Effects for Funargs

Jeremy G. Siek, Michael M. Vitousek, and Jonathan D. Turner
University of Colorado at Boulder
jeremy.siek@colorado.edu

Abstract

Stack allocation and first-class functions don't naturally mix together. In this talk we describe a simple type and effect system that helps these features form a nice emulsion. Our interest in this problem comes from our work on the Chapel language, but this problem is also relevant to lambda expressions in C++ and blocks in Objective C. The difficulty in mixing first-class functions and stack allocation is a tension between safety, efficiency, and simplicity. To preserve safety, one must worry about functions outliving the variables they reference: the classic upward funarg problem. There are systems which regain safety but lose programmer-predictable efficiency, and ones that provide both safety and efficiency, but give up simplicity by exposing regions to the programmer. In this talk we present a simple design that combines a type and effect system, for safety, with function-local storage, for control over efficiency.

1. Introduction

This talk describes a design for integrating first-class functions into languages with stack allocation in a way that does not compromise type safety or performance and that strives for simplicity. This design is intended for use in the Chapel programming language [Chamberlain et al. 2011], but could also provide a safer alternative to the new lambda expressions of C++ [ISO 2011] and a more efficient alternative to the blocks of Objective C [Apple Inc. 2011]. The design is meant for performance-oriented languages in which the run-time overhead for each language construct should be relatively small and predictable.

The straightforward integration of first-class functions into a language with stack allocation poses type safety problems because of the classic *upward funarg problem* [Moses 1970], illustrated in Figure 1 in the Chapel language. The `compose` function returns an anonymous function which refers to parameters `f` and `g` of the surrounding `compose` function. Both `f` and `g` are functions of type `func(int,int)`; the first `int` is the input type and the second `int` is the return type. The example then defines the function `inc` and invokes `compose` to obtain `inc2`, a function that increments its argument twice. Because Chapel allocates parameters and local variables on the call stack, the variables `f` and `g` are no longer live when `inc2` is called; the call to `compose` has completed. During the call to `inc2`, the locations previously allocated to `f` and `g` may contain values of types that are different from `func(int,int)`, so we have a counterexample to type safety.


```
proc compose(f: func(int,int),
             g: func(int,int)): func(int,int) {
   return fun(x:int){ var y=f(x); return g(y); };
}
proc inc(x: int): int { return x + 1; }
var inc2 = compose(inc, inc);
inc2(0);
```

Figure 1. Example of an upward funarg in Chapel.

To avoid the upward funarg problem, most languages with first-class functions do not allocate parameters or local variables on the stack; they allocate them on the heap and use garbage collection to reclaim the memory. However, designing garbage collection algorithms that provide predictable performance is an on-going research challenge whereas stack allocation is reliably fast [Miller and Rozas 1994]. Advanced compilers for functional languages employ static analyses to determine which variables may be safely allocated on the stack [Steele 1978, Goldberg and Park 1990, Tofte and Talpin 1994, Serrano and Feeley 1996], which improves efficiency for many programs, but still does not deliver *programmer predictable* efficiency [Tofte et al. 2004].

For the Chapel programming language, we seek a language design in which upward funargs are caught statically, thereby achieving type safety, but that enables higher-order programming and gives the programmer control over run-time costs.

While such a design is not present in the literature, the key ingredients are. Tofte and Talpin [1994, 1997] designed a typed intermediate language, which we refer to as the Region Calculus, with an explicit *region* abstraction: values are allocated into regions and regions are allocated/deallocated in a LIFO fashion. The Region Calculus uses a type and effect system *à la* Talpin and Jouvelot [1992] to guarantee the absence of memory errors, such as the dangling references that occur within upward funargs. Later work relaxed the LIFO restriction on allocation and deallocation [Aiken et al. 1995, Crary et al. 1999, Henglein et al. 2001]. Tofte et al. [2004] suggest that a programming language (as opposed to an intermediate language) with explicit regions would provide both predictable efficiency and type safety. The work by Grossman et al. [2002] on Cyclone provides evidence that this is the case.

While the type and effect systems of Tofte et al. [2004] and Grossman et al. [2002] support many higher-order programming idioms, they still disallow many useful cases that involve curried functions, such as the above `compose` function. The lambda expressions of C++ [ISO 2011] and the blocks of Objective C [Apple Inc. 2011] offer a simple solution: enable the copying of values into extra storage associated with a function. For example, in the `compose` function, the programmer could elect to copy the values of `f` and `g` into storage associated with the anonymous function, thereby keeping them alive for the lifetime of the function. One might be tempted to make this copying behavior the only seman-

tics, but in many situations the copy is too expensive [Järvi et al. 2007]. (Suppose the copied object is an array.) In the C++ design, the programmer may choose to either make the copy, incurring run-time cost in exchange for safety, or capture the reference, incurring no extra run-time cost but exposing themselves to the potential for dangling references.

We take away the following points from this prior research:

1. The Region Calculus demonstrates that a type and effect system can support many higher-order programming idioms while disallowing upward funargs.
2. Cyclone shows that only a small amount of annotations are needed to support a type and effect system.
3. The C++ and Objective C approach of providing function-local storage enables the full spectrum of higher-order programming while keeping the programmer in control of run-time costs.

In this talk we present a language design that uses a type and effect system to detect and disallow upward funargs with dangling references and that also offers the ability to copy values into function-local storage. However, unlike the Region Calculus and Cyclone, we do not expose the region abstraction to the programmer; doing so would unnecessarily complicate the language from the programmer's viewpoint. In our system, the effect of an expression is a *set of variables* instead of a set of region names. The variables in the effect are those that may be read by the expression.

In the talk we will present a static and dynamic semantics directly for our system and sketch a syntactic proof of type safety. The goal here is to provide both an aid to implementation and a direct understanding of why our type and effect system ensures type safety, and therefore also memory safety [Pierce 2002]. As an added benefit, the direct semantics supports efficient tail calls through a simple restriction in the type system that enables early deallocation. For those familiar with regions, we also present a type-preserving translation from well-typed terms of our system into the Region Calculus.

We look forward to discussing this design with the workshop participants and look forward to obtaining feedback regarding the direction for integrating higher-order functions and stack allocation. A 30 minute time slot for this talk is fine.

A 10-page paper describing our design is on arxiv:
`http://arxiv.org/abs/1201.0023`

References

A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 174–185, New York, NY, USA, 1995. ACM. doi: http://doi.acm.org/10.1145/207110.207137.

Apple Inc. Blocks programming topics. Technical report, Apple Inc., Cupertino, CA, March 2011.

B. Chamberlain, S. Deitz, S. Hoffswell, J. Plevyak, H. Zima, and R. Diaconescu. *Chapel Specification*. Cray Inc, 0.82 edition, October 2011.

K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 262–275, New York, NY, USA, 1999. ACM. doi: http://doi.acm.org/10.1145/292540.292564.

B. Goldberg and Y. G. Park. Higher order escape analysis. In *ESOP*, 1990.

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293, New York, NY, USA, 2002. ACM Press.

F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '01, pages 175–186, New York, NY, USA, 2001. ACM. doi: http://doi.acm.org/10.1145/773184.773203.

ISO. Working draft, standard for programming language C++. Technical Report N3242, ISO, February 2011.

J. Järvi, J. Freeman, and L. Crowl. Lambda expressions and closures for c++ (revision 1). Technical Report N2329, ISO/IEC JTC 1 SC22 WG21, June 2007.

J. S. Miller and G. J. Rozas. Garbage collection is fast, but a stack is faster. AI Memos AIM-1462, MIT Artificial Intelligence Lab, March 1994.

J. Moses. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. *SIGSAM Bull.*, pages 13–27, July 1970. doi: http://doi.acm.org/10.1145/1093410.1093411.

B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

M. Serrano and M. Feeley. Storage use analysis and its applications. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 50–61, New York, NY, USA, 1996. ACM. doi: http://doi.acm.org/10.1145/232627.232635.

G. L. Steele. Rabbit: A compiler for Scheme. Technical report, Cambridge, MA, USA, 1978.

J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 162 –173, jun 1992. doi: 10.1109/LICS.1992.185530.

M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 188–201. ACM Press, 1994.

M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation Journal*, 17:245–265, 2004.