Filet-o-Fish: practical and dependable domain-specific languages for OS development

Pierre-Evariste Dagand
ENS Cachan-Bretagne
France

Andrew Baumann Timothy Roscoe
Systems Group, ETH Zurich
Switzerland

ABSTRACT

We address a persistent problem with using domain-specific languages to write operating systems: the effort of implementing, checking, and debugging the DSL usually outweighs any of its benefits. Because these DSLs generate C by templated string concatenation, they are tedious to write, fragile, and incompatible with automated verification tools.

We present Filet-o-Fish (FoF), a semantic language to ease DSL construction. Building a DSL using FoF consists of safely composing semantically-rich building blocks. This has several advantages: input files for the DSL are formal specifications of the system's functionality, automated testing of the DSL is possible via existing tools, and we can prove that the C code generated by a given DSL respects the semantics expected by the developer.

Early experience has been good: FoF is in daily use as part of the tool chain of the Barrelfish multicore OS, which makes extensive use of domain-specific languages to generate low-level OS code. We have found that the ability to rapidly generate DSLs we can rely on has changed how we have designed the OS.

1. INTRODUCTION

The use of domain-specific languages (DSLs) to generate code for operating systems is a well-known technique, and it appears to offer significant benefits: programmers can express high-level ideas about the system at hand and avoid writing large quantities of formulaic C boilerplate. However, the idea has achieved little traction in the OS community so far: with the notable exception of interface definition languages for remote procedure call (RPC) stubs, most OS code is still written in a low-level language such as C. Where DSL code generators are used in an OS, they tend to be extremely simple in both syntax and semantics.

We conjecture that the effort to implement a given DSL usually outweighs its benefit. Based on our own experience developing code for several operating systems over the years, we identify several serious obstacles to using DSLs to build a modern OS: specifying what the generated code will look like, evolving the DSL over time, debugging generated code, implementing a bug-free code generator, and testing the DSL compiler.

We present Filet-o-Fish (FoF), which addresses these problems by providing a tool to build correct code generators from semantic specifications, in contrast with the traditional approach of using code templates and string concatenation. DSL compilers built using FoF are quick to write, simple, and compact, but encode rigorous semantics for the generated code. They allow formal proofs of the runtime behavior of generated code, and automated testing of the code generator based on randomized inputs, providing greater test coverage than is usually feasible in a DSL.

The results are DSL compilers that OS developers can quickly implement and evolve, and that generate provably correct code. We use FoF to build a number of domain-specific languages used in Barrelfish [1], a new OS for heterogeneous multicore systems. This leads to a DSL-based methodology of OS development that we have found useful in building Barrelfish.

In the next section, we make a case for the use of DSLs in operating systems and identify the shortcomings of current techniques. In Section 3, we demonstrate that FoF is a practical toolkit for the implementation of DSLs, using the example of Hamlet – a key component of Barrelfish that specifies and builds the capability infrastructure. In Section 4, we elaborate on the case for dependable code generation. Finally, in Section 5, we situate FoF in the context of other approaches to dependable OS code, and conclude in Section 6.

2. BACKGROUND

Using little languages to aid in software development is, in general, a tried-and-trusted technique [13]. The specific use-case we consider in this paper is using one or more domain-specific languages to build parts of the kernel and basic subsystems of an OS, by generating low level code (e.g. C) from a high-level specification of functionality.

This approach is typified by the Devil language for hardware access [11]. An OS programmer describes the register set of a hardware device in the high-level Devil language, which is then compiled into a library providing C functions to read and write values from the device registers. In doing so, Devil frees the programmer from having to write extensive bit-manipulation macros or inline functions to map between the values the OS code deals with, and the bit-representation used by the hardware: Devil generates code to do this automatically.

Conceptually, such a DSL both abstracts away low-level details and justifies the abstraction by its semantics. In principle, it reduces development time by allowing the programmer to focus on high-level abstractions. The programmer needs to write less code, in a language with syntax and type checks adapted to the problem at hand, further reducing the likelihood of errors.

Despite this, neither mainstream commercial operating systems (such as Windows, Linux, and BSD) nor research operating systems use DSLs in this way, other than for RPC stub generation. We claim there are a number of reasons for this, although performance is unlikely to be one of them: it is relatively easy to hand-optimize code in critical cases, even within a DSL framework. Instead, our argument is that DSLs today simply do not deliver enough benefit to programmers (over and above hand-coding formulaic C) to justify the costs of building and using them.

The first difficulty is merely that it is a lot of effort to build a compiler for a domain-specific language beyond very simple syntactic sugar. While the DSL syntax itself may be easy to design given an understanding of the problem at hand, the developer must then build a parser for the language and a code generator to output C, C++, or Sing# [7] for example.

Efficient tools for rapidly building parsers are now available (such as the Parsec library [10] we use for the languages described in this paper). In reality, most of the implementation work for a DSL is in the backend that takes the abstract syntax tree produced by the parser, and generates code to be compiled and linked into the OS. In most existing DSLs for OS development (including the early ones we used in developing Barrelfish), this amounts to concatenating strings together to generate code from inline templates. For example, nearly half the source lines of the Devil compiler[1] are devoted to the embedding of a C AST – Devil symbolically manipulates a syntactic abstraction of C, hence complicating the collection and verification of semantic properties, such as in data-flow analyses.

The second challenge is in deciding *what* code to generate. The generated code is likely to be quite specific to a particular system, and since one is essentially writing a program to generate a program, it is usual to hand-code several cases in C before attempting to write the code generator. Unfortunately, having done this, the task of reproducing this work again in the DSL compiler is often unappealing.

The third difficulty is that even when the backend has been written, it does not provide a great deal of assurance for the OS programmer. While the DSL may be semantically rich, this semantics remains informal, and thus cannot be formally reasoned about. More importantly, there is no guarantee that the generated code is correct, whereas hand-coding (for example) device access in C allows the programmer to at least see the complete consequences of his or her code. Systems programmers are frequently suspicious of such DSL compilers, even when they can look at the generated C, in part because they understand how easy it is to make mistakes when writing string concatenation code.

The fourth problem occurs when the output of the compiler is indeed buggy. It is notoriously hard to pinpoint compiler bugs, particularly in OS-level code, and the effort required to diagnose such bugs in a DSL compiler (which will inevitably be less mature, albeit less complex, than one for a more mainstream language) is a barrier to adoption. Even testing a DSL compiler with adequate coverage is a challenge – after 6 months of regular use, a graduate student working in our group found a bug in the DSL compiler we use for device access (similar to Devil); the symptom was a kernel-level segmentation fault at boot time.

Finally, it is hard to evolve such a language. As systems and requirements change, alterations to the DSL syntax or semantics, or the interface between the generated code and the OS, entail corresponding changes to the code generator. This further amplifies the problems of programming and debugging.

[1]From http://phoenix.labri.fr/software/devil/distrib/devil-src.Linux-2.tgz counted using David A. Wheeler's "SLOCCount".

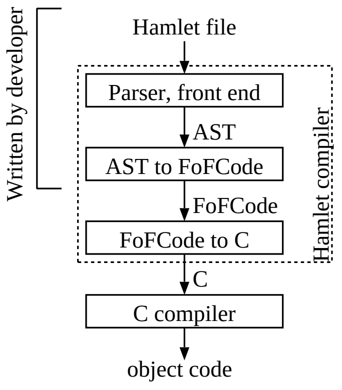


Figure 1: Implementing the Hamlet DSL in FoF

Standing back for a moment, these difficulties mostly arise from the fact that the semantics of such DSLs are only specified informally, and are never propagated through to the generated code. Our goal with FoF is to address this: by attaching stronger semantics to the abstract syntax of a given DSL, we can generate C that a programmer can trust, facilitate automated checking and testing with high coverage, and at the same time reduce the complexity and development time of a DSL compiler. Our goal is to create DSLs that are both practical (easy to implement and use, efficient) and dependable (programmers can rely on their correctness, and the generated code is intuitive and clear).

3. PRACTICAL DSL CONSTRUCTION

Our approach with FoF is to raise the level of abstraction at which a DSL is developed – rather than concatenating strings of C code, a FoF DSL compiler emits code only through the use of higher-order combinators that encode the essential elements of C. From a technical point of view, FoF is simply an *embedding* of C in a functional language, here Haskell. However, because the syntax and semantics of C are extremely weak, we abstract away a large amount of detail. The result is a semantically-clear embedding directed by a mechanized semantics. Using FoF, a back-end can safely manipulate an abstraction of C in the form of an embedded language. Instead of being a string concatenation function, a back-end is now implemented as a composition of FoF building blocks.

To illustrate the use of FoF as a tool for constructing DSLs, we use *Hamlet*, one of four DSLs implemented to date in the development of the Barrelfish OS [1]. As is usual for DSLs, Hamlet defines both a language to specify the Barrelfish capability system, and a compiler to generate code implementing this specification.

Barrelfish has a capability system modeled on seL4 [9] that uses kernel-mediated capabilities both as a security mechanism and for explicit user-level control over kernel memory management. One important operation in this model is *retype*, which takes an existing capability and produces one or more derived capabilities. This operation is used to create new kernel-level memory objects (such as page tables or execution contexts) from capabilities to raw regions of RAM, and thus its correctness is critical to the security of the system.

The behavior of retype is determined by an informal specification of the capability type system, which defines valid capability types, their properties, and their relationships. In particular, this specification describes the retyping policy, *i.e.* given a capability, to what capabilities it can be retyped. As the capability system is

```
/* Phys address range */
cap PhysAddr {
  retype_to {
    RAM:      {base, bits},
    DevFrame: {base, bits},
    PhysAddr: {base, bits}
  };
  eq paddr base;
  eq uint8 bits;
};
```

(a) Input file

```
validateRetypeCode dstType (srcTypeV, validTypesP) =
  do
    return (srcTypeV, (do
      returnc $ cond validTypesP))
  where
    cond validTypes  = foldl orType false validTypes
    orType x srcType = x .||. (dstType .==. srcType)
```

(b) Haskell source of the Hamlet compiler

```
bool is_well_founded(enum objtype fof_y1,
                     enum objtype fof_y2){
  switch (fof_y1) {
  case ObjType_PhysAddr: {
    return ((((false
        || (fof_y2 == ObjType_RAM))
        || (fof_y2 == ObjType_DevFrame))
        || (fof_y2 == ObjType_PhysAddr)));
    break;
  } ...
```

(c) Example of generated C code

Figure 2: Extracts from Hamlet DSL

bound to evolve often, for example with the introduction of new capability types, its implementation needs to be maintainable. Moreover, because it manipulates low-level data-structures, the code is tedious and error-prone to write. Therefore, the Barrelfish capability system forms a good candidate for the use of a DSL, namely Hamlet.

The role of Hamlet is twofold. First, based on a high-level description of the capabilities, a sample of which is shown in Figure 2a, it defines the corresponding data-structures. This amounts to building a set of complex data-types, based on a combination of `struct`, `union`, and base C types. Second, it generates code for a number of predicates that check the validity of user-initiated operations. This consists of translating the high-level policy into low-level manipulations of the capability data structures.

The structure of the Hamlet compiler is shown in Figure 1, and is illustrative of a typical FoF DSL. The DSL developer implements a front-end, usually comprising a parser and some syntax and type checks, and a back-end, which converts an AST to FoF code through the composition of FoF constructs. The FoF compiler then translates this to C.

The process of building a front-end is well-understood, and is made easier by tools such as Parsec [10]. We therefore focus on the back-end, which in the case of Hamlet is responsible for generating two outputs: a C header file containing the data structures, and a source file containing the implementation of the predicates.

An extract of the FoF back-end code for Hamlet and the corresponding generated C are shown in Figures 2b and 2c. This is part of the Hamlet compiler responsible for generating the C function `is_well_founded` – a predicate that determines whether a given source capability type may be retyped to a requested destination type. In particular, `validateRetypeCode` is called repeatedly to generate the multiple cases of a large `switch` statement on the source capability type. The tuple it returns consists of a matched case (`srcTypeV`, or `ObjType_PhysAddr` in the C code) and a series of statements making up the body. Here, the body is a single statement that returns true if the destination type is one of the legal retyping paths and false otherwise.

From this small example, we can observe several important properties of a FoF DSL. First, the combinators used to specify the generated C code (here `returnc`, `.||.` and `.==.`) are similar in structure to the C that we may write, with the omission of syntactical details such as semicolons and the `break` statement. However, through the use of FoF, we have much stronger guarantees than could be achieved by concatenating strings of code: the compiler ensures that the generated code is valid by construction and always *compiles*. Whereas this guarantee is purely syntactic, the following section shows how FoF can be leveraged to generate correct-by-construction C, *i.e.* code respecting the high-level semantics specified by the DSL designer.

Second, the embedding in Haskell promotes a higher-order programming style: FoF code fragments are strongly-typed values that, as such, can be manipulated as data without affecting the validity of the resulting C code. For example, the returned expression in the example is generated by folding the `orType` function (using the standard Haskell `foldl`) over the list of valid destination types. As a result, the code for a FoF-based DSL is typically very concise – the entire Hamlet compiler consists of 700 lines of Haskell.

Finally, FoF enhances the re-use of components between different DSLs, or between different parts of the same DSL. Because the strict typing of Haskell applies to all DSL code, and because the DSL back-end does not directly manipulate or generate strings of source code, the developer need not worry about details such as assigning unique variable and function names in different contexts.

4. DSL DEPENDABILITY WITH FOF

In the previous section we showed how FoF makes it easier to implement DSLs by allowing the developer to write the backend as a collection of typed combinators rather than string concatenations. We now focus on the other main goal of FoF: to improve the correctness of OS code by using DSLs with dependably correct implementations. We emphasize at this point that FoF should *not* be seen as a programming language. FoF is indeed a safe abstraction of C embedded in a functional language, and must naturally be compiled to C (or another systems programming language) to be useful. However, FoF is first and foremost a *semantic language*: it gives a meaning, the semantics, to a DSL.

This novelty has far-reaching consequences. For example, let us assume that in Figure 2b the developer mistakenly wrote `.&&.` instead of `.||.` in the last line, hence replacing an *or* by an *and*. In a syntactic framework, there is no way to catch this mistake: both combinators are meaningless symbols taking two arguments of the same type and returning an element. On the other hand, FoF automatically builds an executable expression denoting the semantics of this code. In the remainder of this section, we show how this allows the the developer to informally reason about the code, automatically test it, or perform a formal correctness proof. Needless to say, at the semantic level, the difference between an *and* and an *or* is striking.

A cornerstone of the greater dependability of FoF-based DSLs is the correctness of the FoF-to-C compiler, for which a formal proof exists for a representative subset of the language [5]. This correctness result ensures that the semantics of FoF are preserved when compiled to C. Given this, it is sufficient to reason from a given DSL down to FoF code without needing to consider the generated C code, since we are guaranteed that the generated C code honors the FoF semantics. Furthermore, since the FoF backend is implemented as Haskell combinators, this reasoning process is made

$$\begin{tikzcd}
\boxed{\text{QuickCheck}} \arrow[r, "\text{Hamlet Semantics}"] \arrow[d, "\text{AST}"] & \text{valid/invalid} \arrow[d, leftrightarrow, "\cong"] \\
\boxed{\text{FoF Interpreter}} \arrow[r, "\text{FoF value}"] & \text{true/false}
\end{tikzcd}$$

Figure 3: Testing infrastructure of Hamlet

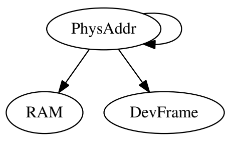


Figure 4: Automatically-generated retyping graph

much easier.

As a working example, we discuss the correctness of the Hamlet-generated predicate `is_well_founded`, a portion of which was presented in the previous section. Proving the correctness of this predicate amounts to showing that the following diagram commutes:

$$\begin{tikzcd}
\text{Hamlet input} \arrow[r, "\text{Retype policy}"] \arrow[d, "\text{Hamlet compiler}"] & \text{valid/invalid} \arrow[d, leftrightarrow, "="] \\
\text{FoFCode} \arrow[r, "\text{FoF semantics}"] & \texttt{true/false}
\end{tikzcd}$$

That is: given a Hamlet input, the retype policy – encoding whether it is valid or not to retype a capability to another – should give the same result as the FoF code generated by the compiler, interpreted according to the semantics of FoF.

We describe the semantics of FoF by means of an interpreter (as opposed to the C code generator) that evaluates FoF code to FoF values. Because this interpreter is implemented in a purely functional style, it denotes a *pure semantics* of the FoF language. We build on the work of Swierstra [14] on the functional semantics of effects: whereas semantics are traditionally described in an operational or denotational framework, Swierstra proposes a lightweight semantics framework based on pure functions, hence implementable – and executable – in a pure functional language. Therefore, proofs of correctness of DSLs can be carried out by equational reasoning, and their semantics can be described in a theorem prover [14]. Being able to actually *execute the semantics* also allows our users to test their FoF code in Haskell, instead of having to test the resulting C code.

As a first step towards a provably-correct DSL compiler, we demonstrate the use of QuickCheck [3] on the Hamlet compiler. QuickCheck is a library for random testing in Haskell. Being a library, it is easy to use, but also efficient. QuickCheck automatically generates data-structures and tests the truth values of user-defined properties. We instruct QuickCheck to generate random yet valid ASTs, and use the FoF interpreter to test the output of the Hamlet compiler against the expected behavior. Note this assumes that the Hamlet *parser* is correct. This assumption is usually made by verified compilers like CompCert [2]. Moreover, the syntax of Hamlet is simple, hence the parser is small and trustable by inspection.

The properties we are to test must *specify* the semantics of Hamlet with respect to `is_well_founded`. Informally, its semantics is the following: given a source and a destination capability type in the AST, `is_well_founded` returns true if and only if the destination type belongs to the valid destinations of the source type. This property is directly expressed in Haskell.

The corresponding infrastructure is shown in Figure 3. QuickCheck generates random ASTs. For a given AST, the Hamlet semantics dictates whether retypings are valid or invalid. The same AST is also compiled to FoF code that is later interpreted: retyping a source capability to a destination capability should succeed if and only if this is allowed by the Hamlet semantics. We have tested this property by generating random input ASTs, and found extensive application of such random testing to be tractable [5].

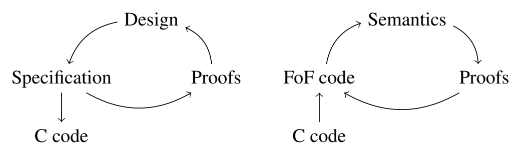
This approach compares favorably to conventional compiler testing, where the developer writes test cases, each of which are then compiled and compared with an expected output. The FoF approach with QuickCheck has four clear advantages. First, no test cases must be written, as QuickCheck generates these itself. Second, coverage is greater because the tests generated by QuickCheck are random. Third, checking the correctness of a given output is precise, because we are able to test whether the output is *semantically* correct or not. Finally, since the process is entirely automated, testing can be done as a background task, and coverage can be increased by devoting more CPU cycles to testing.

Of course, QuickCheck testing does not provide the same level of assurance as a direct proof of the semantics of the Hamlet front-end. To date we have not tried to produce such a proof, in part because QuickCheck provides us (in our opinion, so far) sufficient assurance for practical OS development. Moreover, the property we test with QuickCheck ensures that the generated code is semantically correct for the given capability specification. Although the Hamlet compiler is not proved to preserve the DSL semantics for all inputs, the user is *guaranteed* that, if the compiler does not detect an error for her input, then the generated code *is* correct. Thus, FoF allows the DSL designer to build a lightweight and efficient translation validation infrastructure [12] for her DSL.

So far, we have formally verified the correctness of the FoF-to-C compiler for a subset of the language, thoroughly tested the correctness of Hamlet using QuickCheck, and shown how the Hamlet compiler gives us the guarantee that the generated code is correct. For the DSL designer, this combination of features substantially increases the reliability of DSL code over non-semantic methods, and offers the possibility of exhaustive compiler verification with reasonable effort.

The ability to rely on the correctness of a DSL has affected how we have designed aspects of the Barrelfish OS. In particular, Hamlet means that we can extend the type system for capabilities without requiring cross-cutting changes in the hand-written C code that comprises much of the OS. This has encouraged us to move more OS functionality into this type system. For example, all the required checks on user manipulation of hardware page tables on assorted architectures are now performed this way: we have different capability types for all levels and flavors of page table entries. This has greatly simplified the virtual memory system, but would have been prohibitively expensive without dependable high-level language support.

Similarly, DSLs have influenced the documentation and specification of the OS. Whereas we previously wrote an informal specification of the capability system by hand, Hamlet makes it possible to generate a human-readable specification from the Hamlet description. For example, Figure 4 shows a diagram of capability retyping paths automatically generated from the snippet in Figure 2a.



(a) Top-down approach (b) Bottom-up approach

Figure 5: Reliable design approaches

5. RELATED WORK

We have discussed DSL-based approaches to improving OS code in Section 2. A number of other language-related techniques are complementary to FoF; we present a brief survey here.

One such approach is to implement the OS in a safer low-level language than C, such as Cyclone [8], Deputy [4], or Sing# [7]. While undoubtedly leading to more dependable code, these languages remain by their nature highly general (and hence applicable to OS code), while lacking the ability to directly express very specific OS abstractions (such as the type and representation of capabilities in Barrelfish) in the way that a custom language such as Hamlet can. The contrast is thus to some extent artificial: an attractive hybrid approach is for FoF to generate code for one of these languages, thereby exploiting the best features of both.

The meta-compilation technique has been successfully applied to large-scale systems [6]. An API (such as that of the Linux kernel) implicitly defines a high-level language, and is thus governed by a semantics. This semantics is formalized and used to automatically check C code (which does not itself have strong semantics). This technique can provide useful results but not completeness: it is geared to finding errors, instead of proving (or constructing) their absence. In contrast, FoF gives a semantics to the generated low-level code. Because we can reason about a high-level language supported by a mechanized semantics, we do not give up completeness, and any verification effort can rely on more powerful mathematical and software tools.

Finally, FoF is complementary to the standard top-down approach to reliability, exemplified most recently by the seL4 OS [9]. To prove the correctness of a system top-down, one starts with a high-level model which is then refined to machine code, or at least C. In the case of an OS, the high-level model must handle every corner case encountered by the low-level models. In practice, the high-level design is iteratively improved while the refinement mapping proofs to the lower-level models are worked on. Each time the high-level model is modified, the proofs must be updated (or restarted from scratch).

Figure 5 compares this with the bottom-up approach adopted by FoF. FoF starts with a mature infrastructure in C. The DSL then abstracts the essence of the infrastructure and formalizes its semantics. Because a DSL input *is* a specification, the correctness of the system depends only on the correctness of the DSL compiler. Unlike the top-down approach, which must deal with the system *en masse*, FoF allows developers to concentrate on critical parts of the system. The approaches are not exclusive: the bottom-up approach can be used in top-down proofs, since a DSL shortens the distance between the top-level model and its implementation, simplifying both the proofs and their evolution.

6. CONCLUSION AND FUTURE WORK

In this paper, we have argued for a radically new approach to system code reliability. We advocate a bottom-up philosophy using semantically-rich DSLs, combined and stacked up together to incrementally build critical components of the system. To this end, we have developed FoF. Thanks to its expressiveness, the implementation of DSLs is easier. Thanks to its functional semantics, the verification of DSLs is tractable. Hence, in one go, we are able to abstract away their functionality and formalize their semantics.

FoF is still at an early stage in development, and we expect to gain more experience in building DSLs in the context of Barrelfish. We also plan to implement a translation validation infrastructure [12] in the FoF-to-C compiler. While we have a pen and paper compiler correctness proof for a subset of FoF, we do not yet have complete verification. A validating FoF compiler would provide a proof of correctness of the generated C for a given input. This provides a less burdensome logical framework than that in exhaustive correctness proofs, and by virtue of its integration with the compiler, would track the evolution of FoF.

Beyond this, FoF enables a DSL-intensive development process of low-level systems code, and we are interested in where this leads. One possibility is that the various little languages used for Barrelfish converge – it is already clear that there are common features between RPC stub generation and hardware memory layout, for example. In the limit, C becomes a "glue" language for high-level descriptions with strong semantics.

References

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proc. ACM Symposium on OS Principles*, Big Sky, MT, USA, Oct. 2009.

[2] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. *Formal Methods*, pages 460–475, 2006.

[3] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[4] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. Necula. Dependent types for low-level programming. *Programming Languages and Systems*, pages 520–535, 2007.

[5] P.-E. Dagand. Language support for reliable operating systems. Master's thesis, ENS Cachan-Bretagne, June 2009.

[6] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. 4th Symposium on OS Design and Implementation*, Oct. 2000.

[7] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, Apr. 2007.

[8] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. USENIX Annual Technical Conference*, pages 275–288, 2002.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. ACM Symposium on OS Principles*, Big Sky, MT, USA, Oct. 2009.

[10] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[11] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: an IDL for hardware programming. In *Proc. 4th Symposium on OS Design and Implementation*, pages 17–30, 2000.

[12] G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[13] M. Shapiro. Purpose-built languages. *Commun. ACM*, 52(4):36–41, 2009.

[14] W. Swierstra. *A Functional Specification of Effects*. PhD thesis, University of Nottingham, Nov. 2008.