Writing Systems Software in a Functional Language

An Experience Report

Iavor S. Diatchki
diatchki@galois.com

Thomas Hallgren
hallgren@cse.ogi.edu

Mark P. Jones
mpj@cs.pdx.edu

Rebekah Leslie
rebekah@cs.pdx.edu

Andrew Tolmach
apt@cs.pdx.edu

ABSTRACT

Current practices for developing systems software usually rely on fairly low-level programming languages and tools. As an alternative, our group has been investigating the possibility of using a high-level functional language, Haskell, for kernel and device driver construction, with the hope that it will allow us to produce more reliable and secure software. In this paper, we describe our experience developing a prototype operating system, House, in which the kernel, device drivers, and even a simple GUI, are all written in Haskell. The House system demonstrates that it is indeed possible to construct systems software in a functional language. However, it also suggests some ideas for a new Haskell dialect with features that target specific needs in this domain, including strongly typed support for low-level data structures and facilities for explicit memory accounting.

1. INTRODUCTION

Development of systems software—including device drivers, hypervisors, and operating systems—is particularly challenging when high levels of assurance about program behavior are required. On the one hand, programmers must deal with intricate low-level and performance-critical details of hardware such as fixed-width registers, bit-level data formats, direct memory access, I/O ports, data and instruction caches, and concurrency. On the other hand, to ensure correct behavior, including critical safety and security properties, the same code must also be related directly and precisely to high-level, abstract models that can be subjected to rigorous analysis, possibly including theorem proving and model checking. Failure of computer software can be a major problem in many domains, but the consequences of failure in systems software are especially severe: Even simple errors or oversights—whether in handling low-level hardware correctly or in meeting the goals of high-level verification—can quickly compromise an entire system.

The use of low-level programming languages and tools enables programmers to address performance concerns in systems software, but also makes it much harder to reason formally about the code, and hence much harder to obtain high confidence in the behavior of the resulting software. However, new options are emerging that could allow the development of robust, reliable, and secure software using programming languages that provide a higher level of abstraction than has traditionally been possible in this domain.

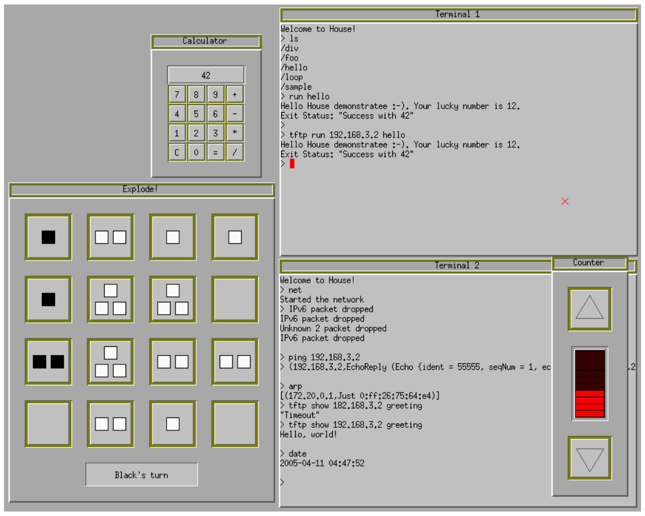
In particular, over the past few years, our group has been investigating the possibility of using the functional programming language Haskell [20] to implement systems-level software. Of course, this offers the advantages that are usually associated with moving to a higher-level language such as increased programmer productivity and software reuse. However, our primary motivation for using Haskell is the hope that it will allow us to produce more secure and more reliable software systems. For example, the type and memory safety properties of Haskell can prevent several nasty kinds of program bugs. In addition, the adoption of a pure functional language—with strong, mathematically defined foundations—enables us to support formal verification and validation of software systems using connections to theorem provers, proof assistants, and model checkers.

This paper summarizes our experience building House (Section 2), a prototype operating system implemented in Haskell, including a discussion of some of the obstacles that we ran into during the process (Section 3). Prompted by these observations, we are developing a new Haskell dialect that we hope will address these problems while also preserving the most important features and benefits of standard Haskell programming. To illustrate this, we summarize (in Section 4) two language extensions that allow bit-level data structures (referred to collectively as *bitdata*) and memory-based tables to be accessed and manipulated in a flexible and efficient manner, with fine control over representation and without sacrificing attractive features of Haskell such as pattern matching and strong typing. We conclude (in Section 5) by describing additional challenges that we are tackling in our current work to better support systems software development in a functional language, including mechanisms for more modular run-time systems, and for more explicit memory accounting.

2. THE HOUSE OPERATING SYSTEM

We are by no means the first to use functional languages to develop systems software—earlier examples include the Fox [10], Ensemble [18], Timber [14], and Hello [7] projects—but, even so, Haskell is not usually thought of as a systems programming language. Nevertheless, the development of

House [9] has shown that it is possible to implement systems software in Haskell without shying away from important low-level concerns. More specifically, House[1] is a prototype operating system that boots and runs on bare metal (IA32) and in which the kernel, a small collection of device drivers (including video, keyboard, mouse, and networking), a simple graphical user interface, and some basic applications, have all been implemented in Haskell (See Figure 1). Among other features, the House kernel supports protected execution of arbitrary user-level binaries and manages the virtual memory of each such process by direct manipulation of the page table data structures that are used by the hardware MMU. Architecturally, House is implemented using standard Haskell code running on top of a hardware abstraction layer that we refer to as the H-interface. In fact the H-interface is also implemented primarily in Haskell, together with a small collection of primitives implemented in C and/or assembler. Safety was a key design goal of the H-interface: with the exception of certain I/O operations, none of the operations that it provides can corrupt the Haskell heap. Another design goal was to facilitate modular validation and verification efforts by capturing the semantics of the H-interface in a collection of formal properties. These properties are intended to be used both to establish security properties of a kernel running on top of the interface, and, independently, to validate its implementation in the mapping to low-level hardware.



```
Welcome to House!
> ls
/div
/foo
/hello
/loop
/sample
> run hello
Hello House demonstratee :-). Your lucky number is 12.
Exit Status: "Success with 42"
>
> tftp run 192.168.3.2 hello
Hello House demonstratee :-). Your lucky number is 12.
Exit Status: "Success with 42"
>
```

```
Welcome to House!
> net
Started the network
> IPv6 packet dropped
IPv6 packet dropped
Unknown 2 packet dropped
IPv6 packet dropped

> ping 192.168.3.2
> (192.168.3.2,EchoReply (Echo {ident = 55555, seqNum = 1, ec

> arp
[(172.20.0.1,Just 0:ff:26:75:64:e4)]
> tftp show 182.168.3.2 greeting
"Timeout"
> tftp show 192.168.3.2 greeting
Hello, world!

> date
2005-04-11 04:47:52

>
```

Figure 1: A screen shot of House running in graphical mode and showing a simple calculator, a game, and a couple of open terminal windows, one of which is being used to run some simple networking demos, all of which are written in Haskell.

3. ISSUES RAISED

Our experience using Haskell to build House, as well as an ongoing effort to build a second kernel based on the L4 specification [17], have shown that it is possible to implement a kernel in a functional language. However, it has also exposed several critical issues and problems, which we discuss in this section.

Low-level Operations.

One of the most common questions that is raised when we talk about House is how we obtain access to low-level features such as registers, I/O ports, and memory-based tables. In fact, this was fairly easy to address using the Haskell foreign function interface (FFI); some of the more general-purpose functions that were needed (for example, to peek or poke into memory) were already provided by the FFI, while other, more specialized operations were implemented quite easily by packaging low-level functions coded in C or assembly language as new Haskell primitives [2]. Some of these functions, however, are potentially unsafe and require careful and disciplined use to ensure that the normal type and memory safety guarantees of Haskell are not compromised. For example, it is possible to construct an arbitrary machine address as an argument for the function that pokes a byte value to memory. We designed the H-interface so that these functions would not be needed by client operating system kernels, and so that their uses could be restricted to the implementation of the interface. Nevertheless, this did not prevent us from making some programming mistakes. For example, an error in formulating the code to zero a 4K page of bytes in an early version of House resulted in a loop that actually wrote 4097 bytes, overflowing the page by a single byte. This error was detected during a code review, but otherwise would probably have resulted in highly unpredictable behavior, and would have been very difficult to debug. However, if the type system had enforced the restriction of the loop index to 12 bits, or if array bounds checking had been supported (viewing a page as an array of 2^{12} bytes), then this mistake would have been more easily detected or prevented, possibly even at compile-time.

Performance.

The other common issue that is raised in discussions about House is the question of performance. There are long-standing concerns in the systems community that the use of any high-level language prevents the attention to low-level performance details (such as sensitivity to cache issues) that is needed in a real-world system. However, the House prototype has not yet been subjected to rigorous performance analysis (or tuning). We do not know, for example, if the House kernel can be tuned or enhanced to provide reasonable throughput with the workload of a conventional desktop or server, or if its network interface can keep pace with line speed on a standard Ethernet connection. Although it may be possible to achieve these kinds of performance goals—especially, in the long term, if we can rely on aggressive, whole program optimization—it is unlikely that they will be met with current Haskell systems. We have recently begun some preliminary experiments to evaluate specific aspects of the performance of House, and we know even from code inspection that there will be several opportunities to increase performance. Further investigation of this important topic, however, remains as future work.

Run-time System Issues.

Our work on House has also exposed some deeper and, in some cases, more subtle problems that require careful

[1]House was developed within our group, primarily by Thomas Hallgren and Andrew Tolmach, and expanding on previous work on the hOp system [1] by Sébastian Carlier and Jérémy Bobbio.

attention. The first of these has to do with the run-time system that is used to provide services for memory allocation, garbage collection, and concurrency in Haskell programs. It follows, in particular, that any argument we make about the security, reliability, or correctness of a Haskell program requires a corresponding argument about the run-time system. However, in the House prototype, the run-time system—derived fairly directly from the run-time system of GHC, a general purpose Haskell compiler—is a large and complex software component in its own right (around 50K lines of code, depending on how it is measured), most of which is written in C. Such factors make it difficult, at best, to provide a compelling argument for correctness of a small, systems software component. Fortunately, experience with other functional language implementations suggests that it will be possible to address this problem by developing a dramatically smaller and more focused run-time system. A simple but complete garbage collector for a functional language can be implemented in just 100–300 lines of code, and a number of such implementations have been (or are currently being) formally verified by ourselves and others [19]. There are fielded production-quality collectors offering good performance (e.g., by using generations) and supporting a rich set of language features (e.g., many kinds of memory blocks, weak pointers, etc.) in under 2000 lines.

Another run-time system issue is the question of which services should be provided and what the associated semantics should be. Dynamic memory allocation, for example, is a very important facility that we might expect to be provided by the run-time system. In particular, providing a general framework for memory allocation would avoid the wasteful and error-prone approach of writing (and debugging) new allocators from scratch as part of each new application. However, it is difficult to build a framework that is sufficiently generic to handle the wide range of memory allocation techniques that are in use or to accommodate the vagaries of specific hardware platforms. For example, the structure for an IA32 page table does not suggest an obvious way to include any extra 'tag' data that could be used for run-time type identification, garbage collection, etc. Our House implementation addresses this by using multiple allocators, including a garbage-collected heap that is implemented by the Haskell run-time system as well as an independently garbage collected pool of machine pages that is implemented as part of the H-interface.

There is also some duplication of functionality to support concurrency in our House prototype. The run-time system includes facilities for running and switching between multiple Haskell threads; this is used, for example, to support some aspects of interrupt handling. A second form of concurrency is implemented by the H-interface to facilitate programmed context switching between multiple threads of user-level code. Unfortunately, we cannot use Haskell threads for the latter in a kernel that requires, for example, priority-based scheduling because the (current) run-time system does not provide this kind of control over scheduling.

In summary, it is unlikely that we could use a single run-time system facility for either memory allocation or concurrency that would be sufficiently generic, while also being simple enough to enable a manageable, verifiable implementation. In the long term, a better approach is to develop a modular run-time system that can be configured to suit a particular application by including only those (possibly customized) components that are required. This strategy would also help to reduce the overall size of the run-time system.

Resource Management.

High-level languages are designed to distance programmers from the details of an underlying machine so that they can focus more directly on coding their particular application. For example, a language like Haskell that supports automatic garbage collection presents the programmer with the illusion of a computer with an infinite memory: the programmer can allocate arbitrary amounts of memory without (in theory) ever worrying about deallocation, and without even having to check to see if there is any memory available before asking for more. Instead, an automatic garbage collector quietly identifies unreachable blocks of memory on demand so that they can be recycled and reused. Attempts to deallocate a single block of memory either prematurely or else multiple times using `free()` are notorious sources of bugs in C programs that can be completely avoided by using a garbage collector.

But, of course, the illusion is not perfect. Real computers have only a limited amount of memory, and cannot continue to execute properly if the memory becomes full. Moreover, if multiple processes share a single memory, then it is possible for one process to starve the others (or even to halt the entire system) by allocating all of the memory to itself. Unfortunately, Haskell does not provide any way for a program to determine how much free memory is available or, more seriously, to account for fair sharing of memory between multiple threads. Despite all of our efforts to maintain type and memory safety, it is still possible to crash House by running a program that causes the kernel to allocate a large number of internal data structures and thus overflow the Haskell heap.

We believe that new language mechanisms, backed by run-time system support, are needed to enable more explicit control over memory usage, allowing dynamic partitioning and reallocation of memory between threads as well as facilities for detecting and recovering from memory overflow. Issues like these have received significant attention in the systems community—for example, fine-grained memory accounting techniques and per-process user-level memory management are key elements of Singularity [11] and recent L4 proposals [6, 8, 15]—but we are not aware of any previous work to integrate such functionality into a general-purpose functional language design.

4. TOWARDS "SYSTEMS HASKELL"

In the previous section, we described four high-level issues that we encountered in the development of House: (i) the loss of strong typing and safety that results from the use of FFI primitives, which we attribute to weaknesses in the Haskell type system for describing low-level operations; (ii) performance concerns; (iii) the inflexibility of a large, monolithic runtime system; and (iv) resource management in a language that tries to abstract away from explicit control of memory allocation and use. We are working actively to address these problems in the design and implementation of a new dialect of Haskell that we are tentatively referring to as "Systems Haskell". To give a flavor of what this new dialect might look like, this section describes the language features that we have developed to address the first of the four issues listed above. Our presentation here is necessarily brief, but more details can be found elsewhere [3, 4, 5]. A pleasing aspect of this work was the realization that we do not require a fundamentally new type system or language framework: the desired functionality can be provided quite elegantly by building on the existing features of the Haskell type system including kinds, qualified types, and improvement.

4.1 Bitdata

Haskell has excellent support for manipulating tree-like data structures via *algebraic datatypes*. Unfortunately, ordinary algebraic datatypes are not sufficient in situations where the representation of the data is of importance. This is quite common in systems programming and, more generally, it occurs on the boundaries between different systems: either between different software components (e.g., the interface between an OS kernel and a user process), or between software and hardware (e.g., when a driver interacts with a device). In such situations, the data representation is determined by a predefined, external protocol. For this reason, we should not delegate the choice of representation to the compiler but instead, we should provide a mechanism for programmers to specify the representation of data.

Our work on bitdata [4, 5] provides one solution to this problem. We adopt the ideas of algebraic datatypes but we extend them to support programmer specified data layouts at the bit level. Each constructor of a data declaration may be augmented with a declaration that specifies the bit patterns to be used for the values constructed with that constructor. For example, here is a datatype that is useful when configuring a machine based on the IA32 architecture [12]:

```
bitdata SegType

  = DataSeg { expDown :: Bool
            , writable :: Bool
            , accessed :: Bool
            } as B10 # expDown # writable # accessed

  | CodeSeg { conform :: Bool
            , readable :: Bool
            , accessed :: Bool
            } as B11 # conform # readable # accessed

  | TaskSeg { busy :: Bool
            } as B010 # busy # B1
```

The type `SegType` has three constructors, `DataSeg`, `CodeSeg`, and `TaskSeg`, each with a number of fields. For each field we specify a label and a type (the type `Bool` is represented with one bit). The difference from ordinary algebraic datatypes is in the **as** clause that comes after each constructor—it specifies the representation that should be used by the compiler when it generates code. Note that the symbols `B1`, `B10`, and `B010` used here are binary literals (of one, two, and three bits, respectively) that represent fixed bit patterns in the representation of the different segment types. For example, we can define a strongly typed `stack_seg` segment as follows:

```
stack_seg = DataSeg { expDown = True
                    , writable = True
                    , accessed = False
                    }
```

The corresponding (untyped) bit pattern that will be used to represent `stack_seg` is `10110`.

As with ordinary algebraic datatypes, we use pattern matching to examine and deconstruct values—the compiler generates code to check the appropriate 'tag' bits (e.g., `10` in the declaration of `DataSeg`) and to access the bits corresponding to the fields.

Using bitdata declarations in place of more traditional approaches, such as 'bit twiddling', simplifies the job of the systems programmer. It makes programs easier to understand and maintain because it reduces the amount of 'coding' and automates mundane (yet error prone) tasks such as accessing bitfields. In addition, bitdata declarations correspond fairly closely to the specifications used in technical manuals (e.g., the box diagrams used in the specification of L4[17] and IA32[12]) which means that they are easy to write and check for correctness. We have also developed algorithms to analyze bitdata declarations to help programmers detect potential bugs in their specifications (e.g., to detect when there are no 'tag' bits to distinguish values defined with different constructors).

4.2 Memory Areas

Bitdata types are useful when we work with fairly small pieces of data, typically data that fits in a single machine word. For many problems in the systems domain we also need to manipulate larger structures that have an explicit layout in memory. Examples of such structures include page directories and tables, and data structures that contain machine state that has been stored by the hardware (e.g., the thread control blocks that are used in L4 implementations).

Our work on *memory areas* [3, 5] imports and extends the ideas from traditional system programming languages, such as C [16], to the context of modern functional programming languages. We introduce a set of reference and pointer types that identify memory areas, and a collection of types that are used to describe the layout of memory areas. The basic building blocks for describing memory areas are the bitdata types that we briefly introduced in Section 4.1. We can combine descriptions of memory areas into descriptions of larger memory areas by using arrays and structures. As usual, arrays describe a sequence of adjacent identical areas, while structures describe adjacent areas with different layouts.

Our design differs from other languages in several ways. One difference is that the types that we use to describe memory area specify *precisely* the layout of the data in memory. For example, implementations are *not* free to add padding between the elements of a structure—if padding is required, then programmers have to declare it explicitly. Also, we were able to utilize the fairly advanced type system that is available in Haskell-like languages in several different ways. We use different types to describe memory areas that contain big-, little-, or native-endian representations for a bitdata value. By using Haskell's overloading mechanism [21, 13] we were able to present programmers with a unified interface for manipulating these different types, while the compiler uses the types to insert endianness coercions automatically. We also use the type system to track the size of arrays, which enables us to provide safe array access without the need to augment the representation of arrays with size information. Another place where we used the type system was to annotate pointers and references with alignment information, which enables us to specify rather precise types (e.g., we can write a function whose type guarantees that it will be applied only to properly aligned pointers). For example, the following declaration introduces a reference, `pdir`, to an IA32 page directory, which is a table of 1024 page directory entries, and which must be aligned on a 4K (page) boundary.

```
area pdir :: ARef 4K (Array 1024 (Stored PageDirEntry))
```

Note that the size and alignment properties of `pdir` are not just annotations or hints to the compiler, but are actually captured in its type, and enforced by the type checker; we cannot access an array element outside the valid range, and the correct alignment is guaranteed.

5. CONCLUSIONS AND FUTURE WORK

Developing systems software is, at best, a challenging endeavor, so it is important to look for steps that might simplify the task or improve the quality of the results that we obtain. In this paper, we have described our investigation of the role that higher-level languages, specifically Haskell, can play in supporting the construction of a reliable and secure operating system. We have concentrated here on aspects of language design. Independently, members of our group are developing techniques to support the verification of kernel security features, which we hope will be facilitated by our use of a functional language; we will discover the extent to which that goal is realized as work in that area progresses.

Readers familiar with Haskell may be surprised that we have made no mention of its 'lazy evaluation' strategy that delays the execution of each computation until results are demanded. This promotes a flexible and modular programming style, and has been put to good use in House, but it also makes it harder to anticipate the likely cost, in both time and space, of program execution. We have not yet concluded whether lazy evaluation is appropriate for Systems Haskell, although, to increase predictability, we suspect that it may be necessary to make strict evaluation the default.

6. ACKNOWLEDGMENTS

The authors would like to thank the members of the Programatica project at Portland State University for helpful discussions and comments on the ideas presented here.

7. REFERENCES

[1] Sébastian Carlier and Jérémy Bobbio. hOp. `http://etudiants.insia.org/~jbobbio/h0p/`, 2004.

[2] Manuel M. T. Chakravarty and the Haskell FFI Team. *Haskell 98 Foreign Function Interface (1.0)*, 2003. `http://www.cse.unsw.edu.au/~chak/haskell/ffi`.

[3] Iavor S. Diatchki and Mark P. Jones. Strongly Typed Memory Areas. In *Proceedings of ACM SIGPLAN 2006 Haskell Workshop*, pages 72–83, Portland, Oregon, September 2006.

[4] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. High-level Views on Low-level Representations. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 168–179, Tallinn, Estonia, September 2005.

[5] Iavor Sotirov Diatchki. *High-Level Abstractions for Low-Level Programming*. PhD thesis, OGI School of Science & Engineering at Oregon Health & Science University, May 2007.

[6] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. A memory allocation model for an embedded microkernel. In *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, Sydney, Australia, January 2007.

[7] Guangrui Fu. Design and Implementation of an Operating System in Standard ML. Master's thesis, University of Hawaii at Manoa, August 1999.

[8] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proceedings of the Eighth Asia-Pacific Computer Systems Architecture Conference (ACSAC'03)*, Aizu-Wakamatsu City, Japan, September 2003.

[9] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A Principled Approach to Operating System Construction in Haskell. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 116–128, Tallinn, Estonia, September 2005.

[10] Robert Harper, Peter Lee, and Frank Pfenning. The Fox project: Advanced Language Technology for Extensible Systems. Technical Report CMU-CS-98-107, School of Computer Science, Carnegie Mellon University, January 1998.

[11] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.

[12] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual (Volume 3a)*, January 2006. `http://www.intel.com/products/processor/manuals/index.htm`, date viewed: 25 April 2007.

[13] Mark P. Jones. A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.

[14] Mark P. Jones, Magnus Carlsson, and Johan Nordlander. Composed, and in Control: Programming the Timber Robot. Technical report, OGI School of Science & Engineering at OHSU, August 2002.

[15] Bernhard Kauer. L4.sec Implementation - Kernel Memory Management. Diploma Thesis, Dresden University of Technology, May 2005.

[16] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[17] L4ka Team. *L4 eXperimental Kernel Reference Manual*, January 2005. `http://l4ka.org/`.

[18] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Ken Birman, and Robert Constable. Building Reliable, High-Performance Communication Systems from Components. In *Proceedings of the Seventeenth ACM Symposium on Operating System Principles*, pages 80–92, Kiawah Island Resort, SC, USA, December 1999.

[19] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, pages 468–479, San Diego, CA, June 2007.

[20] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[21] Philip Wadler and Stephen Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the Sixteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76, Austin, TX, USA, January 1989.