPROACHES

In this section we will analyze a number of different existing languages and how they solve the funarg problem. To our knowledge the system presented in this paper is the only language that will allow the type of a closure to be fully written out with text (ie, it does not generate an implicit type or class).

In C++, each declaration of a lambda implicitly declares a new class [4] and therefore type in the system. Since these classes are auto-generated, it is impossible to write their types using text, and programmers must rely on the auto keyword. These C++ lambdas are indeed stack allocated since their underlying representation is a class with a operator() defined. The problems begin however as soon as you start passing around or returning lambdas. Higher order functions that consume lambdas must be made generic over all of these implicit lambda classes. This makes it is impossible to add any sort of constraints on the types of the arguments of the lambda, or the return type of the lambda. Returning a lambda also becomes very annoying, and the auto keyword must be used as the return type. Once again, we see that it is impossible to constrain the return type to anything in particular. The solution used by most C++ programmers is to make use of the standard library std::function, which turns the stack allocated closure into a heap allocated entity. Perhaps in the future, clever use of C++ concepts could add extra type safety to the use of these implicit lambda classes.

To our knowledge, the only other language that has similar capabilities to what is presented in this paper is Rust [5]. Just as in C++, the Rust system operates by generating a unique type for every closure. However Rust goes a step further with its trait system. Like in C++, accepting a closure as a parameter to another function requires the use of a single generic parameter. Unlike C++, the type of this parameter can be constrained with traits. Additionally, returning a closure requires the textual use of one of these traits.

6 FUTURE WORK

In the future we would like to further extend our type system for enriched languages. We are interested in how adding conditionals and sum types to the language will force changes in the handling of the lexical scopes. We also would like to give a more thorough description of how to handle the let-rec case.

We would also like to extend this system for the impredicative System F without restrictions as much as possible, which we predict to potentially force additional changes in the representation of the lexical scopes.

7 CONCLUSION

In this paper we have described an extension to the prenex fragment of System F that solves the funarg problem. Our solution solves both the downwards and the upwards funarg problem and allows closures to be completely stack allocated. Functional languages can take advantage of this new system to optimize memory usage, and this extension is particularly useful on embedded systems. We also discuss extensions to our language and identify the possibility to extend this work to solve the conditional problem as well as stack allocation for first-class polymorphism. We discussed how existing languages solve the funarg problem, and conclude that

our approach is the first so far that allows writing closure type signatures without auto-generated types or classes.

REFERENCES

[1] William J Bowman and Amal Ahmed. 2018. Typed closure conversion for the Calculus of Constructions. (Aug. 2018). arXiv:1808.04006 [cs.PL]

[2] Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Vol. 7. Cambridge university press Cambridge.

[3] Caleb Helbling and Samuel Z Guyer. 2016. Juniper: a functional reactive programming language for the Arduino. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. 8–16.

[4] ISO. 2017. *ISO/IEC 14882:2017 Information technology — Programming languages — C++* (fifth ed.). 1605 pages. https://www.iso.org/standard/68564.html

[5] Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

[6] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96* (St. Petersburg Beach, Florida, United States). ACM Press, New York, New York, USA.

[7] Greg Morrisett and Robert Harper. 1998. Typed closure conversion for recursively-defined functions. *Electron. Notes Theor. Comput. Sci.* 10 (1998), 230–241.

[8] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.* 21, 3 (May 1999), 527–568.

[9] Joel Moses. 1970. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. *SIGSAM Bull.* 15 (July 1970), 13–27.

[10] Erik Sandewall. 1971. A proposed solution to the FUNARG problem. *SIGSAM Bull.* 17 (Jan. 1971), 29–42.

[11] Gabriel Scherer and Jan Hoffmann. 2013. Tracking Data-Flow with Open Closure Types. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer Berlin Heidelberg, 710–726.

[12] Joseph Weizenbaum. 1968. The funarg problem explained. *unpublished memorandum, MIT* (1968).