

Language Support for Reliable Operating Systems

Master's Thesis

Pierre-Evariste DAGAND

June 2009

Under supervision of: Prof. Timothy Roscoe
Systems @ ETH Zurich

Contents

1	Introduction	3
2	The Barrelfish Operating System	5
2.1	Barrelfish architecture	5
2.2	Enhancing safety	6
3	Filet-o-Fish	9
3.1	Design rationale	9
3.2	Designing a DSL with Filet-o-Fish	11
3.3	Support for formal verification	11
4	Motivating Example: Fugu	13
4.1	The state of the art	13
4.2	Error definition language	13
4.3	Fugu back-end	14
5	Filet-o-Fish Compiler Correctness	18
5.1	Problem statement	18
5.2	Filet-o-Fish dynamic semantics	20
5.2.1	Evaluating pure expressions	20
5.2.2	Evaluating statements	20
5.3	Filet-o-Fish compiler specification	23
5.3.1	Compilation of pure expressions and types	23
5.3.2	Compilation of statements	23
5.4	Correctness proof	26
6	Case Study: Hamlet	31
6.1	Barrelfish capability infrastructure	31
6.2	Generating the capability infrastructure	31
6.3	Assessing Hamlet correctness	33
7	Discussion	36
8	Conclusion	37
A	Filet-o-Fish Compiler Correctness Proof	39

1 Introduction

Operating systems are among the most complex software developed to date. Being at the interface with the hardware, they have to deal with unpredictable behavior, yet they are asked to present a consistent interface to the applications executing on top of them. Hence, the operating system developer has to master this uncertainty to fulfill users' expectations.

As we can imagine, such development is extremely difficult and error-prone. Moreover, operating systems are historically developed in the C programming language. Because this language offers access to the low-level details of the machine, this makes it possible to express *bit twiddling*, a requirement to be able to implement an operating system. Because of these strong, low-level requirements, C is still the language of choice, even after 40 years of progress in programming language design. Indeed, on one hand, this progress has been governed by more abstraction power and more high-level constructs. On the other hand, operating system developers cannot afford invasive abstractions as they hide the low-level behavior of the code as well as the physical layout of data.

This proximity to the hardware also increases the difficulty of understanding the behavior of the code: hardware and software are so tightly-coupled that it is impossible to understand one without the other. Hence, to understand the behavior of operating system code, we first need a specification of the underlying hardware. Whereas an informal specification can be sufficient for informal debugging, automatic or semi-automatic software verification techniques cannot handle this inaccuracy. Software verification techniques require a perfect specification of hardware devices, whereas commodity hardware, frequently, does not even comply to any well-defined specification and requires ad-hoc workarounds. Another difficulty is the use of pointers, in particular function pointers: whereas this idiom is pervasive in an operating system, it is hardly dealt with by software verification tools.

However, the usage of C is double-edged: whereas it allows low-level operations, it also sacrifices some high-level guarantees. First and foremost, C provides an extremely lightweight type system. Therefore, it is up to the developer to maintain type invariants, which would have been automatically enforced by a type-checker in a strongly typed language. Type-safety is not only helpful while writing code from scratch, it is also of first importance for its painless evolution. The second drawback of C is the absence of memory-safety: in privileged mode, a pointer can read or write to any address in the computer memory. Therefore, even absolutely correct code can be endangered by a completely unrelated wild pointer. To alleviate this issue, operating systems usually rely on hardware protection, such as the memory management unit (MMU). However, this mechanism has significant performance cost and the protection granularity remains too coarse grained to compete with usual guarantees offered by memory-safe languages. Last but not least, the modularity mechanism offered by C, or lack thereof, makes the design and maintenance of abstractions extremely fragile. Whereas objects, aspects, or polymorphisms are the corner stone of modern languages, C does not natively support them, and the developer is required to carefully encode and respect abstraction boundaries.

The extreme difficulty of applying software verification techniques to operating systems also has a big impact on their safety. First of all, being unable to use automatic verification tools forbids any claim about the behavior of the system. Whereas in-depth manual reviews might give some confidence, the lack of a formal, exhaustive proof threatens the safety of the whole architecture. Moreover, because user-level processes crucially rely on its correctness, a single bug in the operating system is enough to cause the whole system to malfunction. Worse, the visible effect of a bug frequently occurs far from its cause, after a long succession of events that might be hard to reproduce. Tracking down such bugs is known to be a long and difficult process.

Domain-specific languages for reliable operating systems

We have met these challenges in the context of the Barrelfish operating system [2, 35]. Faced with the classical shortcoming of C, the Barrelfish team have designed a set of domain-specific languages (DSLs) to translate high-level descriptions into low-level C code. In particular, these efforts have been focused on error-prone code, such as device interactions and communication interfaces. Using these languages, it is possible to regain type and memory-safety as well as ad-hoc re-usability without having to suffer the burden of a general-purpose high-level language. In this setting, our work has aimed at pushing this approach one step further. Whereas these DSLs are currently specified by an informal, in-English semantics, they still lack a formal, mechanized semantics from which a human or an automated tool could reason about.

We are convinced that formally specified DSLs can foster a new way of designing reliable operating systems. Traditionally, safety is enforced in a top-down manner: the system is implemented in a high-level language and proved correct in this formalism. Then, a sequence of proofs shows that this correctness is preserved along the way to machine code. As we have seen above, this approach is not practical for commodity operating systems. Therefore, we advocate a bottom-up approach: while developing the operating system, we are to face repeating patterns that would deserve to be abstracted away and formalized. Driver interactions or communication interfaces are two such examples. By defining a DSL per problem, we are able to address two issues at the same time: first, we abstract the low-level details behind semantically-rich DSL constructs and, second, we formalize the problems thanks to a mechanized semantics. Unlike the top-down approach that aims at solving all problems at once, we propose a philosophy of small-steps, in which abstractions are iteratively built and stacked up.

Our contribution

This lead us to design *Filet-o-Fish*, hereafter abbreviated FoF, as a language to specify the semantics of DSLs. As a side-effect of its extreme precision, FoF can also be compiled to C. Therefore, DSL compilers can entirely rely on FoF to implement the *back-end*, which transforms a processed input into C. Not only does FoF provide a formal ground to DSLs but it also considerably eases their development.

To support our claims, we have developed two DSLs, Fugu and Hamlet. Fugu is a self-contained proof-of-concept that conveys the essence of FoF design philosophy. Beyond the pedagogical interest, it also alleviates the difficult problem of error management in an operating system. Hamlet, on the other hand, is a full-featured DSL: it aims at automatically generating the resource accounting infrastructure from a set of declarative rules. Because resource management is safety-critical, Hamlet also demonstrates how FoF can be leveraged to reason at a higher-level thanks to its mechanized semantics. A correctness proof of the FoF-to-C compiler backs our position: we do not have to reason about C code anymore and we can conveniently reason about FoF code only, being ensured that the generated C code respects the FoF semantic.

Overview

This report is organized as follows. In Section 2, we further describe the Barrelfish operating system and motivate the use of DSLs in this context. Then, in Section 3, we describe Filet-o-Fish design and implementation. We motivate its strengths in Section 4 by describing the Fugu Error Definition Language. In this section, we show that FoF makes the implementation of DSLs easier. Then, in Section 5, we prove the correctness of our compiler to C. Convinced of the correctness of FoF, we implement a more ambitious language for specifying resource management policies, Hamlet, in Section 6. Thanks to this case study, we demonstrate that the reliability of system code is enhanced by the use of DSLs. Finally, we conclude in Section 8. The reader is referred to our bibliography study [11] for an in-depth literature survey.

2 The Barrelfish Operating System

In the following, we introduce the reader to the Barrelfish operating system, with a bias toward the mechanisms that enhance its reliability. In Section 2.1, we describe its architecture and its characteristics. In Section 2.2, we study two DSLs widely used in Barrelfish. Finally, we pinpoint some shortcomings of their design and draw some conclusions.

2.1 Barrelfish architecture

Barrelfish is an operating system built from scratch at ETH Zurich, in collaboration with Microsoft Research Cambridge. It is developed by a small group of developers: Andrew Baumann, Simon Peter, Jan Rellermeyer, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhanian at ETH, and Paul Barham, Tim Harris, and Rebecca Isaacs at Microsoft Research. Barrelfish implements a novel architecture, termed *multikernel*, to tackle the issues raised by the generalization of multi-core machines. These machines call for operating systems that are able to scale to a large number of heterogeneous cores.

First of all, Barrelfish is a *microkernel* [29]. Therefore, the portion of code running in privileged mode is very small – about 7,000 lines of code – and only provides functionality that cannot be implemented in user-mode. By implementing very basic services, the kernel is simpler, hence more trustable. Because the kernel is part of the trusted computing base (TCB), the remainder of the system can rely on this reduced TCB to offer more powerful services. The kernel is mostly composed of architecture specific code, the CPU driver, that manages the processors. Two schedulers are also implemented, to multiplex user-mode processes. The low-level communication primitives are also implemented in the kernel. Finally, resource management is also part of the kernel. The remaining subsystems one would find in a monolithic operating system, such as device drivers, network stack, file-systems, etc., are implemented out of the kernel and executed in user-mode.

The security model of Barrelfish is inspired by the seL4 microkernel [15]. seL4 is a microkernel for secure embedded systems. It has the appealing characteristic of being entirely formalized and proved correct. Whereas Barrelfish, targeting commodity hardware, had to give up exhaustive verification, it builds upon some security mechanisms engineered by the seL4 team. In particular, the resource management infrastructure is borrowed from seL4 and relies on *partitioned capabilities*, as described in Section 6. Hence, the Barrelfish kernel, if not verified, is trustworthy.

Moreover, Barrelfish is a new point in the operating system design space: it is the first implementation of the *multikernel* model [2]. The multikernel model aims at tackling the challenges raised by new hardware architectures. Indeed, we are observing the generalization of multi-core processors: since 2005, Moore’s law seems to govern the number of cores, instead of CPU frequency. Whereas the increase of CPU frequency had no visible architectural effects – same instruction set, same interfaces – the increasing number of cores cannot be ignored by operating systems, which have to run efficiently on these machines. These new architectures are composed of complex cache hierarchies, with performance impacts depending on their usage. Intricate interconnect topologies ensure inter-core communications: again, performance can be greatly improved by taking advantage of knowledge of the topology. Cache and interconnect performance are also strongly affected by the cache-coherency protocol, which is dependent on the specific architecture, and is very likely to evolve in future CPU generations. Finally, a current trend in processor design is to introduce specialized cores, which would handle some operations more efficiently. It is again up to the operating system to efficiently deal with this heterogeneity.

All these radical changes in the underlying architecture call for a radical change in the operating system organization. The multikernel model consists in three key principles. Firstly, instead of a single monolithic kernel executing on all cores, the multikernel model proposes to run one kernel on each core. This set of kernels then acts as a distributed system, communicating with message-passing primitives. By not relying on the shared-memory abstraction used by traditional operating systems, a multikernel system

```

register status rw addr(base, 0x0008) "Device status" {
    fd      1 "Link full duplex configuration";
    lu      1 "Link up";
    lan_id  2 "LAN ID";
    txoff   1 "Transmission paused";
    tbimode 1 "TBI mode";
    speed   2 type(linkspeed) "Link speed setting";
    asdv    2 type(linkspeed) "Auto speed detection value";
    phyra   1 "PHY reset asserted";
    -       8 mbz;
    gio_mes 1 "GIO master enable status";
    -       12;
};

```

Figure 2.1: Mackerel code snippet: e1000 device status register

can benefit from the wealth of existing, proved distributed protocols such as, for instance, consensus protocols. Moreover, the message-passing abstraction allows dynamic optimizations for the very precise underlying hardware. Relying on messages, this also helps in tackling heterogeneity: the sender does not need to know much detail about the receiver, be it a similar core, a co-processor, a GPU, or even an FPGA. An appealing benefit is that cache-coherency is not a requirement. Hence, it can be relaxed or disabled and the multikernel would benefit from the significative performance improvement.

Secondly, the multikernel model advocates *split-phase* communications, contrasting with the Posix tradition of synchronous system calls – the caller blocks until the kernel answers back. A multikernel decouples the action of sending a request from its reception. This has the effect of relaxing the latency requirement: whereas the latency of synchronous calls is critical, the latency of asynchronous calls is of less importance. Indeed, we are able to *batch* requests, *i.e.* to aggregate several requests in a single message, as well as *pipeline* requests, *i.e.* to send several requests in a row without individually waiting for the responses. These techniques compensate for the higher latency with increased throughput. Moreover, when necessary, blocking calls can still be implemented at a higher-level, as an optimization for latency-sensitive operations.

Thirdly, a multikernel system relies on state replication instead of state sharing. This comes as a consequence of the message-passing interface. Hence, instead of relying on shared memory protected by locks, a multikernel avoids the usual contention and engineering difficulties introduced by locks. A standard technique to reduce locks contention is to replicate states. In a multikernel, the inverse approach is adopted: originally, all states are replicated. They are efficiently kept consistent by a distributed protocol adapted to the problem at hand. Note that this might include transparently using shared-memory protected by locks, hence demonstrating the greater flexibility of this approach.

2.2 Enhancing safety

Now that we have a better understanding of the whole architecture, we are going to study two tools widely used in Barrellfish that aim at enhancing its safety. We show their strengths as well as their weaknesses.

The first tool is *Mackerel*, an interface definition language (IDL) developed by Timothy Roscoe and inspired by Devil [30]. Interacting with device drivers is a complex and error-prone task: this involves tedious sequences of bit-level operations on various hardware registers. Whereas these bit twiddling code are the actual implementation of a protocol specified in the device data-sheet, it is extremely difficult to link these low-level code with the corresponding high-level protocol. It is therefore not surprising if 85% of the system failures in an operating system such as Windows XP are caused by faulty drivers [40]. *Mackerel*, and its ancestor *Devil*, define a high-level language to exhaustively describe device interfaces. Hence, a developer can almost directly translate a data-sheet into the *Mackerel* language. Provided with this description, *Mackerel* compiles it into several C functions to read, write, and print the device registers. Hence, the developer is freed from the tedious bit twiddling work: she just has to specify the device interface and, then, use semantically-rich functions to interact with it.

```

call init_device_irq(
    uint64 class_code,
    uint64 sub_class,
    uint64 prog_if,
    uint64 vendor_id,
    uint64 device_id,
    cap cap
);

```

Figure 2.2: Flounder code snippet: PCI driver initialization message

A typical example of Mackerel code is shown in Figure 2.1. This code snippet is part of the e1000 network card description. At the time of writing, this description is approximately 1200 lines long.

The Mackerel compiler is implemented in Haskell [22], a purely functional language, and works as follows. First, it parses the device description and builds an abstract syntax tree (AST). Starting from this AST, a front-end checks it for syntactic errors – such as undeclared identifiers – and processes the AST in a form suitable for the back-end. The role of the back-end is simply to traverse the data-structures provided by the front-end and translate them into C code. All in all, this process is fairly standard and benefit from a relative simplicity. However, the back-end suffers from an important weakness: it builds its output, a C code, by concatenating strings together. Therefore, we cannot take advantage of Haskell’s strong type system to *ensure* that the resulting string of characters is indeed a syntactically correct C code.

The second tool is Flounder, originally developed by Timothy Roscoe and then taken over by the author. Flounder is a language for defining communication interfaces. It draws its design from classical remote-procedure call stub compiler techniques [5], with some extensions to fit the Barrelfish split-phase message-passing model. Indeed, the Barrelfish kernel only provides low-level communication primitives. Above these a name service has been implemented – to register and connect to services using meaningful identifiers, some marshaling and un-marshaling functions for basic types, a message abstraction providing a `send` operation, and an event-driven interface that handles the reception of messages per service.

Although these abstractions are convenient to write small applications, they do not scale to more complex ones. Indeed, marshaling and un-marshaling code must be carefully kept consistent between the sender and receiver sides. Moreover, a lot of boilerplate, error-prone code must be written that could be automated, such as event handler registrations, generic service logic, etc. A last point is that facing such a burdensome task, the developer is likely to stick with the simplest design possible. However, a good overall design calls for defining as many messages as is semantically relevant. Hence increasing the complexity of the code. Flounder aims at solving this problem by relieving the developer from the burdensome tasks, hence taming the complexity issue.

In Flounder, the developer simply defines messages. Then, the Flounder compiler generates message sending stubs that take care of marshaling. It also generates un-marshaling stubs that call user-defined handlers. Overall, this replaces the per-service event system with a finer and more convenient per-message event-driven abstraction. The developer is only asked to manipulate or instantiate semantically-rich send functions and receive handlers. A sample of Flounder code is shown in Figure 2.2. This code is part of the PCI driver communication interface, which is about 50 lines long.

The mechanics of the Flounder compiler follow those of Mackerel. Therefore, it suffers from the same drawbacks. Moreover, because these back-ends can be seen as ad-hoc string concatenation functions, there is almost no code re-use between Flounder and Mackerel and absolutely no functionality re-use. Therefore, the work done in the context of Mackerel is of no use in Flounder, whereas both are describing interfaces and share some functionalities, such as marshaling and un-marshaling.

Summary

These two tools are a good illustration of the bottom-up approach to reliability. First of all, they are built around a DSL that both abstracts away the low-level details and justifies the abstraction by an informal semantics. The compilers being small and written in a high-level language, they are also more trustable than the original code. Moreover, the developer has to write less code, in a language designed to solve her problem, with syntax and type checks adapted to the problem at hand. Hence, her efficiency is increased and the likelihood of errors is reduced.

This approach contrasts with the traditional verification techniques: instead of trying to ensure the correctness of the whole system, we adopt a pragmatic philosophy. Barrelfish being an experiment, its source code has not yet matured and is quickly evolving. Hence, Barrelfish developers first need to gain experience with the low-level APIs. Only later, when the APIs have matured, do they feel the need to formalize these protocols. A DSL is a perfect fit here, as it both formalizes the protocols and offers rich abstractions to the developer.

At the scale of an operating system, we expect the development of a myriad of moderately small DSLs: we have described Mackerel and Flounder; in this report, we present Fugu and Hamlet; and recent work by Padioleau et al. [32] makes a compelling case for the use of more DSLs in system code. As mentioned above, small DSLs are more trustable while increasing developer efficiency. Thanks to the use of Haskell, it is extremely easy to write a parser as well as a compiler front-end. However, this trend is limited by a major weakness: the back-end. As we have seen with Mackerel and Flounder, the current back-end design forbids any sort of re-use. Hence, a back-end is unable to re-use functionality provided by another back-end, and the Haskell type-system is unable to ensure the validity of the resulting C code. Therefore, during the compilation process, we might lose the semantics of the high-level language, *i.e.* output some invalid C code.

3 Filet-o-Fish

In the previous section, we have motivated a bottom-up approach to safer operating system construction and we have described two examples of this principle in action. Doing so, we have identified two major weaknesses: first, the back-ends, as currently implemented, provide extremely weak guarantees on the resulting code. Second, although these languages are semantically rich, this semantic remains informal, and thus cannot be formally reasoned about. To solve these two issues together, we have developed Filet-o-Fish. In the following, we first describe FoF as a framework and as a language. Then, we give a bird’s-eye view of the DSL design process advocated by FoF. Lastly, we emphasize that FoF automatically defines a formal semantics to the languages using it.

3.1 Design rationale

Functional languages, such as Haskell, are powerful enough to build *embedded languages*: inside Haskell, we can define a data type and a set of functions, called *combinators*, operating on this data type. Then, just as in a full-blown programming language, we can write programs by composing these combinators. A foundational example of an embedded language is presented in Figure 3.1. This example comes from Hudak seminal paper [18] on domain-specific embedded languages (DSELs). This example follows the definition above: first, a `Region` type is defined, which represents a geometric region. Then, a combinator `inRegion` is defined that tests whether a `Point` is in a given `Region` or not. The `circle`, `outside`, and `∧` combinators are implemented similarly. The power of this approach is demonstrated by the `annulus` combinator: this function is entirely implemented by the composition of the functions previously introduced. Hence, inside Haskell, it is now possible to compute complex geometric region analysis.

From a practical point of view, FoF is simply an *embedding* of C in a functional language. However, because C syntax and semantics are extremely weak, we had to abstract away a fair amount of confusing details. The result is a syntactically 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. Thus, we solve the re-usability issue raised by the Mackerel and Flounder back-ends. Instead of being a string concatenation function, a back-end is now implemented on the model of the `annulus` function: it is simply a type-safe composition of FoF building blocks.

FoF is defined in two layers, a core and a modular set of extensions. The core of the language consists of *pure expressions*, on the model of the pure expressions of C [7]. The syntax tree of expressions is

```
-- Geometric regions are represented as functions:
type Region = Point → Bool

-- So to test a point's membership in a region, we do:
inRegion    :: Point → Region → Bool
p 'inRegion' r = r p

-- Given suitable definitions of 'circle', 'outside', and ∧:
circle     :: Radius → Region          -- creates a region with given radius
outside    :: Region → Region          -- the logical negation of a region
(∧)        :: Region → Region → Region -- the intersection of two regions

-- We can then define a function to generate an annulus:
annulus    :: Radius → Radius → Region
annulus r1 r2 = outside (circle r1) ∧ circle r2
```

Figure 3.1: Embedded language in Haskell: a geometric region analysis language

<pre> expr ::= int n intsize signedness float f ref type id op1 expr expr1 op2 expr2 sizeof type (type) expr expr1 ? expr2 : expr3 </pre>	<pre> Integers Float Reference Unary expr. Binary expr. Sizeof operator Cast operator Test operator </pre>	<pre> type ::= voidT intT intsize signedness floatT pointer type mode array type struct id fields union id fields function id type type* </pre>
<pre> op1 ::= - ~ ! </pre>	<pre> Unary operators </pre>	
<pre> op2 ::= + - × % << >> & ^ ≤ ≥ < > == != </pre>	<pre> Arithmetic op. Bit-wise op. Relational op. </pre>	<pre> signedness ::= Signed Unsigned intsize ::= I8 I16 I32 I64 fields ::= (id, type)* mode ::= Available Read (b) Filet-o-Fish types </pre>

(a) Filet-o-Fish expressions

Figure 3.2: Filet-o-Fish core language

represented in Figure 3.2(a). It covers the standard integers and floats of varying size and signedness. It also covers unary and binary operations, conditional expressions, the `sizeof`, and the `cast` operations. A novelty is introduced by `ref` that represents a *reference* to a value stored in a variable named `id`. The types of expressions are also part of the core language definition, as shown in Figure 3.2(b). These types mirror the expressions: we declare a `voidT`, `intT`, `floatT`, and `pointer` types, corresponding to `int`, `float`, and `ref` expressions. Note that `voidT` is not inhabited, as one would expect.

Although this core language is sufficient to write simple, pure expressions, it does not support basic constructs such as mutable variable, functions, arrays, structures, or unions. Also, it does not support interaction with externally-defined variables or functions. Hence, we need to be able to incrementally extend the constructs supported by FoF. Similarly, it is natural to consider a compiler from FoF constructs to C. As we will see in the next section, it is also of first importance to be able to evaluate FoF terms. Also, we want to provide a verified compiler to Clight [7], a subset of C described by a mechanized semantics.

This problem is an instance of the expression problem [41]. Given a data type, here the constructs of our language, and some functions operating on this data type, here the compilers and interpreters, we would like to be able to both extend the data type with new constructs without modifying the functions, and to add new functions without modifying the data type. Until recently, this problem had no type-safe solution. Indeed, in the functional paradigm, it is straightforward to extend the set of functions operating on a data type. Extending the data type, on the other hand, requires updating every single function manipulating it. Conversely, in the object-oriented paradigm, it is straightforward to extend a data type, by the means of a new object. However, extending the set of functions for this data type consists of adding new methods to every single object.

Thanks to the work of Swierstra [37], we are able to honor our promises of modularity. Thus, FoF users can incrementally add new constructs to the FoF language as well as add more post-processing functions such as compilers and interpreters. While there has been a fair amount of work in the domain of modular interpreters [28], FoF is, to the best of our knowledge, the first modular compiler. This extreme flexibility is transparent to the user: she is provided with a *monadic* interface, in which she can compose FoF constructs in a programming style close to a traditional imperative style. The syntax of a subset of the currently-implemented constructs is presented in Figure 3.3. The `do` and `return` operators are the standard monadic operations, which are here used to sequence statements. Then, we define three operators on *reference cells* that abstract C variables. A reference cell is created by `newRef`, in a given initial state. It can be read from and written to thanks to `readRef` and `writeRef`. Similarly, we manipulate arrays with `newArray`, `readArray`, and `writeArray`. Interestingly, we can call externally-defined functions,

```

stmt ::= do stmt stmt
      | return expr
      | r ← newRef expr
      | r ← readRef expr
      | writeRef expr expr
      | r ← newArray [expr]
      | r ← readArray expr expr
      | writeArray expr expr expr
      | assert expr
      | ...

```

Figure 3.3: Subset of Filet-o-Fish statements

such as `assert`, provided by the standard C library: it suffices to extend FoF with the corresponding construct. The exhaustive list of constructs supported by FoF is too long to be described in this report, hence we refer the reader to the literate code of FoF [12].

Being freed from the expression problem, we have implemented an interpreter for the FoF language, in addition to a compiler to the Clight language. Being able to interpret FoF code is of first importance: following recent work on functional specification of effects [39], this allows our users to test their FoF code in Haskell, instead of having to test the resulting C code. Thanks to testing tools such as QuickCheck [8], FoF users are able to assess the correctness of their back-end. This is a first step toward more formal verification, and a convincing argument for FoF: we offer the comfort of a high-level model that can be reasoned about and thoroughly tested, with the assurance that the low-level code behaves accordingly.

3.2 Designing a DSL with Filet-o-Fish

Now that we have a better understanding of FoF mechanics, we highlight some guidelines governing the development of a DSL with FoF.

First of all, we want to stress the importance of starting from a mature C code, representing the expected output of the DSL compiler for a particular input. This draft is meant to be the support for discussions and debates with the end-users of the DSL: whereas not everyone is familiar with Haskell and FoF, C is a *lingua franca*. Thanks to this support, users are given the power to influence the design of the DSL, without having to handle its machinery. Moreover, this C file is an inexpensive support for testing and debugging: its behavior can be quickly modified without dealing with the compiler machinery. Another benefit is that this draft is very likely to be used as a Rosetta stone by new users: they will be able to match the high-level descriptions to the corresponding low-level code, hence grabbing the intuition of the high-level semantics of the DSL. We have observed this process several times in our development, and it has significantly lowered the barrier to entry of our DSLs. Conversely, when we failed to provide such self-contained use-cases, the first reaction of our users was to build their own high-level input and observe the compiled code.

A second, potentially disturbing, point is that FoF should *not* be seen as a programming language. So far, we have stressed the fact that FoF is a safe abstraction of C embedded in a functional language. We have also mentioned the practical interest of being able to compile this embedding to C. However, as we will see in the following section, FoF is, first and foremost, a *semantic language*: it gives a meaning, the semantics, to a DSL. The fact that it can be compiled to C comes as an extra. Unlike a programming language, FoF must avoid any ambiguity for its own semantics to be crystal clear. While this comes at the price of a more verbose language, we believe that FoF is reasonably concise: being embedded in Haskell, it benefits from the usual higher-order programming style, which reduces the amount of boilerplate code.

3.3 Support for formal verification

In our presentation of FoF, we have claimed that it fosters a new way of thinking about DSLs: a DSL abstracts away the problem it solves while, at the same time, formalizing this problem by the mean of its

semantics. However, as we have noticed, current DSLs fall short of the latter goal, as they lack formal semantics. In this section, we demonstrate how FoF solves this issue.

As described above, FoF is provided with an interpreter, evaluating FoF code into a FoF value. Because this interpreter is implemented in a purely functional style, it actually denotes a *pure semantics* of the FoF language. This approach follows the work of Swierstra [38,39] on the functional semantics of effects: whereas semantics are typically 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.

While close to the denotational framework, a functional semantics has several appealing properties. We have already mentioned the practical benefits: being able to *execute* the semantics means that we can experiment as well as extensively test it. The theoretical benefits of this approach have been explored in Swierstra’s Ph.D. dissertation [38]. First, he has shown that a proof of correctness can be carried out by equational reasoning, with relative ease. Second, he has demonstrated that these semantics can be described in a theorem prover, with great accuracy. We now discuss these points in more detail.

Functional programs have the reputation of being easy to reason about [19]. This ease has been successfully demonstrated by the exception compiler of Hutton et al. [20]: in this work, a small imperative language with exceptions and an interpreter for this language were implemented. This language is also compiled down to a stack machine. The compiler, the interpreter, and the stack machine are specified as pure Haskell functions. This work succeeds in proving the correctness of the compiler by equational reasoning.

Similarly, Bird [4] has shown how the functional approach can be fruitfully used to reason about pointer manipulation algorithms. In this paper, he first specifies pointers in a functional setting. Using this specification, he attacks the correctness proof of the Schorr-Waite algorithm, an algorithm extensively using pointers. In the tradition of the algebra of programming [17], he succeeds in *calculating* such a proof. Swierstra [39] has also used equational reasoning to prove the correctness of effectful functions, such as the `echo` command that reads characters and prints them back. All in all, these examples prove that functional specifications can be formally dealt with.

A second point that is worth highlighting is the importance of the *purity* of the specification. A pure function in Haskell exactly corresponds to a mathematical function, with a domain, a co-domain (decorated with \perp , the value of non-terminating programs), and a total mapping from the domain to the co-domain. Being this close to a mathematical formulation, functional specifications can be manipulated as mathematical objects. In particular, modulo the issue of non-termination, a formal mechanization of these specifications in a theorem prover is reachable with reasonable effort. This has been demonstrated by Swierstra in his Ph.D. dissertation [38]: he has described the semantics of Haskell mutable variables in the Agda dependently-typed programming language. We are convinced that these results could profit FoF.

Summary

In this section, we have motivated the design goals of FoF. We have emphasized the importance of FoF having a functional semantics, as a way to transitively provide a semantics to the DSLs. We have described the architecture of FoF, organized around a small, trustable core and extensible at will. The extreme modularity of FoF has been highlighted, allowing transparent definition of new and re-usable constructs, but also new post-processing functions.

4 Motivating Example: Fugu

So far, we have discussed the weaknesses of standard DSL designs and the strengths of FoF. In this section, we support our claims by describing the implementation of Fugu, an error definition language (EDL). This use-case is meant as a self-contained demonstration of FoF’s capabilities. For a complete, in-depth description of Fugu, we refer the reader to Fugu literate code [13]. In Section 4.1, we study the state of the art in error management. In Section 4.2, we describe our solution. Finally, we show some representative samples of the back-end, as implemented with FoF.

4.1 The state of the art

The need for an error definition system arises from the deficiencies of the traditional scheme, namely the `errno` model used in Posix systems. In this file, developers typically aggregate the error codes of all errors that could potentially occur in the system. In order to scale, they quickly “overload” the meaning of these error codes: instead of defining distinct error codes, they use a single code which has multiple causes. For example, the `EINVAL` error in FreeBSD vaguely signals that “*some* invalid argument was supplied: *for example*, specifying an *undefined* signal to a signal function *or* a kill system call”(our emphasis). In parallel, they also abuse the return values of functions. For instance, a function returning a `NULL` pointer is typically signaling an error. However, this does not provide much information to the developer.

Although this might have been historically relevant, there is no incentive today to limit the error code space to 255. Using 16 bits would allow us to define 65535 error codes. In such a huge space, we would be freed from the burden of overloading error codes and functions return values: when it is relevant, the developer should be able to define new error codes. Hence, she could handle errors more accurately, in the code as well as during the debugging process.

However, by imposing a flat name-space, the `errno` approach reduces the benefit of defining more precise error codes. When an error occurs, the developer is not able to relate it to a particular subsystem: it is simply defined as one error among thousands of others. In a sense, the error is more precise but still lacks a context. Being able to give a context to an error would be a step forward, notably during debugging.

Finally, it is often the case that an error in one function must be reported to several functions in the call-stack. Although the depth of the call-stack is generally small, it is extremely inconvenient to define case-specific error codes at each level – the number of cases growing exponentially with the depth. Hence, developers generally give up accuracy and report a global, overloaded error code.

4.2 Error definition language

Dealing with errors is of first importance in an operating system. Indeed, any real system is bound to face exceptional behavior: even if the software were perfect, hardware malfunction is common. Also, malicious users are likely to abuse the system to make it malfunction. In these circumstances, the operating system cannot give up: it is asked to handle errors, and it must also recover from these errors. Therefore, it needs accurate information on the cause of the failure. As we have seen in the previous section, error management is tedious work, made even more painful by the lack of good abstractions. As a result, this complexity is often a source of bugs and safety violations.

To alleviate these issues, we have designed Fugu, an error definition language. The developer is provided with a high-level language in which she describes error cases. A sample of Fugu code used in Barrelfish is shown in Figure 4.1, the meaning of which will become clear in the following.

To give a “spatial” context to errors, the developer can define classes of errors per *component*. The notion of component is purposely kept undefined: it is up to the developer to define their scope, on

```

errors retype_err {
  failure RETYPE_INVALID "Incorrect retype path",
  failure REVOKE_FIRST  "Retyping an already retyped cap",
  success RETYPE_OK      "Retyping success",
};

```

Figure 4.1: Sample of Fugu code

a case-by-case basis. For example, it could be a single function or a class of tightly-related functions. Figure 4.1 defines the class `retype_err`, which corresponds to an error occurring during the execution of the `caps_retype` function.

By encoding error codes with 16-bit values, we are able to remove the artificial limitation of the state-space. But we can also take advantage of this sub-word quantity in modern 64-bit machines. Indeed, in one machine word, we can aggregate four error codes. Hence, we can build a call-trace by *pushing* an error code in the previous one, and so on, along the erroneous call-path. Hence giving a “temporal” context to errors.

By relying on a tool to generate code for us, we can also automate a lot of boilerplate. For example, when defining an error code, we label it with a short acronym, *e.g.* `RETYPE_INVALID`, as well as a descriptive message, *e.g.* `"Incorrect retype path"`. Then, given an error code, the user is provided some functions to report it in a meaningful way.

Finally, in the event of a wild write – such as a buffer overflow – on a variable storing an error code, we would like the error code to become meaningless, instead of signaling an unrelated error. This is why we use random numbers to identify errors. Hence, the common consequences of a wild write, such as overwriting with zero or with an ASCII character, is detected more easily.

4.3 Fugu back-end

In the following, we study some samples of the Fugu back-end. While this is not meant as an exhaustive tutorial on FoF, we hope that it gives a convincing overview of the FoF programming style.

First of all, we define the data types manipulated throughout the back-end: an `err_code` is a 16-bit unsigned integer representing an individual error code. An `errval_t` is a 64-bit unsigned integer representing a stack of four `err_codes`. The respective types are suffixed by the letter `T`.

```

err_codeT :: TypeExpr
err_codeT = typedef uint16T "err_code"

```

```

errval_tT :: TypeExpr
errval_tT = typedef uint64T "errval_t"

```

Let us start with the `err_no` function that picks the first `err_code` out of an `errval_t` error stack. This consists in doing some standard bit masking to retrieve the 16 least-significant bits of the 64-bit value:

```

err_no_int :: [PureExpr] → FoFCode Def
err_no_int (err : []) =
  returnc (err .&. ((uint64 1 .<<. uint64 17) .-. uint64 1))

```

Then, we define the `err_no` function as static, in-lined, taking an `errval_t`, and returning an `err_code`:

```

err_noF :: (Def :<: f) ⇒ FoFCode f
err_noF = def [Static, Inline] "err_no" err_no_int err_codeT [errval_tT]

```

Similarly, we define `err_is_ok`, a predicate that takes an `errval_t`, extracts the error code by calling `err_no`, and checks whether this error code is an error or success. By design, a success code is an odd number, hence the parity test:

```

err_is_okF :: (Def :<: f) ⇒ FoFCode f
err_is_okF = def [Static, Inline]
  "err_is_ok" err_is_ok_int
  boolT [errval_tT]
  where err_is_ok_int :: [PureExpr] → FoFCode Def

```

```

err_is_ok_int (err : []) =
  do
    err_no ← err_noF
    err_no_e ← call err_no [err]
    returnc (err_no_e .%. uint64 2)

```

Whereas, so far, the functions were independent from the user input, `err_is_list` is representative of a function depending on the error definitions but also a typical example of higher-order programming with FoF. Indeed, `err_is_list` actually generates a *list* of functions, one per error code, that each tests whether the given argument corresponds to their respective error code. To this end, we have defined a higher-order FoF function `err_is` that is mapped onto the list of error codes:

```

err_is_list :: (Enum => f, Def => f) =>
  Enumeration -> [String] -> [FoFCode f]
err_is_list enum codes = map err_is codes
  where err_is name =
    def [Static, Inline]
      ("err_is_" ++ map toLower name) err_is_f
    boolT [errval_tT]
    where err_is_f :: [PureExpr] ->
      FoFCode (Def :+> Enume)
      err_is_f (err : []) = do
        err_no ← err_noF
        err_no_e ← call err_no [err]
        code ← ofEnum enum name
        returnc (err_no_e .==. code)

```

A more complex function is `err_print_calltrace` that unwinds an error stack and print a descriptive error message at each step. Its implementation is presented in Figure 4.2. It is presented here for the mere purpose of giving a taste of more advanced FoF programming. The curious reader is referred to the Fugu literate code for more details.

Last but not least, the Fugu back-end is implemented by the `backend` function, shown in Figure 4.3. To give an idea of its simplicity, it is shown here as-is in the actual code. Note the type of this function: the back-end function transforms an `Errors` data type, as returned by the front-end, and builds a `FoFCode` value. The `FoFCode` value can be further compiled or interpreted. Again, we refer the reader to the literate code for a more comprehensive study. Nonetheless, we note the declaration of `errval_tT` and `err_codeT` as aliases. We also note the definitions of `err_no` and `err_is_ok`. The use of `err_is_list` is enclosed in a call to `sequenceSem` that folds the compilation of each individual `err_is` function definition. Finally, we recognize the definition of `err_print_calltrace`, taking some of the preceding functions in argument.

Summary

Thanks to this self-contained example, we hope to have demonstrated some of the strengths of FoF. With `err_no`, we have shown that Fugu is able to handle bit twiddling code with exactly the same ease as C. This is justified by the fact that these operations are part of the pure core of C, hence are directly translatable in FoF. We have also demonstrated the benefit of higher-order programming for automatic code generation with `err_is_list`. This programming style alleviates the verbosity of FoF without impeding type-safety, *i.e.* the guarantee of generating correct C code. With `err_is_ok` and to a greater-extent with `err_print_calltrace`, we prove that FoF can capture the semantics of intricate code, such as function calls, conditionals, loops, and mutable variables.

Beyond being a simple proof-of-concept, Fugu is used extensively in Barrelfish, including in the kernel. Because the kernel is part of the trusted computing base, Fugu-generated code had to satisfy the high-standard ordinarily required for any code in the safety-critical path. Obviously, it has to be safe, that is, at least, trustworthy. But it also needs to be maintainable in the long term, and should not slow the development process. Finally, it must be reasonably efficient, as the kernel cannot suffer any performance penalty. All in all, Fugu meets these requirements and has already been integrated into the main development tree.

```

err_print_calltraceF :: (Conditionals <: f, Def <: f, Ref <: f, Printf <: f) =>
    PureExpr → PureExpr → PureExpr →
    FoFCode f
err_print_calltraceF err_getdomain err_getstring err_getcode =
    def [Static, Inline]
        "err_print_calltrace" err_print_calltrace_int
        voidT [errval_tT]
        where err_print_calltrace_int :: [PureExpr] → FoFCode (Ref :+:
            Def :+:
            Conditionals :+:
            Printf)

err_print_calltrace_int (err : []) =
    do
        err_no ← err_noF
        err_is_fail ← err_is_failF

        err_ref ← newRef err

    ifc (do
        is_fail_e ← call err_is_fail [err]
        return is_fail_e
            :: FoFCode (Ref :+: Def))
    (do
        x ← newRef $ uint16 0

        while (do
            err_ref_val ← readRef err_ref
            err_ref_no ← call err_no [err_ref_val]
            writeRef x err_ref_no
            return $ err_ref_no .!=. (uint16 0)
                :: FoFCode (Ref :+: Def))
        (do
            err_ref_val ← readRef err_ref
            err_ref_dom ← call err_getdomain [err_ref_val]
            err_ref_str ← call err_getstring [err_ref_val]
            err_ref_code ← call err_getcode [err_ref_val]
            printf "Failure: (%15s) %20s [%10s]\n"
                [err_ref_dom, err_ref_str, err_ref_code]
            writeRef err_ref (err_ref_val >>. (uint64 16))
                :: FoFCode (Ref :+: Def :+: Printf))
        :: FoFCode (Ref :+: Def :+: Conditionals :+: Printf))
    (do return $ uint64 0
        :: FoFCode Def)

```

Figure 4.2: Fugu back-end: err_print_calltrace

```

backend :: Errors → FoFCode (EnumE :+: Assert :+: Ref :+:
                             Def :+: StaticArray :+: Conditionals :+:
                             Typedef :+: Printf)
backend errors =
  do
    (err_domains, err_codes, err_descs) ← err_arrays errors
    alias errval_tT
    alias err_codeT

    aliasE "<stdlib.h>" charT
    aliasE "<stdbool.h>" boolT

    newEnum "err_code" enumErrCodes

    err_no ← err_noF
    err_push ← err_pushF
    err_is_ok ← err_is_okF
    err_is_fail ← err_is_failF
    err_assert_ok ← err_assert_okF
    err_assert_fail ← err_assert_failF

    err_is_tests ← sequenceSem $ err_is_list enumErrCodes errorCodes

    err_getstring ← err_get_array "string" err_descs mapIdDescription (err_code noCodes)
    err_getcode ← err_get_array "code" err_codes mapIdAcronym (err_code noCodes)
    err_getdomain ← err_get_array "domain" err_domains mapIdDomain (err_code noDomains)

    err_print_calltrace ← err_print_calltraceF err_getdomain err_getstring err_getcode

    return $ uint64 0
      where ...

```

Figure 4.3: Fugu back-end

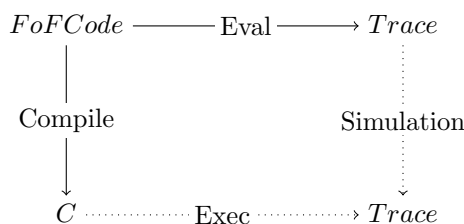
5 Filet-o-Fish Compiler Correctness

In the previous sections, we have claimed that FoF relieves the DSL designer from thinking about tangled strings of C code. Instead, FoF proposes a safe abstraction of C described by a functional semantics. Therefore, the developer can design and reason about her compiler at a high-level. Then, the FoF compiler translates FoF code to C. However, reasoning at FoF level is valid if and only if the FoF compiler is correct, *i.e.* the semantics of the FoF code is preserved by the translation to C. In this section, we prove the correctness of the FoF compiler, for a subset of the language. In Section 5.1, we state more precisely the problem we are to solve. In Section 5.2, we develop a dynamic semantics for FoF. In Section 5.3, we specify a compiler from FoF to C. In Section 5.4, we present the correctness proof, showing that the FoF compiler indeed preserves FoF’s dynamic semantics after compilation to C. Finally, we draw some conclusions from this work and mention some opportunities for future work.

5.1 Problem statement

In a nutshell, a compiler correctness proof consists in showing that the semantics of the source language is preserved while compiled to the target language. Therefore, to be able to perform this proof, we need to be provided the semantics of the source language as well as the semantics of the target language. Obviously, we also need a formal specification of the compiler. In our case, the proof will consist of showing that the FoF semantics is preserved when FoF is compiled to C.

The standard proof technique to show this is by *simulation*. Restated, this involves showing that the following diagram commutes:



That is, the trace of observable events produced along the evaluation of the FoF code is *similar* to the trace produced by the execution of the C code generated by the compiler applied to this input. Observed events correspond to external function calls – system calls, for example – and to external memory accesses – such as accessing an externally-defined variable. Then, “similar” informally means that both FoF and C code execute exactly identical external function calls and memory accesses, *i.e.* the same arguments and results. We formalize this notion when we introduce the proof mechanics. In the following, we only consider the subset of FoF described in Figure 5.1. It is a stripped-down version of FoF, with the *do* operator for sequencing statements and the reference cell operators. In C, this maps to intra-procedural code manipulating variables and pointers.

```

stmt ::= do stmt stmt
      | r ← newRef expr
      | r ← readRef expr
      | writeRef expr expr
  
```

Figure 5.1: Formally-verified subset of Filet-o-Fish

$$\begin{array}{c}
\frac{E(id) = b \text{ or } (id \notin \text{dom}(E)) \text{ and } \text{symbol}(G, id) = [b]}{G, E \vdash id, M \stackrel{\Leftarrow}{\simeq} (b, 0)} \text{Clight (1)} \\
\\
\frac{G, E \vdash a, M \stackrel{\Leftarrow}{\simeq} \text{ptr}(l)}{G, E \vdash *a, M \stackrel{\Leftarrow}{\simeq} l} \text{Clight (2)} \\
\\
\frac{}{G, E \vdash n, M \stackrel{\Leftarrow}{\simeq} \text{int}(n)} \text{Clight (5)} \\
\\
\frac{}{G, E \vdash f, M \stackrel{\Leftarrow}{\simeq} \text{float}(f)} \text{Clight (6)} \\
\\
\frac{G, E \vdash a, M \stackrel{\Leftarrow}{\simeq} l \quad \text{loadval}(\text{type}(a), M, l) = [v]}{G, E \vdash a, M \stackrel{\Leftarrow}{\simeq} v} \text{Clight (8)} \\
\\
\frac{G, E \vdash a, M \stackrel{\Leftarrow}{\simeq} l}{G, E \vdash \&a, M \stackrel{\Leftarrow}{\simeq} \text{ptr}(l)} \text{Clight (9)} \\
\\
\frac{G, E \vdash a_1, M \stackrel{\Leftarrow}{\simeq} l \quad G, E \vdash a_2, M \stackrel{\Leftarrow}{\simeq} v \quad \text{storeval}(\text{type}(a_1), M, l, v) = [M']}{G, E \vdash (a_1 = a_2), M \stackrel{\Leftarrow}{\simeq} \text{Normal}, M'} \text{Clight (20)}
\end{array}$$

Figure 5.2: Subset of Clight execution rules

As mentioned above, our target language, here C, needs to be described by a semantics. However, the semantics of C is very complex, if it exists at all. Moreover, in our case, we are not willing to give up the possibility of performing the correctness proof in a theorem prover. Hence, the simpler the C semantics is, the better. Thanks to Leroy’s recent work on certified C compilers [6, 24], a mechanized memory model and semantics of a subset of C has been developed and entirely described in the Coq theorem prover [7, 25]. This subset of C, called Clight, is actually expressive enough for our purposes. Clight is described by a coinductive big-step operational semantics [26]. Therefore, it benefits from the ease of reasoning offered by big-step semantics, while still being able to reason about non-terminating programs. We refer the reader to the aforementioned paper for more details about the benefits of this semantic framework.

Similarly, we refer the reader to the Clight paper [7] for the complete formalization of the language. For this report to be self-contained, we briefly outline the derivation rules used in our proof. The derivation rules are presented in Figure 5.2. As is usual in C, there is a distinction between l-value and r-value, respectively evaluated by $\stackrel{\Leftarrow}{\simeq}$ and $\stackrel{\Leftarrow}{\simeq}$. An l-value refers to a memory location, and thus can appear in the left-hand side of the assignment symbol, “=”. Conversely, an r-value is a term appearing on the right-hand side of the assignment sign. Note that an l-value might also appear on the right-hand side, but its semantics is not the same, as shown by rule (8).

We rely on two rules for evaluating l-values: rule (1) corresponds to evaluating an identifier, either local or global, to its location. Rule (2) corresponds to dereferencing an address in memory. Concerning r-values, we use four rules. Rules (5) and (6) trivially evaluate an integer or float value to itself. Rule (8) evaluates a location by recursively evaluating the corresponding l-value, and retrieving the value from memory. Rule (9), on the other hand, evaluates the reference of a variable to its actual address.

Finally, rule (20) handles the evaluation of an assignment. This consists in evaluating the l-value a_1 to an address, evaluating the r-value a_2 to an actual value, and updating the memory state accordingly. We note the presence of the *Normal* keyword: when evaluating a statement, several *outcomes* could happen that influence the control-flow. If the C statement is a `return(v)` instruction, the outcome is *Return(v)*, which leads to all statements being skipped until the end of the function. Similarly, a `return` instruction results in a *Return* outcome. Concerning loops, `break` and `continue` instructions respectively result in

Break and *Continue* outcomes, with obvious effects on the evaluation. A *Normal* outcome, on the other hand, does not influence the control-flow.

In our setting, `loadval(.)` and `storeval(.)` always reduce to `load(.)` and `store(.)`. `load(κ, M, b, m)` reads at offset n in block b of the memory state M , over a length determined by the size of elements of type κ . `store(κ, M, b, n, v)` updates the memory state M by storing the value v at block b , offset n , over a length determined by the size of elements of type κ .

Using these rules and their intuitive understanding, we will be able to prove the correctness of FoF compiler. But first, we need to describe the semantics of FoF.

5.2 Filet-o-Fish dynamic semantics

In the previous sections, we have highlighted the importance of FoF having a functional semantics. Indeed, this is a perfect support for mechanized testing and reasoning. However, the compiler correctness proof is easier to handle if both source and target languages are described in a fairly close semantic framework. As we have seen above, Clight is described in a big-step operational semantics, which is quite far from the almost denotational functional semantics. Therefore, we face a difficult choice: either we describe both languages in the functional framework, or we describe them in the operational framework. In the former scenario, we would have to translate Clight to the functional framework, with the benefit of carrying out the proof by equational reasoning. In the latter scenario, we had to translate FoF's semantics to an operational semantics, and then use structural induction on the derivation trees to meet our goals. Because FoF is much less complex than Clight, we chose the latter approach, hence reducing the risk of error. In the last part of this section, we further motivate this choice. The dynamic semantics we develop covers the language presented in Figure 5.1.

5.2.1 Evaluating pure expressions

We start with the semantics of pure expressions. Following Clight, we describe it in the denotational framework, with an evaluation function $\mathbb{E}[\cdot]$. Because pure expressions always converge to a value, the evaluation function is straightforward: values, such as integer, float, and reference, are returned as such. Composed expressions are simply interpreted in the meta-language: for example, if op_1 is the “+” symbol, it is interpreted as the addition.

$$\begin{aligned}
\mathbb{E}[\textit{int } n \textit{ intsize } \textit{signedness}] &= \textit{int } n \textit{ intsize } \textit{signedness} \\
\mathbb{E}[\textit{float } f] &= \textit{float } f \\
\mathbb{E}[\textit{ref } \textit{type } \textit{id}] &= \textit{ref } \textit{type } \textit{id} \\
\mathbb{E}[op_1 e] &= [op_1] \mathbb{E}[e] \\
\mathbb{E}[e_1 op_2 e_2] &= \mathbb{E}[e_1] [op_2] \mathbb{E}[e_2] \\
\mathbb{E}[\textit{sizeof}(type)] &= [\textit{sizeof}(type)] \\
\mathbb{E}[(type)e] &= e && \text{if } \textit{validCast}(type, e) \\
\mathbb{E}[e_1 ? e_2 : e_3] &= \mathbb{E}[e_2] && \text{if } \mathbb{E}[e_1] = 0 \\
\mathbb{E}[e_1 ? e_2 : e_3] &= \mathbb{E}[e_3] && \text{if } \mathbb{E}[e_1] \neq 0
\end{aligned}$$

5.2.2 Evaluating statements

As for statements, they are described in the coinductive big-step operational semantics, following Clight. In particular, we record observable behavior in traces: external function calls and memory accesses are listed in the order they happened, in a potentially infinite list of events. An evaluation step has the following shape:

$$mem, heap, s \Rightarrow out, mem', heap' \mid t$$

Meaning that, starting from memory state mem , with the variable declared in $heap$, the evaluation of statement s results in an outcome out , updated memory state and heap, and produces a trace t . This trace might be infinite if the program does not terminate, *i.e.* diverges. A memory state is a partial

$$\begin{array}{c}
\frac{mem, heap, s_0 \Rightarrow Normal, mem', heap' \mid t_0 \quad mem', heap', s_1 \Rightarrow out, mem'', heap'' \mid t_1}{mem, heap, do\ s_0\ s_1 \Rightarrow out, mem'', heap'' \mid t_0.t_1} \text{ (eval-do-cvg-normal)} \\
\\
\frac{mem, heap, s_0 \Rightarrow out, mem', heap' \mid t_0 \quad skipNext(out)}{mem, heap, do\ s_0\ s_1 \Rightarrow out, mem', heap' \mid t_0} \text{ (eval-do-cvg-skip)} \\
\\
\frac{mem, heap, s_0 \overset{\infty}{\Rightarrow} \mid T}{mem, heap, do\ s_0\ s_1 \overset{\infty}{\Rightarrow} \mid T} \text{ (eval-do-dvg-1)} \\
\\
\frac{mem, heap, s_0 \Rightarrow Normal, mem', heap' \mid t_0 \quad mem', heap', s_1 \overset{\infty}{\Rightarrow} \mid T_1 \quad T \cong t_0.T_1}{mem, heap, do\ s_0\ s_1 \overset{\infty}{\Rightarrow} \mid T} \text{ (eval-do-dvg-2)} \\
\\
\frac{v = \mathbb{E}[e] \quad tv = \text{typeof}(v)}{\left(\begin{array}{l} mem, \\ heap, \\ r \leftarrow newRef\ e \end{array} \right) \Rightarrow \left(\begin{array}{l} Normal, \\ mem[r \mapsto v], \\ heap[r \mapsto pointerT\ tv\ Available] \end{array} \right) \mid \epsilon} \text{ (eval-newRef-anon)} \\
\\
\frac{\mathbb{E}[r_2] = ref\ type\ l \quad tr_2 = \text{unfoldPointerType}(type) \quad l \in dom(mem)}{\left(\begin{array}{l} mem, \\ heap, \\ r_1 \leftarrow readRef\ r_2 \end{array} \right) \Rightarrow \left(\begin{array}{l} Normal, \\ mem[r_1 \mapsto mem(l)], \\ heap[r_1 \mapsto tr_2] \end{array} \right) \mid \epsilon} \text{ (eval-readRef-anon)} \\
\\
\frac{\mathbb{E}[r] = ref\ type\ l \quad \mathbb{E}[e] = v \quad l \in dom(mem) \quad \text{unfoldPointerType}(type) = \text{typeof}(v)}{\left(\begin{array}{l} mem, \\ heap, \\ writeRef\ r\ e \end{array} \right) \Rightarrow \left(\begin{array}{l} Normal, \\ mem[l \mapsto v], \\ heap \end{array} \right) \mid \epsilon} \text{ (eval-writeRef-anon)}
\end{array}$$

Figure 5.3: FoF dynamic semantics

mapping from variable names to FoF values, *i.e.* $mem : id \mapsto v$. A heap is a mapping from variable names to their types, *i.e.* $heap : id \mapsto type$.

Evaluation of the sequencing operator:

Evaluating a sequence of statements is almost straightforward: we evaluate the first term s_0 starting from the initial state, then evaluate the second term s_1 with the updated state. However, there are two subtleties: first, whether terms converge or diverge, and second whether the outcome is *Normal* or not. When the outcome of the first term is normal and both computations converge, this is the standard case as shown in Figure 5.3 by rule (eval-do-cvg-normal).

When the first statement converges prematurely, *i.e.* with an outcome asking to skip the following statements, then we do not need to evaluate the next statement and only report the evaluation result of the first term (rule (eval-do-cvg-skip)). The definition of skipNext corresponds to the intuitive notion of “prematurely”:

$$\begin{aligned} \text{skipNext}(x) = \text{true} \quad \Leftrightarrow \quad & x = \text{Continue} \\ & \vee x = \text{Break} \\ & \vee x = \text{Return} \\ & \vee x = \text{Return}(v) \end{aligned}$$

Then come the diverging cases. In case the first term diverges, we never evaluate the second and only report the infinite trace produced by this term (rule (eval-do-dvg-1)).

When the first term converges to a *Normal* outcome but the second term diverges, we then report a trace T bisimilar to a trace containing the finite trace of the first term followed by the infinite trace produced by the second term (rule (eval-do-dvg-2)). Note that T is not required to be *exactly* the concatenation of t_0 and T_1 : equality of coinductive structures is much stronger than bisimilarity in, for example, Coq constructive logic, therefore making proofs more complex and sometimes impossible. As we do not require this extra power, we stick to the simpler notion of bisimilarity of streams.

Evaluation of reference operators:

As for the evaluation of reference operators, we observe various interesting patterns. Let us look at these operators in turn.

As one would expect, the evaluation of *newRef* always terminates and its outcome is *Normal*. Its side effect is to register the new variable r in the *heap* and to map this variable to the actual value of e , as shown in Figure 5.3 rule (eval-newRef-anon). Similarly to the *type(.)* function of Clight, mapping Clight expressions to their type, we define a *typeof(.)* function, mapping FoF expressions to their FoF type. Note that ill-typed terms, such as applying a binary operation on two terms of distinct types, are not handled. In this case, the evaluator is blocked, modelling an erroneous input. The *validCast(.)* operation simply verifies that the term e can indeed be safely cast to type t ; it is undefined otherwise. As a result, we obtain the following definition of *typeof(.)*:

$$\begin{aligned} \text{typeof}(\text{int } n \text{ intsize signedness}) &= \text{intT intsize signedness} \\ \text{typeof}(\text{float } f) &= \text{floatT} \\ \text{typeof}(\text{ref type id}) &= \text{type} \\ \text{typeof}(\text{op}_1 e) &= \text{typeof}(e) \\ \text{typeof}(e_1 \text{ op}_2 e_2) &= t && \text{if } \text{typeof}(e_1) = \text{typeof}(e_2) = t \\ \text{typeof}(\text{sizeof}(type)) &= \text{intT I64 unsigned} \\ \text{typeof}((type)e) &= \text{type} && \text{if } \text{validCast}(type, e) \\ \text{typeof}(e_1 ? e_2 : e_3) &= t && \text{if } \text{typeof}(e_2) = \text{typeof}(e_3) = t \end{aligned}$$

Like *newRef*, *readRef* always terminates with a *Normal* outcome. Reading a reference results in the definition of a new local variable r_1 mapped to the content of the reference cell r_2 (rule (eval-readRef-anon))

The type of r_1 , as registered in the *heap*, needs to be computed from the type of r_2 . Because r_2 is a reference cell, its type must be a *pointerT*. Then, two cases can occur. The base case corresponds to a

reference cell containing a base type, such as an integer or a float. Thus, the type of the read value is precisely this base type. This corresponds to the second definition below. The complex case corresponds to a reference cell itself enclosing a reference cell, of type $pointerT\ type\ m'$. When reading out a pointer of a cell, we record that this pointer has been read. While this does not have any importance for the evaluator as such, it simplifies the correctness proof. Hence, the following definition of `unfoldPointerType(.)`:

$$\begin{aligned} \text{unfoldPointerType}(pointerT\ (pointerT\ type\ m')\ m) &= pointerT\ type\ Read \\ \text{unfoldPointerType}(pointerT\ type\ m) &= type \end{aligned}$$

Finally, the evaluation of `writeRef` comes without surprise: it converges with a *Normal* outcome. Its effect is to re-map r to the value of e (rule (eval-writeRef-anon)).

At this point, we have completely described the semantics of our subset of FoF. The last step before carrying out the proof is to specify the compiler. This is the subject of the following section.

5.3 Filet-o-Fish compiler specification

In the following, we specify the FoF-to-C compiler. We start with the compilation of pure FoF expressions and FoF types. Then, we describe the compilation of statements.

5.3.1 Compilation of pure expressions and types

Similar to the evaluation function, the pure expressions of FoF are dealt with in a simple and convenient way: values are translated to their direct Clight equivalent whereas operators are simply quoted to the corresponding C operator. A subtlety arises in the compilation of references. As mentioned above, a reference is initialized in an *Available* mode: the stored value is available *in* the variable var , hence var denotes the reference cell. Therefore, we compile the cell to its identifier. However, a cell read from an enclosing cell is in a *Read* mode: var actually *points to* the variable that denotes the reference cell. Therefore, we compile the cell to the dereferenced identifier.¹

$$\begin{aligned} \text{toC}_e(int\ n\ sz\ sg) &= n \\ \text{toC}_e(float\ f) &= f \\ \text{toC}_e(ref\ (pointerT\ type\ Available)\ var) &= var \\ \text{toC}_e(ref\ (pointerT\ type\ Read)\ var) &= *var \\ \text{toC}_e(ref\ type\ var) &= var \\ \text{toC}_e(op_1\ e) &= 'op'_1\ \text{toC}_e(e) \\ \text{toC}_e(e_1\ op_2\ e_2) &= \text{toC}_e(e_1)\ 'op'_2\ \text{toC}_e(e_2) \\ \text{toC}_e(sizeof(e)) &= 'sizeof'\ (\text{toC}_e(e)) \\ \text{toC}_e((type)\ e) &= '('\ \text{toC}_t(type)\ ')'\ \text{toC}_e(e) \\ \text{toC}_e(e_1\ ?\ e_2\ :\ e_3) &= \text{toC}_e(e_1)\ '?'\ \text{toC}_e(e_2)\ ':\ '\ \text{toC}_e(e_3) \end{aligned}$$

Similarly, we compile FoF types to C types by direct translation. As we are not using them in our restricted language, we do not describe the compilation of complex types, such as arrays, structures, and unions.

$$\begin{aligned} \text{toC}_t(voidT) &= void \\ \text{toC}_t(intT\ intsize\ signedness) &= int\ intsize\ signedness \\ \text{toC}_t(floatT) &= float \\ \text{toC}_t(pointerT\ type\ mode) &= pointer(\text{toC}_t(type)) \end{aligned}$$

5.3.2 Compilation of statements

For the sake of clarity, the statement compiler is specified in an operational setting. A compiler step is described by a relation of the following shape:

¹This apparent complexity comes from the fact that FoF's reference cells abstract C variables *and* C pointers at the same time. Hence the extra machinery in the compiler, to handle and differentiate both cases. Historically, this aspect of FoF has been the most difficult to get right.

$$\begin{array}{c}
\frac{\left(\begin{array}{c} \textit{binding}, \\ s_0 \end{array} \right) \xrightarrow{c} \left(\begin{array}{c} \textit{code}', \\ \textit{binding}' \end{array} \right) \quad \left(\begin{array}{c} \textit{binding}', \\ s_1 \end{array} \right) \xrightarrow{c} \left(\begin{array}{c} \textit{code}'', \\ \textit{binding}'' \end{array} \right)}{\left(\begin{array}{c} \textit{binding}, \\ \textit{do } s_0 \textit{ } s_1 \end{array} \right) \xrightarrow{c} \left(\begin{array}{c} \textit{code} + \textit{code}', \\ \textit{binding}'' \end{array} \right)} \text{(compile-do)} \\
\\
\frac{\begin{array}{c} dc_e = \text{deref}(t_e, c_e) \\ ct_e = \text{toC}_t(t_e) \\ t_e = \text{typeof}(e) \quad c_e = \text{toC}_e(e) \end{array}}{\left(\begin{array}{c} \textit{binding}, \\ r \leftarrow \textit{newRef } e \end{array} \right) \xrightarrow{c} \left(\begin{array}{c} \textit{binding}[r \mapsto \textit{pointerT } t_e \textit{ Available}], \\ \left[\begin{array}{l} \textit{local: } ct_e \textit{ } r \\ \textit{stmts: } r = dc_e \end{array} \right] \end{array} \right)} \text{(compile-newRef)} \\
\\
\frac{\begin{array}{c} ct_{r_1} = \text{toC}_t(t_{r_1}) \\ c_{r_2} = \text{toC}_e(r_2) \quad t_{r_1} = \text{unfoldPointerType}(r_2) \end{array}}{\left(\begin{array}{c} \textit{binding}, \\ r_1 \leftarrow \textit{readRef } r_2 \end{array} \right) \xrightarrow{c} \left(\begin{array}{c} \textit{binding}[r_1 \mapsto t_{r_1}], \\ \left[\begin{array}{l} \textit{local: } ct_{r_1} \textit{ } r_1 \\ \textit{stmts: } r_1 = c_{r_2} \end{array} \right] \end{array} \right)} \text{(compile-readRef)} \\
\\
\frac{\begin{array}{c} dc_e = \text{deref}(t_e, c_e) \\ c_e = \text{toC}_e(e) \\ c_r = \text{toC}_e(r) \quad t_e = \textit{binding}(e) \quad \text{liftType}(\text{unfoldPointerType}(r)) = \text{typeof}(e) \end{array}}{\left(\begin{array}{c} \textit{binding}, \\ \textit{writeRef } r \textit{ } e \end{array} \right) \xrightarrow{c} \left(\begin{array}{c} \textit{binding}, \\ \left[\textit{stmts: } c_r = dc_e \right] \end{array} \right)} \text{(compile-writeRef)}
\end{array}$$

Figure 5.4: FoF-to-Clight compiler specification

$$\left(\begin{array}{c} \text{binding}, \\ s \end{array} \right) \xrightarrow{c} \left(\begin{array}{c} \text{code}, \\ \text{binding}' \end{array} \right)$$

Hence, the compiler state is carried by *binding*, which maps variable names to their type: *binding* : *id* \mapsto *type*. Compiling a statement *s* results in some *code* and an updated compiler state. The *code* generated by the compiler is composed of a triple (*global* \times *local* \times *stmts*) of global declarations, *i.e.* global variables, local declarations, *i.e.* intra-procedural variables, and statements, *i.e.* intraprocedural code.

Compilation of the sequencing operator:

As shown in Figure 5.4 by rule (compile-do), compiling sequential code is trivial: we compile the first statement, and compile the second based on the updated compiler state. The code generated by both statements is concatenated as one would expect: variable declarations are concatenated in the order of definition, and statements are presented in the order they have been generated.

Compilation of the reference operators:

The compilation of *newRef* is more involved, as shown by rule (compile-newRef). Let us consider the creation of a reference cell *r* initialized to a FoF value *e*. First of all, we need to compute the type *t_e* of *e*, to be able to type the reference cell. Because we initialized the reference cell to *e*, we need to compile the FoF value *e* to its corresponding C value *dc_e*. Compiling this statement then consists in declaring a variable *r* and initializing it to *dc_e*.

Note the use of *deref*(*typeof*(*e*), *toC_e*(*e*)) to compile the FoF value to a C value. Its definition is the following:

$$\begin{aligned} \text{deref}(\text{pointerT } t \ m, c_e) &= \&c_e \\ \text{deref}(\text{otherwise}, c_e) &= c_e \end{aligned}$$

The meaning is clear: the C value corresponding to a reference cell *c_e* (identified by a *pointerT* type) is the *address* of the variable that stores the cell's content. Otherwise, *i.e.* for integer and float values, the FoF value directly maps to the C value, hence the direct translation.

Overall, the compilation of *readRef* is similar (rule (compile-readRef)). We first compute the FoF reference *r₂* to a C value and compute the type of the read data based on the unfolded type of *r₂*. Compiling this statement simply involves defining a new variable and setting it to the content of the read reference cell.

The compilation of *writeRef* is similar (rule (compile-writeRef)). First, it consists of compiling the FoF reference cell *r* to C, in *c_r*; it also retrieves the type of *e* from the *binding*, and compiles the C value *dc_e* from the FoF value *e*. The generated code consists in updating *c_r* with *dc_e*. However, we need to sanity-check that the type of the written value corresponds to the type of the value already contained in the reference cell. To check types for equality, we need them to be in normal form. However, by definition of the *pointerT* type, *pointerT type Read* is equivalent to *type*. Hence, we must first lift types to normal form, by simplifying *Read* pointers, before comparing them. Note that we do not need to recurse inside the *pointerT* definition: by construction, the *type* inside a *pointerT* is always available, either a base value or an *Available pointerT*.

$$\begin{aligned} \text{liftType}(\text{pointerT } \text{type } \text{Read}) &= \text{type} \\ \text{liftType}(\text{type}) &= \text{type} \end{aligned}$$

5.4 Correctness proof

At this point of our presentation, we have enough material to get started with the proof. This proof is by structural induction on the derivation tree of the evaluation semantics. Hence, we consider a FoF statement s and assume that the following relations hold for suitably defined equivalence relations \sim_{heap} and \sim_{mem} :

1. $\text{heap} \sim_{\text{heap}} \text{binding}$
2. $M \sim_{\text{mem}} \text{mem}$

We aim to prove that if we can take an evaluation step on s :

$$\text{heap}, \text{mem}, s \Rightarrow \text{out}, \text{heap}', \text{mem}'$$

... and we can compile this statement to:

$$\text{binding}, s \xrightarrow{c} \text{code}, \text{binding}'$$

... and we can further execute the compiled code to:

$$\text{code.global}, \text{code.local} \vdash \text{code.stmt}, M \xrightarrow{C} \text{out}', M'$$

... then the following relations hold:

$$\begin{array}{ccc} \text{heap}' & \sim_{\text{heap}} & \text{binding}' \\ M' & \sim_{\text{mem}} & \text{mem}' \\ \text{out} & = & \text{out}' \end{array}$$

... meaning that if the evaluation and the corresponding compilation/execution steps converge, then the \sim_{mem} relation between FoF memory and the machine memory is preserved. This proves that, for terminating executions, the FoF semantics is respected by the code executed by the machine, provided a correct C compiler.

Because the subset of FoF we are studying always converges, we are not going to study the diverging cases involving traces. Therefore, we ignore traces. However, this machinery would be necessary as soon as we extend the considered language with loops. In this simplified scheme, the simulation proof amounts to show that the following diagram commutes:

$$\begin{array}{ccc} \left\{ \begin{array}{l} \text{heap}, \text{mem}, s \\ \text{binding}, s \end{array} \right. & \xrightarrow{\text{Eval}} & \text{heap}', \text{out}, \text{mem}' \\ \downarrow \text{Compile} & & \downarrow \sim_{\text{heap}} \times \text{=} \times \sim_{\text{mem}} \\ \left\{ \begin{array}{l} \text{code}, \text{binding}' \\ \text{code.global}, \text{code.local} \vdash \text{code.stmts}, M \end{array} \right. & \xrightarrow{\text{Clight Exec}} & \left\{ \begin{array}{l} \text{binding}' \\ \text{out}', M' \end{array} \right. \end{array}$$

Let us define the equivalence relations stated above, namely \sim_{heap} and \sim_{mem} . First, $\text{heap} \sim_{\text{heap}} \text{binding}$ states that a *heap*, i.e. a map $id \mapsto \text{type}$, is equal to *binding*, i.e. a map $id \mapsto \text{type}$. Formally, this translates to standard function equality:

$$\begin{aligned} \text{heap} \sim_{\text{heap}} \text{binding} &\Leftrightarrow \wedge \forall x \in \text{dom}(\text{heap}), \wedge x \in \text{dom}(\text{binding}) \\ &\quad \wedge \text{heap}(x) = \text{binding}(x) \\ &\wedge \forall x \in \text{dom}(\text{binding}), \wedge x \in \text{dom}(\text{heap}) \\ &\quad \wedge \text{heap}(x) = \text{binding}(x) \end{aligned}$$

We next define $b \sim_{\text{mem}}(M, \text{mem}) r$, with b a Clight memory block and r a FoF reference value. b and r are equivalent if and only if they map to the same term, that is the same value if they map to an integer or

float term, or to memory-equivalent terms if they map to a block/reference. Formally, this corresponds to:

$$\begin{aligned}
b \sim_{\text{mem}(M, \text{mem})} r \Leftrightarrow & 1. \wedge v_b = M(b)(0) \\
& 2. \wedge v_r = \text{mem}(r) \\
& 3. \wedge t = \text{type}(v_b) = \text{toC}_t(\text{typeof}(v_r)) \\
& 4. \wedge \text{a.} \vee \wedge t \in \{\text{int}, \text{float}\} \\
& \quad \wedge v_b = \text{toC}_e(v_r) \\
& \text{b.} \vee \wedge t = \text{pointer} \\
& \quad \wedge M(v_b) \sim_{\text{mem}(M, \text{mem})} \text{mem}(v_r)
\end{aligned}$$

Then, we generalize the $\sim_{\text{mem}(M, \text{mem})}$ relation to the complete C and FoF memory: $M \sim_{\text{mem}} \text{mem}$ if and only if both map from equivalent blocks/references to equal values or equivalent blocks/references. Hence, the following definition:

$$\begin{aligned}
M \sim_{\text{mem}} \text{mem} \Leftrightarrow & \text{a.} \vee M \text{ and } \text{mem} \text{ are empty} \\
& \text{b.} \vee \exists M', \text{mem}', b, r, x, v. 1. \wedge M' \sim_{\text{mem}} \text{mem}' \\
& \quad 2. \wedge b \sim_{\text{mem}(M', \text{mem}')} r \\
& \quad 3. \wedge x \sim_{\text{mem}(M', \text{mem}')} v \\
& \quad 4. \wedge M = M'[b \mapsto (0, \text{sizeof}(b), 0 \mapsto x)] \\
& \quad 5. \wedge \text{mem} = \text{mem}'[r \mapsto v]
\end{aligned}$$

Armed with these definitions, we are now able to prove the correctness of the compiler. Because of the length and complexity of the proof, we will not get into the details here. The reader is referred to Section A for the complete, in-depth proof. In the following, we simply sketch the organization of the proof along with the key techniques employed. For the sake of clarity, our proofs are written in a hierarchical format [23].

PROOF SKETCH: We proceed by structural induction on the derivation tree. Hence, we do a case analysis on the last term of the tree.

\langle 1 \rangle 1. CASE: $s = \text{do } s_0 \ s_1$

PROOF SKETCH: By applying the structural induction hypothesis, we know that each statement is correctly compiled. The correctness of the compilation of a sequence of statements naturally follows.

\langle 1 \rangle 2. CASE: $s = r \leftarrow \text{newRef } e$

PROOF SKETCH: Here, we show that *newRef* statements are correctly compiled. To do so, we have to evaluate the term on one hand and compile/execute it on the other hand. Then, we show that the \sim_{heap} and \sim_{mem} relations are preserved and that the outcomes are the same.

$$\langle 2 \rangle 1. \text{ Evaluation: } \left(\begin{array}{l} \text{mem}, \\ \text{heap}, \\ r \leftarrow \text{newRef } e \end{array} \right) \Rightarrow \left(\begin{array}{l} \text{Normal}, \\ \text{mem}[r \mapsto v], \\ \text{heap}[r \mapsto \text{ref } (\text{pointerT } t_e \text{ Available}) r] \end{array} \right)$$

$$\langle 2 \rangle 2. \text{ Compilation: } \left(\begin{array}{l} \text{binding}, \\ r \leftarrow \text{newRef } e \end{array} \right) \xrightarrow{c} \left(\begin{array}{l} \text{binding}[r \mapsto \text{ref } (\text{pointerT } t_e \text{ Available}) r], \\ \left[\begin{array}{l} \text{local: } \quad ct_e \ r \\ \text{stmts: } \quad r = dc_e \end{array} \right] \end{array} \right)$$

\langle 2 \rangle 3. Execution: $\text{code.global}, \text{code.local} \vdash (r = dc_e), M \xrightarrow{C} \text{Normal}, M'$

\langle 2 \rangle 4. $\text{heap}' \sim_{\text{heap}} \text{binding}'$

\langle 2 \rangle 5. $\text{out} = \text{out}'$

\langle 2 \rangle 6. $M' \sim_{\text{mem}} \text{mem}'$

PROOF SKETCH: To prove that the \sim_{mem} relation is preserved, we have to take a closer look at the Clight memory operation and show that they extend the memory with equivalent maps. Hence, we explore the Clight derivation tree associated with step \langle 2 \rangle 3, starting with Clight rule (20).

\langle 3 \rangle 1. $\text{code.global}, \text{code.local} \vdash dc_e, M \xrightarrow{c} x$ with $x \sim_{\text{mem}(M, \text{mem})} v$

PROOF SKETCH: To show that the computed value x is indeed equivalent to v , we must again explore the Clight derivation that leads to this state. Two cases must be considered: either we are evaluating a base type, such as integer or float, or it is a pointer.

⟨4⟩1. CASE: t_e is an integer or a float

PROOF SKETCH: In case of a base type, the execution directly returns a value, which is equal to v .

⟨4⟩2. CASE: t_e is a pointer

PROOF SKETCH: In case of a pointer, the term dc_e is not the same if we are compiling an available or a read pointer. Hence, the following two cases.

⟨5⟩1. CASE: t_e is an available pointer

PROOF SKETCH: After exploring the Clight derivations further by applying rules (1) followed by (9), we succeed in proving that dc_e is indeed executed to a value equivalent to v .

⟨5⟩2. CASE: t_e is a read pointer

PROOF SKETCH: This proof follows ⟨5⟩1 by applying Clight derivation rule (1) then rule (8).

⟨3⟩2. $M' = M[b \mapsto x]$

⟨3⟩3. Q.E.D.

PROOF: By ⟨2⟩1, ⟨3⟩1, and ⟨3⟩2

⟨2⟩7. Q.E.D.

PROOF: By ⟨2⟩4, ⟨2⟩5, and ⟨2⟩6

⟨1⟩3. CASE: $s = r_1 \leftarrow readRef\ r_2$

PROOF SKETCH: In case of *readRef* statement, the proof is as follows: as usual, we start by evaluating the statement on one hand and compiling/executing it on the other hand. Then, we show that both outcomes are the same and that the \sim_{mem} and \sim_{heap} relations are preserved.

⟨2⟩1. Evaluation: $\left(\begin{array}{l} mem, \\ heap, \\ r_1 \leftarrow readRef\ r_2 \end{array} \right) \Rightarrow \left(\begin{array}{l} Normal, \\ mem[r_1 \mapsto mem.ra(l)], \\ heap[r_1 \mapsto ref\ tr_2\ r_1] \end{array} \right)$

⟨2⟩2. Compilation: $\left(\begin{array}{l} binding, \\ r_1 \leftarrow readRef\ r_2 \end{array} \right) \xrightarrow{c} \left(\begin{array}{l} binding[r_1 \mapsto ref\ t_{r_1}\ r_1], \\ \left[\begin{array}{l} local: ct_{r_1}\ r_1 \\ stmts: r_1 = c_{r_2} \end{array} \right] \end{array} \right)$

⟨2⟩3. Execution: $code.global, code.local \vdash (r_1 = c_{r_2}), M \xrightarrow{C} Normal, M'$

⟨2⟩4. $heap' \sim_{heap} binding'$

⟨2⟩5. $out = out'$

⟨2⟩6. $M' \sim_{mem} mem'$

PROOF SKETCH: As in the *newRef* case, the memory equivalence step requires exploring the Clight derivation tree further by first applying rule (20). In particular, the execution of the r-value requires more work.

⟨3⟩1. $code.global, code.local \vdash c_{r_2}, M \xrightarrow{C} x$ with $x \sim_{mem(M, mem)} mem(l)$

PROOF SKETCH: c_{r_2} compiles to distinct terms depending on the type *type* of r_2 . Hence, the following case analysis on this type:

⟨4⟩1. CASE: $type = pointerT\ (pointerT\ type'\ m')\ m$

PROOF SKETCH: Based on the case assumptions above, we can unroll the Clight derivation tree with rules (1), (2), and (8) up to values that, by the induction hypothesis, are known to be equivalent. Then, we simply show that this equivalence is preserved along the derivations.

⟨4⟩2. CASE: $type = pointerT\ type'\ m$

PROOF SKETCH: The proof is similar to the previous case ⟨4⟩1 with the exception that it follows rules (1) and (8).

⟨3⟩2. $M' = M[b \mapsto x]$

⟨3⟩3. Q.E.D.

PROOF: By ⟨2⟩1, ⟨3⟩1, and ⟨3⟩2

⟨2⟩7. Q.E.D.

PROOF: By ⟨2⟩4, ⟨2⟩5, and ⟨2⟩6

⟨1⟩4. CASE: $s = writeRef\ r\ e$

PROOF SKETCH: This proof adopts the by-now standard technique: we evaluate the *writeRef* statement and also compile/execute it. We easily show that both outcomes are similar and \sim_{heap} is preserved.

A little more work is required to show that \sim_{mem} is also preserved.

$$\langle 2 \rangle 1. \text{ Evaluation: } \begin{pmatrix} \text{mem}, \\ \text{heap}, \\ \text{writeRef } r \ e \end{pmatrix} \Rightarrow \begin{pmatrix} \text{Normal}, \\ \text{mem}[m \mapsto v], \\ \text{heap} \end{pmatrix}$$

$$\langle 2 \rangle 2. \text{ Compilation: } \begin{pmatrix} \text{binding}, \\ \text{writeRef } r \ e \end{pmatrix} \xrightarrow{c} \begin{pmatrix} \text{binding}, \\ [\text{stmts: } c_r = dc_e] \end{pmatrix}$$

$$\langle 2 \rangle 3. \text{ Execution: } \text{code.global}, \text{code.local} \vdash (c_r = dc_e), M \xrightarrow{C} \text{Normal}, M'$$

$$\langle 2 \rangle 4. \text{heap}' \sim_{\text{heap}} \text{binding}'$$

$$\langle 2 \rangle 5. \text{out} = \text{out}'$$

$$\langle 2 \rangle 6. M' \sim_{\text{mem}} \text{mem}'$$

PROOF SKETCH: As usual, to solve this goal, we have to go deeper in Clight derivation tree. In particular, we have to execute the right-hand side of the assignment after having followed rule (20).

$$\langle 3 \rangle 1. \text{code.global}, \text{code.local} \vdash dc_e, M \xrightarrow{C} x \text{ with } x \sim_{\text{mem}(M, \text{mem})} v_e$$

PROOF SKETCH: As in the *newRef* case, we might write a base type or a pointer. Because these two cases do not compile to the same code, we have to handle them separately.

$$\langle 4 \rangle 1. \text{CASE: } t_e \text{ is an integer or float type}$$

PROOF SKETCH: In this case, both Clight and FoF evaluate dc_e to the same value. The proof is trivial, by application of Clight rules (5) or (6).

$$\langle 4 \rangle 2. \text{CASE: } t_e \text{ is a pointer}$$

PROOF SKETCH: This case requires further Clight derivation unrolling. However, the derivation path depends on whether v_0 is in *Read* or *Available* mode. Therefore, we make another case analysis.

$$\langle 5 \rangle 1. \text{CASE: } \text{type} = \text{pointerT } \text{type}' \text{ Read}$$

PROOF SKETCH: At this stage, we know enough, thanks to the case assumptions, to unroll the Clight derivation tree with rules (1) then (8) until two equivalent Clight and FoF values are reached. Then, we show that the derivations preserve this equivalence.

$$\langle 5 \rangle 2. \text{CASE: } \text{type} = \text{pointerT } \text{type}' \text{ Available}$$

PROOF SKETCH: This case is similar to $\langle 5 \rangle 1$, following rules (1), (9), and (8).

$$\langle 3 \rangle 2. M' = M[b \mapsto x]$$

$$\langle 3 \rangle 3. \text{Q.E.D.}$$

PROOF: By $\langle 2 \rangle 1$, $\langle 3 \rangle 1$, and $\langle 3 \rangle 2$

$$\langle 2 \rangle 7. \text{Q.E.D.}$$

PROOF: By $\langle 2 \rangle 4$, $\langle 2 \rangle 5$, and $\langle 2 \rangle 6$

$$\langle 1 \rangle 5. \text{Q.E.D.}$$

PROOF: By $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, and $\langle 1 \rangle 4$

This concludes the correctness proof. Several lessons can be learned from this proof, from which we draw some conclusions in the next section. First of all, we note that the proof is structured by deeply-nested case analysis: most of the actual work consists of identifying the relevant cases, the remainder is simply about applying derivation rules and induction hypothesis. This leads to a second observation: although individual proof steps are not extremely difficult, the actual complexity arises from the consequent number of derivation steps and case analysis. Finally, once the structure is set up, the sequence of proof steps is intuitive. The positive impact is that writing and reading the proof is more natural. The negative impact is the risk of overlooking a corner case, as is often the case in informal compiler verification.

Summary

Although this pen and paper proof gives some strong guarantees about the correctness of the FoF compiler, it also raises several deeper issues. First of all, we have studied a very restricted subset of FoF, namely reference cells. While we believe that similar proofs could be carried out for the remaining constructs of the language, this would require significantly more work. Indeed, the current proof is already quite involved and, consequently, long and hard to handle. Worse, whereas the FoF infrastructure can be flawlessly extended with new constructs, this modularity does not apply to our proofs: introducing a new construct calls for updating the whole proof. Roughly, the complexity of the proof increases exponentially with the number of constructs. We have experienced this effect during FoF implementation, when informally verifying the correctness of our code. Finally, FoF was designed to evolve quickly, to fit its users requirements. Because the paper proof is disconnected from the actual implementation, this requires an extra work to synchronize the proof with the actual implementation. Then, the complexity of the proof strikes again: facing the need for a modification, we either risk losing the formal guarantees provided by a proof, or block the update unless a long and tedious proof of correctness is provided.

To manage the complexity of language correctness proofs, a recent and successful trend [1, 6, 24, 36] has been to *mechanize* the language semantics in a theorem prover and to carry out proofs in this formal framework. Because the proof is entirely developed and checked by a trusted theorem prover, the correctness result proven is as trustworthy as the logic of the theorem prover itself. Needless to say, we have extremely strong guarantees on the correctness of these provers. Moreover, most proving systems support *code extraction*: because proofs are programs, the theorem prover is able to extract executable Haskell code from the proofs. Hence, the proof is never out of sync with the implementation: the implementation is actually extracted from the proof. However, several issues remain. First, mechanizing a semantics and carrying the complete correctness proof is a significant amount of work. Second, the evolution of the compiler is again slowed down: before using a new construct, we must first extend the mechanized compiler, hence updating the correctness proof. Although this trade-off is acceptable in the context of general-purpose programming languages – the language constructs are supposed to be stable, this can hardly be defended in the case of FoF: it needs to be able to evolve quickly, in order to embrace extremely diverse use-cases.

For these reasons, we are currently considering *translation validation* techniques [31, 33, 34]. Standard proof techniques strive for proving the complete correctness of the compiler: the compiler must behave correctly for *any* input. This high-standard has a direct impact on the proof complexity, as we have seen above. Translation validation, on the other hand, gives weaker guarantees: along the compilation process, a validating compiler builds a derivation tree justifying the correctness of the translation. This derivation tree can then be checked by an external, trusted prover. Therefore, a validating compiler only gives a proof that, for a particular input, its output is correct. In case of a compiler bug, it automatically identifies the source of miscompilation and the corresponding generated code. Because of this flexibility, the logical framework is less burdensome than the one used in exhaustive correctness proofs. Being lighter, these proofs can be automatically generated by the compiler and then automatically checked by a specialized prover. Hence, we expect this technique to catch up with the quick evolution of our compiler. Moreover, being integrated in the compiler, it would naturally respect and follow the extreme modularity of FoF compiler. This promising technique is currently being adapted to FoF.

6 Case Study: Hamlet

In the previous sections, we have explored the possibilities offered by FoF. We have also claimed that FoF provides a sound basis for DSL development, and we have substantiated this claim by proving the correctness of a subset of FoF. Given this confidence in FoF, we develop, in this section, a more ambitious DSL that aims at handling the Barrelfish *capability* system. In Section 6.1, we introduce the reader to the concept of capabilities. Then, in Section 6.2, we describe the key aspects of Hamlet, our language for capability description. In Section 6.3, we show that FoF can be leveraged to assess the correctness of a DSL, here Hamlet.

6.1 Barrelfish capability infrastructure

Historically, capabilities [27, 42] have been a major step toward more reliable operating systems. In a nutshell, capabilities are a powerful and expressive system to manage resources in an operating system. We refer the reader to Levy’s comprehensive study of capability systems for further information [27].

In the following, it is sufficient to understand a capability in Barrelfish as a *key*. When the operating system wants to provide a process with the right to access a resource, it gives the process a capability, *i.e.* an unforgeable key allowing the process to access that particular resource. For example, when a process creates a file according to some access mode – read, write, or execute, the operating system returns a capability referring to that file, with the corresponding access rights.

The process can then copy the capability, *i.e.* duplicate the key, and share access with another process. When any process wants to manipulate the file, such as by reading it, it invokes the kernel with the file capability. The kernel then checks that the operation invoked is legal with respect to the capability rights. Furthermore, it is sometimes possible to *retype* a capability: for instance, a process owning a file with read and write mode might want to offer a read-only access of this file to another process. Hence, it needs to be able to retype a capability to another, according to some high-level policy – ensuring, for example, that it is not possible to get a “powerful” capability out of a “less powerful” one. Finally, the process might decide to *revoke* the capability of the read-only file: the kernel has to find this capability but also, potentially, all its copies and to destroy them.

Thanks to capabilities, the system designer is provided with a single mechanism to solve all resource management problems at once. Hence, capabilities apply both to hardware and software resources, such as access to device drivers, memory, communication channels, file-systems, etc. The price of this generality is that the implementation of the capability system is a challenge in itself and that this code is absolutely critical for the safety of the system.

6.2 Generating the capability infrastructure

As we have seen, unforgeability is a fundamental requirement for capability system. In Barrelfish, the per-kernel capability database is physically protected by the MMU: it is stored in a set of pages that cannot be mapped, hence accessed, by user-mode processes. Because a capability is a memory object, this makes the actual physical layout of capabilities an intrinsic parameter of the capability system. Indeed a major part of the capability code consists in the description of the various capabilities structures, according to their parameters – such as, for instance, block offset and access modes for a file capability.

This low-level memory layout of capabilities is then abstracted away by four predicates. Thanks to these predicates, the capability system manipulates the capabilities without having to interact with their low-level implementation. Hence, these predicates are of first importance for the whole infrastructure: they enable a higher-level programming model, *i.e.* a more trustworthy code. The first predicate is `is_well_founded(A,B)`, which checks whether it is legal, with respect to some high-level policies, to retype

```

cap RAM {
    /* RAM memory object */

    retype_to {
        RAM: { base, bits },
        Frame: { base, bits },
        Dispatcher: { mem_to_phys(dcb), sizeof(dcb) },
        CNode: { cnode, cnode_size + bits },
        VNode4: { base, vnode_size },
        VNode3: { base, vnode_size },
        VNode2: { base, vnode_size },
        VNode1: { base, vnode_size }
    };

    eq paddr base; /* Base address of untyped region */
    eq uint8 bits; /* Address bits that untyped region bears */
};

```

Figure 6.1: Hamlet sample: the RAM capability

a capability of type A to a capability of type B. Encoding these high-level policies in the C programming language is, in itself, a painful and error-prone task. The second predicate is `is_revoked_first(D)`, which verifies that a capability D has been revoked, before doing any modification on it. However, depending on some high-level policies, this test should not be applied to certain capabilities. The third predicate is `is_copy(D,E)`, which tests whether D and E are copies. This involves comparing some specific fields of both capability structures, according to an high-level definition of equality. The fourth and last predicate is `is_ancestor(D,E)`, which tests whether a capability E has been obtained through one or several retyplings of a capability D. Again, this involves some low-level comparisons of fields, governed by a high-level comparison operation. However, because they are fairly dependent on low-level details, they are bug-prone, hard to verify, hard to maintain, and painful to extend. For instance, whereas the code was mature, extensively used and tested, we have found a major bug during our work on Hamlet: a user was able to retype a memory capability to a bigger memory region, *i.e.* a much more “powerful” capability.

To sum up, two key characteristics govern the Barrelfish capability system. First, a capability is, in itself, a low-level memory object, result of the projection of semantically-rich abstract properties, *e.g.* a file capability and its fields. Second, capability management is greatly simplified by four predicates that translate high-level policies into low-level manipulation on the physical capability representation. These sources of difficulty would be manageable if the set of capabilities were defined once and for all. However, the contrary actually occurs: because capabilities are so good in abstracting resource management, developers constantly feel the need to extend them. Because this code is complex and safety-critical, developers have to refrain from experimenting with new ideas, or they spend a considerable amount of time dealing with bug-prone low-level code.

In this setting, the role of Hamlet is twofold. First, based on a high level description of the capabilities, it defines the corresponding data-structures. This amounts to build a set of complex data-types, based on a combination of `struct`, `union`, and base C types. Second, it generates the code corresponding to the four predicates described above. This consists of translating the high-level policies into low-level manipulations of the capability data-structures. For further information, we refer the reader to Hamlet literate code [14]. The high-level description of a capability is shown in Figure 6.1. From such description, Hamlet generates two C files, respectively containing the capability data-structures and the predicates. These files are then directly integrated into the Barrelfish source code, without any manual intervention.

```

prop_validPaths_allSrcCap :: Capabilities → Bool
prop_validPaths_allSrcCap allCapabilities =
  and [ srcCap 'elem' srcCapsRes
       | (srcCap, validDstCaps) ← validPath ]
where caps = ...
      enum = ...
      validPathsRes = ...
      srcCapsRes = ...
      validPath = ...

```

Figure 6.2: Hamlet test: example of property

6.3 Assessing Hamlet correctness

In the previous sections, we have argued that FoF eases the implementation of DSLs. With Fugu, we have gathered some evidence sustaining this claim. Because FoF gives a functional semantics to the DSL, we have also argued that it helps in verifying the correctness of DSLs. In this section, we demonstrate this point by partially testing the correctness of Hamlet. We make the assumption that the parser of Hamlet is correct, hence computing an AST representative of the user input. This assumption is usually made by certified compilers, such as CompCert [6]. Moreover, the syntax of Hamlet is simple, hence the parser is small and reliable. We also assume that the FoF-to-C compiler is correct, and show how we can leverage this confidence to conveniently perform powerful reasoning.

In particular, we will focus on the predicate `is_well_founded` described above. This function implements the retyping policy, as specified in the Hamlet description. The compilation from the AST to FoF code is decomposed in three functions, from the closest to the AST to the final result. The first function is:

```
validPaths :: Capabilities → [(FoFCode f, [FoFCode f])]
```

The role of `validPaths` is to extract the list of retypable capabilities and their respective list of retype destinations. A capability A can be retyped to B if and only if (A, L) belongs to this extracted list and B belongs to L . The second function is:

```
validateRetypeCode :: PureExpr → [FoFCode f] → FoFCode f
```

The first argument of `validateRetypeCode` is a FoF expression denoting the type of the destination capability. The second argument is a list of valid destination types. This function compiles an `if` statement that checks whether the given type is indeed correct with respect to the valid destinations. The last function is:

```
is_well_founded :: Capabilities → FoFCode f
```

This function generates the implementation of the `is_well_founded` function used in Barrelfish. It compiles to a C function that takes two arguments, a source capability type and a destination capability type, and returns true if the retyping is allowed, false otherwise. It relies on `validPaths` and `validateRetypeCode`.

In the following, we demonstrate the use of QuickCheck [8] to assess the correctness of `is_well_founded`. QuickCheck is a library for random testing in Haskell. Being a library, it is therefore easy to use, but also efficient. While random testing does not give any formal guarantee, it is a first step toward more formal verification techniques. In particular, the invariants we test actually encode the correctness of the compiler. For a formal proof, we would have to prove the truth of the same properties. Hence, random testing is a lightweight technique to start the verification effort.

In our case, we have instantiated the AST, of type `Capabilities`, as `Arbitrary`: QuickCheck is able to randomly generate valid ASTs of varying size and complexity. Therefore, we can test the truth of any property of type `Capabilities → Bool`, for instance.

The first invariants concern `validPaths`. It consists of three predicates:

```

prop_validPaths_allSrcCap :: Capabilities → Bool
prop_validPaths_allDstCap :: Capabilities → Bool
prop_validPaths_notDstnotValid :: Capabilities → Bool

```

Property	Time (s)
<code>prop_validPaths_allSrcCap</code>	< 1
<code>prop_validPaths_allDstCap</code>	2
<code>prop_validPaths_notDstnotValid</code>	3
<code>prop_validateRetypeCode_in</code>	49
<code>prop_is_well_founded_in</code>	1306

Table 6.1: Timing of Hamlet tests

The first property states that all capability types specified in Hamlet still belong to the result of `validPaths`, meaning that no capability is forgotten. The second property enforces that all destination capability types specified by the user are preserved by the transformation, meaning that the computed set of valid destinations is at least as big as the specified one. The last property enforces that all destination capability types computed from the specification were already present, meaning that the computed set of valid destinations is at most as big as the specified one. All in all, these three properties ensure that the retyping path is preserved by this transformation. To illustrate their simplicity, a sample of such a predicate is shown in Figure 6.2.

The correctness of `validateRetypeCode` is tested by the following property:

```
prop_validateRetypeCode_in :: Capabilities → Bool
```

This property relies on the correctness of `validPaths`. Remember that `validateRetypeCode` takes a destination capability type and a list of valid destinations. This test verifies that the FoF code calls `return` with `true` if the given destination is valid, and `false` otherwise.

Finally, the correctness of `is_well_founded` is specified by the property:

```
prop_is_well_founded_in :: Capabilities → Bool
```

This property relies on `validPaths` and `validateRetypeCode`. It ensures that, given a source and a destination capability type, `is_well_founded` returns true if and only if the destination type belongs to valid destinations of the source type.

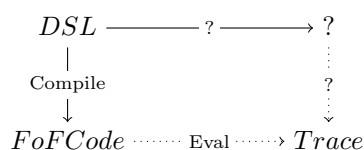
We have tested these properties by running each of them against 100 randomly generated input ASTs. Our experiments have been performed on a 64-bit Intel Core 2 duo, clocked at 3GHz. All tests were successful; the time to complete 100 tests per property is reported in Table 6.1. Although these results do not *prove* the correctness of Hamlet, they give strong evidence of its correctness. Therefore, it is reasonable to believe that the Hamlet semantics is preserved down to the FoF code. If we further assume the correctness of the FoF-to-C compiler, then we can be confident in the C code generated from any Hamlet description. This must be compared with the state of the art in compiler verification. The majority of compilers, including most DSLs, are tested through a simple harness: the compiler is run against a set of inputs and its output is checked against an expected compilation result. Here, we randomly generate inputs, at will. Furthermore, the verification does not consist of verifying that the compilation produces a file “similar” to a correct one: thanks to FoF, we are able to test whether the output is *semantically* correct or not. This has two major impacts. First, the DSL designer is freed from the need to write tedious test cases and manually check the correctness of the output. Second, the process of generating inputs *and* verifying the outputs is completely automated, allowing a better exploration of the state space.

Note that these properties apply to *any* input AST and, if true, ensure that the generated code is correct. In particular, provided a capability specification by a user, the Hamlet compiler is able to ensure that the code generated for this input is semantically correct. Meaning that, 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 an extremely lightweight and efficient translation validation infrastructure for her DSL.

Summary

First, as for Fugu, it is worth mentioning that Hamlet is not a simple proof-of-concept but a mature tool, currently used in Barrelfish. Unlike Fugu, Hamlet is a much more substantial piece of software, both in term of the DSL compiler but also in term of the amount of generated code. Moreover, the generated code actually lays the foundations of the whole capability infrastructure of Barrelfish. The safety of the operating system thus relies entirely on the correctness of the compiler. This emphasizes, once again, the importance of a formally-verified compilation process. Interestingly, the introduction of Hamlet in Barrelfish had the effect of further reducing the amount of low-level code in the TCB: 250 lines of low-level C code have been replaced by 100 lines of high-level Hamlet code.

With Hamlet, we make a case for *meta-testing* system code with FoF, as a first step before meta-proofs. Indeed, we never had to test or prove that the C code implementing the predicates is correct. Nor did we need to test or perform a proof on the FoF code that compiles to this C code. Instead, we test the DSL *compiler* to ensure that it always generates semantically correct code. In the general case, this consists in testing or proving that the following diagram commutes:



Maybe surprisingly, the price to pay for this generality is actually very small. First, QuickCheck makes meta-testing in Haskell simple while still efficient. Needless to say, such testing is at best prohibitive on C code and gives fewer guarantees. Moreover, when testing or proving the correctness of a particular instance of the problem, it is often the case that the details of the instance do not matter and are ignored by the proof – such as which source type can retype to which destination type. It is therefore easy, and in any case a good practice, to generalize the proof to abstract over the particularities of the instance: this corresponds to the DSL compiler correctness proof. More lightweight than the DSL compiler correctness proof, we have seen that translation validation is straightforward in this setting: the properties we are to prove can be encoded in Haskell and checked at compile-time for the specific user input. Thanks to this infrastructure, we are ensured that the generated code is *always* correct if the compiler succeeds.

From a Barrelfish developer’s point of view, Hamlet is not only a DSL that avoids the need to write C code. Hamlet is actually the *specification* of the Barrelfish capability system. Hence, the developer is relieved from the task of proving that her code respects the semantics of the capability system: this is done once and for all in the DSL compiler. By design of Hamlet, the developer writes correct code, without suffering the burden of proofs: she can modify or add new capabilities, transparently and without having to test or prove any code. This is a clear step forward compared to a top-down approach to verification, where a single modification of the code requires updating the whole proof.

Because the capability system is a crucial component, we are willing to carry out more verification work on Hamlet descriptions. This verification effort can take several forms. As we have seen, random testing a la QuickCheck is one: it is a cheap and efficient way to get some assurance of the correctness of the DSL. Obviously, pen and paper or mechanized correctness proofs are another approach. We plan to explore these opportunities in future work.

7 Discussion

In this section, we contrast the bottom-up approach advocated by FoF with other approaches used to increase the reliability of system code.

A tempting solution to enhance reliability is to use a safe low-level language, such as Cyclone [21], Deputy [10], or Sing# [3]. Because these languages strive to be as good as C in every domain, they must be able to safely express all idioms that are allowed by the unsafe C language. This has a direct consequence on the language design: for example, Cyclone was initially backed by a formal semantics. However, during its evolution, the formal and the real semantics came out of sync, highlighting the extreme complexity of the language. This also has an indirect consequence: the more complex the language, the less trustable the compiler. Furthermore, although the compiler is correct, it is extremely hard to compete with the performance achieved by a standard C compiler, thoroughly engineered for several decades. Moreover, let us consider the bit twiddling code for device interaction. Implementing this code either in C or in one of aforementioned safe language does not solve our problems: first, both implementations are bound to look the same and, worse, we still lack a high-level semantics for these operations. While a Mackerel user describes the semantics of a device register, a user of a safe low-level language is bound to meaninglessly interact with an array of bits. We argue that both approaches are complementary. In this sense, we could extend FoF to output Cyclone, Deputy, or Spec# code.

The meta-compilation [16] technique has been successfully used to verify large-scale systems, such as the Linux kernel. The basic idea of meta-compilation is that an API implicitly defines a high-level language, which is therefore governed by a semantics. The contribution has been to formalize these semantics and to use automated tools to find safety violations in C code, *i.e.* violation of the high-level semantics, whereas C is intrinsically deprived of any semantics. While this technique gives impressive results, this requires a fair amount of work and gives up *completeness*: it is able to *find* bugs but it cannot prove the *absence* of bugs. On the other hand, FoF aims to give a semantics to this low-level code. Hence, we only have to reason about a high-level language, supported by a mechanized semantics. The direct benefit of our approach is that, first, we do not give up completeness – hence the ability to prove the absence of bugs – and the second, analysis can rely on more powerful mathematical and software tools.

Finally, FoF enables an alternative to the standard top-down approach to reliability. Usually, to prove the correctness of a system, one must start with a high-level model and refine it down to machine code, or at least C. This approach has been used to perform the full verification of the seL4 operating system [15]. Whereas this technique is fairly standard in software verification, its application at the scale of an operating system is challenging. Indeed, the high-level model has to handle every corner cases that will be dealt with by the lower-level models. In a sense, the designer is asked to predict future needs and build her model accordingly. Obviously, even experienced designers cannot achieve this level of perfection. Therefore, the high-level design is iteratively improved while the refinement mapping proofs to the lower-level models are worked on. The direct impact of these iterations is that each time the high-level model is modified, the proofs have to be checked again and potentially restarted from scratch, as reported by the developers of seL4 [9]. On the other hand, the bottom-up approach we advocate starts from a mature infrastructure, written in C and having passed the test of time. Here, the DSL designer abstracts away the essence of the infrastructure and formalizes its semantics at the same time. Perhaps surprisingly, we have shown that this lightweight approach can offer the same guarantee as top-down verification: with the DSL as the specification, we rely on the correctness of the DSL compiler to ensure that any generated code respects the DSL semantics, *i.e.* the specification. Even though the DSL has not been proved, we can also rely on a cheap translation validation infrastructure to ensure that, for the given input, the resulting implementation is indeed correct. Unlike the top-down approach, which requires full verification or nothing, our approach allows the developers to devote their energy to the critical parts of the system.

8 Conclusion

It is popular wisdom that operating system design is a difficult art. Nonetheless, as the operating system is a critical component of a computer, its correctness is critical. The developers have to spend a significant amount of energy in tracking and correcting bugs. Moreover, their task is made difficult by the proximity of the hardware: whereas the operating system is asked to present a consistent interface, the hardware does not. Indeed, hardware can be defective or, worse, might not respect its specification.

Moreover, the actions performed by an operating system are too low-level to be implemented efficiently in a high-level language. Therefore, developers are bound to use C. They have to give up type and memory safety as well as a well-specified semantics. Similarly, typical system code is hardly amenable to software verification. The extensive use of pointers and the proximity with hardware make most, if not all, analyses intractable or incomplete.

These difficulties have been faced during the development of the Barrelfish operating system. However, the ambitious design goals of Barrelfish – as described in Section 2 – have also raised new challenges. First, Barrelfish targets commodity multi-core machines. Hence, it has to support various hardware and work around their potential defects. Second, the multikernel model followed by Barrelfish organizes the operating system as a distributed system. This radical organization has motivated the use of DSLs, such as Mackerel and Flounder. Hence, DSLs automate the implementation of error-prone code by translating high-level descriptions into low-level C code.

In Section 3, we have shown that Filet-o-Fish grows from this trend of enhancing the safety of system code thanks to DSLs. In particular, FoF proposes to materialize this bottom-up approach by offering formal guarantees: developing a DSL with FoF consists of abstracting a problem with a high-level language while, for free, formalizing the semantics of the DSL. Thanks to the mechanized semantics offered by FoF, it becomes possible to reason about the DSLs in a rigorous manner, whereas it is at best prohibitive to formally reason about C code. Also, by relying on the correctness of the FoF-to-C compiler, it is now *sufficient* to reason at the level of the DSL semantics, hence completely ignoring the generated C code. Unlike safe low-level languages, our DSLs compilers are small, hence more trustable. Moreover, in Section 5, we have proved the correctness of the compiler for a subset of the language.

To gather practical evidence of its strengths, we have developed two DSLs based on FoF. The first, Fugu, aims at alleviating the difficulty of handling errors in an operating system. Because the back-end uses several FoF idioms, it offers an exhaustive view of FoF programming style. We have shown some of them in Section 4 and argued that FoF is convenient as a *programming* tool.

We have also developed Hamlet, a language to specify the capability system of Barrelfish. In Section 6, we have described how Hamlet translates declarative resource management policies into low-level data-structure manipulations. Because resource management is safety critical, we have given a practical example of *meta-testing*: instead of testing C code or even FoF code, we used QuickCheck to assess the correctness of the Hamlet compiler. Therefore, a developer specifying the capability system is relieved from the burden of proofs: her code is correct *by construction*. Beyond testing, we have also identified several promising avenues for verification techniques. With Hamlet we have shown that FoF is powerful as a *reasoning* tool as well.

Last but not least, these two languages, and by extension FoF, are used extensively in Barrelfish. Fugu is a key element in the on-going effort to improve the management of errors in Barrelfish. Hamlet has discharged the developers from the burden of maintaining and evolving the error-prone details of the capability infrastructure. At the time of writing, the code generated by these tools has been deployed for more than four weeks, without any inconvenience or noticeable dysfunction.

In this thesis, 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 easy. Thanks to its functional semantics, the verification of DSLs is tractable. Hence, in one go, we are able to abstract away the functionality and formalize their semantics.

Future work

As part of future work, we plan to implement a translation validation infrastructure (TVI) in FoF, as advocated in Section 5.4. We are convinced that this technique could benefit FoF by respecting its modular design and not hindering its dynamism. While, from a practical point of view, FoF is already a mature tool, translation validation would complete the theoretical basis of FoF far beyond the current correctness proof.

Finally, guided by the work of Padioleau et al. [32] as well as by the needs of Barrelfish developers, we plan to develop more DSLs, and probably to apply our techniques outside of Barrelfish, such as on the Linux kernel.

A Filet-o-Fish Compiler Correctness Proof

ASSUME: 1. $heap \sim_{heap} binding$
 2. $M \sim_{mem} mem$

LET: s a FoF statement

PROVE: If

$$\begin{array}{ccc} heap, mem, s & \Rightarrow & out, heap', mem' \\ binding, s & \xrightarrow{c} & code, binding' \end{array}$$

$$code.global, code.local \vdash code.stmts, M \xrightarrow{C} out', M'$$

Then the following relations hold:

$$\begin{array}{ccc} heap' & \sim_{heap} & binding' \\ M' & \sim_{mem} & mem' \\ out & = & out' \end{array}$$

PROOF SKETCH: We proceed by structural induction on the derivation tree. Hence, we do a case analysis on the last term of the tree.

\langle 1 \rangle 1. CASE: $s = r \leftarrow newRef\ e$

PROOF SKETCH: Here, we show that $newRef$ statements are correctly compiled. To do so, we have to evaluate the term on one hand and compile/execute it on the other hand. Then, we show that the \sim_{heap} and \sim_{mem} relations are preserved and that the outcomes are the same.

LET:

- $v = \mathbb{E}[e]$
- $t_e = \text{typeof}(e)$
- $mem' = mem[r \mapsto v]$
- $heap' = heap[r \mapsto \text{pointerT } t_e \text{ Available}]$

$$\langle 2 \rangle 1. \left(\begin{array}{c} mem, \\ heap, \\ r \leftarrow newRef\ e \end{array} \right) \Rightarrow \left(\begin{array}{c} Normal, \\ mem', \\ heap' \end{array} \right)$$

PROOF: Because the only rule that can be applied to s is (eval-newRef-anon)

LET:

- $ct_e = \text{toC}_t(t_e)$
- $c_e = \text{toC}_e(e)$
- $dc_e = \text{deref}(t_e, c_e)$
- $binding' = binding[r \mapsto \text{pointerT } t_e \text{ Available}]$

$$\langle 2 \rangle 2. \left(\begin{array}{c} binding, \\ r \leftarrow newRef\ e \end{array} \right) \xrightarrow{c} \left(\begin{array}{c} binding', \\ \left[\begin{array}{l} \text{local: } ct_e\ r \\ \text{stmts: } r = dc_e \end{array} \right] \end{array} \right)$$

PROOF: Because the only that can be applied to s is (compile-newRef-anon)

LET:

$out' = Normal$

$$\langle 2 \rangle 3. code.global, code.local(r = dc_e), M \xrightarrow{C} out', M'$$

PROOF: Because the only rule that can be applied to $r = dc_e$ computed in \langle 2 \rangle 2 is CLight rule (20)

$$\langle 2 \rangle 4. heap' \sim_{heap} binding'$$

PROOF:

- By Induction Hypothesis (IH) $\langle 0 \rangle 1$, we had $heap \sim_{heap} binding$.
- In $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$, we extend $heap$ and $binding$ with identical mappings.

$\langle 2 \rangle 5$. $out = out'$

PROOF:

- By $\langle 2 \rangle 1$, $out = Normal$,
- By $\langle 2 \rangle 3$, $out' = Normal$

$\langle 2 \rangle 6$. $M' \sim_{mem} mem'$

PROOF SKETCH: To prove that the \sim_{mem} relation is preserved, we have to take a closer look at the Clight memory operation and show that they extend the memory with equivalent maps. Hence, we explore the Clight derivation tree associated with step $\langle 2 \rangle 3$, starting with Clight rule (20).

$\langle 3 \rangle 1$. $code.global, code.local \vdash r, M \stackrel{\xi}{\Leftarrow} (b, 0)$ with $b \sim_{mem(M, mem)} r$

PROOF:

- By $\langle 2 \rangle 3$, we have applied Clight rule (20)
- Thus, we evaluate r as a l-value
- By definition of $code.local$ at $\langle 2 \rangle 2$, $b \sim_{mem(M, mem)} r$

$\langle 3 \rangle 2$. $code.global, code.local \vdash dc_e, M \stackrel{\xi}{\Rightarrow} x$ with $x \sim_{mem(M, mem)} v$

PROOF SKETCH: To show that the computed value x is indeed equivalent to v , we must again explore the Clight derivation that leads to this state. Two cases must be considered: either we are evaluating a base type, such as integer or float, or it is a pointer.

$\langle 4 \rangle 1$. CASE: t_e is an integer or a float

PROOF SKETCH: In case of a base type, the execution directly returns a value, which is equal to v .

$\langle 5 \rangle 1$. $v = e$

PROOF:

- By Case Assumption $\langle 4 \rangle 1$, e is an integer or a float
- Thus, by definition of $\mathbb{E}[\cdot]$, e reduces to itself

$\langle 5 \rangle 2$. $dc_e = c_e$

PROOF:

- By Case Assumption $\langle 4 \rangle 1$, t_e is either $intT$ or $floatT$
- Thus, by definition of $deref(\cdot, \cdot)$, $deref(otherwise, c_e) = c_e$

$\langle 5 \rangle 3$. $code.global, code.local \vdash dc_e, M \stackrel{\xi}{\Rightarrow} v$

PROOF:

- By $\langle 2 \rangle 3$, we have applied Clight rule (20)
- Thus, we evaluate dc_e as a r-value
- By Case Assumption $\langle 4 \rangle 1$, dc_e is an integer or a float
- Thus, we apply either Clight rule (5) or (6) to evaluate the r-value
- By definition of rule (5) or (6), dc_e evaluates to the actual value v

$\langle 5 \rangle 4$. Q.E.D.

PROOF:

- By $\langle 5 \rangle 3$, dc_e evaluates to the same value v
- By definition of $\sim_{mem(M, mem)}$, we have $v \sim_{mem(M, mem)} v$

$\langle 4 \rangle 2$. CASE: t_e is a pointer

PROOF SKETCH: In case of a pointer, the term dc_e is not the same if we are compiling an available or a read pointer. Hence, the following two cases.

$\langle 5 \rangle 1$. $v = ref(pointerT t_e m) r$

PROOF:

- By Case Assumption $\langle 4 \rangle 2$, t_e is a pointer
- Thus, by definition of $\mathbb{E}[\cdot]$, e reduces to itself, *i.e.* to a reference

$\langle 5 \rangle 2$. $dc_e = \&c_e$

PROOF:

- By Case Assumption $\langle 4 \rangle 2$, t_e is a pointer
- Thus, by definition of $deref(\cdot, \cdot)$, $deref(pointerT t m, c_e) = \&c_e$

⟨5⟩3. CASE: t_e is an available pointer

PROOF SKETCH: After exploring the Clight derivations further by applying rules (1) followed by (9), we succeed in proving that dc_e is indeed executed to a value equivalent to v .

⟨6⟩1. $v = \text{ref } (\text{pointerT } t_e \text{ Available}) r'$

PROOF:

- By ⟨5⟩1, v is a reference
- By Case Assumption ⟨5⟩3, v is available

⟨6⟩2. $c_e = r'$

PROOF:

- By ⟨6⟩1, v is an available reference
- By Definition of c_e at ⟨2⟩2, $c_e = \text{toC}_e(e)$
- By Definition of toC_e applied to an available reference, $c_e = r'$

⟨6⟩3. $dc_e = \&r'$

PROOF:

- By ⟨5⟩2, $dc_e = \&c_e$
- By ⟨6⟩2, $c_e = r'$
- Thus, $dc_e = \&r'$

⟨6⟩4. $\text{code.global}, \text{code.local} \vdash c_e, M \stackrel{\mathcal{L}}{\Leftarrow} (b_v, 0)$ with $b_v \sim_{\text{mem}(M, \text{mem})} r'$

PROOF:

- By ⟨6⟩2, $c_e = r'$ with r' a previously defined variable
- Thus, by (IH) ⟨0⟩:2, r' has a corresponding block b_v in the C memory model, with $b_v \sim_{\text{mem}(M, \text{mem})} r'$
- Thus, by Clight rule (1), the variable c_e took as an l-value evaluates to b_v

⟨6⟩5. $\text{code.global}, \text{code.local} \vdash dc_e, M \stackrel{\mathcal{L}}{\Leftarrow} \text{ptr}(b_v, 0)$

PROOF:

- By ⟨6⟩4, c_e left-evaluates to b_v
- Thus, by CLight derivation rule (9), dc_e right-evaluates to $\text{ptr}(b_v, 0)$

⟨6⟩6. Q.E.D.

PROOF:

- By ⟨6⟩4, we have $b_v \sim_{\text{mem}(M, \text{mem})} r'$
- By ⟨6⟩5, we have $x = \text{ptr}(b_v, 0)$
- Thus, by definition of $\sim_{\text{mem}(M, \text{mem})}$, we have $x \sim_{\text{mem}(M, \text{mem})} v$

⟨5⟩4. CASE: t_e is a read pointer

PROOF SKETCH: Again, this proof follows ⟨5⟩3 by applying Clight derivation rules (1) then (8).

⟨6⟩1. $v = \text{ref } (\text{pointerT } t_e \text{ Read}) r'$

PROOF:

- By ⟨5⟩1, v is a reference
- By Case Assumption ⟨5⟩4, v is read

⟨6⟩2. $c_e = *r'$

PROOF:

- By ⟨6⟩1, v is a read reference
- By definition of c_e at ⟨2⟩2, $c_e = \text{toC}_e(e)$
- By definition of toC_e applied to a read reference, $c_e = *r'$

⟨6⟩3. $dc_e = r'$

PROOF:

- By ⟨5⟩2, $dc_e = \&c_e$
- By ⟨6⟩2, $c_e = *r'$
- Thus, $dc_e = r'$

⟨6⟩4. $\text{code.global}, \text{code.local} \vdash dc_e, M \stackrel{\mathcal{L}}{\Leftarrow} (b_v, 0)$ with $b_v \sim_{\text{mem}(M, \text{mem})} r'$

PROOF:

- By ⟨6⟩3, $dc_e = r'$ with r' a previously defined variable
- Thus, by (IH) ⟨0⟩:2, r' has a corresponding block b_v in the C memory model, with $b_v \sim_{\text{mem}(M, \text{mem})} r'$

- Thus, by CLight rule (1), the variable dc_e left-evaluates to b_v

LET:

$$p_v = \text{loadval}(\text{type}(r), M, (b_v, 0))$$

$$\langle 6 \rangle 5. p_v = \text{load}(\text{type}(r), M, (b_v, 0))$$

PROOF:

- By definition of p_v , $p_v = \text{loadval}(\text{type}(r), M, (b_v, 0))$
- By Case Assumption $\langle 4 \rangle 2$, b_v is accessed by value (pointer)
- Thus, by definition of $\text{loadval}(\cdot)$, p_v loads the value from block b_v

$$\langle 6 \rangle 6. p_v \sim_{\text{mem}(M, \text{mem})} v$$

PROOF:

- By $\langle 6 \rangle 4$, $b_v \sim_{\text{mem}(M, \text{mem})} r'$
- By $\langle 6 \rangle 5$, p_v corresponds to the value stored in b_v
- By $\langle 6 \rangle 1$, v is a read reference to r'
- Thus, by definition of $\sim_{\text{mem}(M, \text{mem})}$, $p_v \sim_{\text{mem}(M, \text{mem})} v$

$$\langle 6 \rangle 7. \text{code.global}, \text{code.local} \vdash dc_e, M \stackrel{c}{\Rightarrow} p_v$$

PROOF:

- By $\langle 6 \rangle 3$, $dc_e = r'$ with r' a variable
- Thus, by CLight derivation rule (8), we load the value p_v from memory

$$\langle 6 \rangle 8. \text{Q.E.D.}$$

PROOF:

- By $\langle 6 \rangle 7$, dc_e right-evaluates to $x = p_v$
- By $\langle 6 \rangle 6$, p_v is equivalent to v
- Thus, dc_e right-evaluates to $x \sim_{\text{mem}(M, \text{mem})} v$

$$\langle 3 \rangle 3. M' = M[b \mapsto (0, \text{sizeof}(t_e), 0 \mapsto x)]$$

$$\langle 4 \rangle 1. M' = \text{storeval}(\text{type}(r), M, (b, 0), x)$$

PROOF:

- By $\langle 3 \rangle 1$, the l-value r designates block b
- By $\langle 3 \rangle 2$, the r-value dc_e evaluates to x
- By CLight rule (20), we store the r-value x at the block b designated by the l-value

$$\langle 4 \rangle 2. M' = \text{store}(\text{type}(r), M, b, 0, x)$$

PROOF:

- By $\langle 4 \rangle 2$, $M' = \text{storeval}(\text{type}(r), M, (b, 0), x)$
- As r is a pointer, it is accessed by value
- Thus, by definition of $\text{storeval}(\cdot)$, this corresponds to store x in block b

$$\langle 4 \rangle 3. \text{Q.E.D.}$$

PROOF:

- By $\langle 2 \rangle 1$, $\text{mem}' = \text{mem}[r \mapsto v]$
- By $\langle 3 \rangle 1$, $b \sim_{\text{mem}(M, \text{mem})} r$
- By $\langle 3 \rangle 2$, $x \sim_{\text{mem}(M, \text{mem})} v$
- By $\langle 4 \rangle 2$, $M' = M[b \mapsto x]$
- By (IH) $\langle 0 \rangle 2$, we started from $M \sim_{\text{mem}} \text{mem}$
- Thus, by definition of \sim_{mem} , $M' \sim_{\text{mem}} \text{mem}'$

$$\langle 3 \rangle 4. \text{Q.E.D.}$$

PROOF: By (IH), $M \sim_{\text{mem}} \text{mem}$, by $\langle 2 \rangle 1$, by $\langle 3 \rangle 2$, by $\langle 3 \rangle 3$, and by definition of \sim_{mem}

$$\langle 2 \rangle 7. \text{Q.E.D.}$$

PROOF:

- By $\langle 2 \rangle 4$, $\text{heap}' \sim_{\text{heap}} \text{binding}'$
- By $\langle 2 \rangle 5$, $\text{out} = \text{out}'$
- By $\langle 2 \rangle 6$, $M' \sim_{\text{mem}} \text{mem}'$

$$\langle 1 \rangle 2. \text{CASE: } s = r_1 \leftarrow \text{readRef } r_2$$

PROOF SKETCH: In case of *readRef* statement, the proof is as follows: as usual, we start by evaluating the statement on one hand and compiling/executing it on the other hand. Then, we show that both

outcomes are the same and that the \sim_{mem} and \sim_{heap} relations are preserved. For space reasons, the explanations are less detailed. However, they pretty much follow the ones described in the *newRef* case.

LET:

- $tr_2 = \text{unfoldPointerType}(type)$
- $v_2 = \mathbb{E}[[r_2]]$
- $mem' = mem[r_1 \mapsto mem(l)]$
- $heap' = heap[r_1 \mapsto tr_2]$

$$\langle 2 \rangle 1. \left(\begin{array}{l} mem, \\ heap, \\ r_1 \leftarrow readRef\ r_2 \end{array} \right) \Rightarrow \left(\begin{array}{l} Normal, \\ mem', \\ heap' \end{array} \right)$$

PROOF: By (eval-readRef-anon)

LET:

- $c_{r_2} = \text{toC}_e(r_2)$
- $t_{r_1} = \text{unfoldPointerType}(r_2)$
- $ct_{r_1} = \text{toC}_t(t_{r_1})$
- $binding' = binding[r_1 \mapsto t_{r_1}]$

$$\langle 2 \rangle 2. \left(\begin{array}{l} binding, \\ r_1 \leftarrow readRef\ r_2 \end{array} \right) \xrightarrow{c} \left(\begin{array}{l} binding', \\ \left[\begin{array}{l} local: ct_{r_1}\ r_1 \\ stmts: r_1 = c_{r_2} \end{array} \right] \end{array} \right)$$

PROOF: By (compile-readRef)

LET:

$out' = Normal$

$$\langle 2 \rangle 3. code.global, code.local \vdash (r_1 = c_{r_2}), M \xrightarrow{C} out', M' | \epsilon$$

PROOF: By CLight rule (20)

$$\langle 2 \rangle 4. v_2 = ref\ type\ l\ \text{with } l \in dom(mem)$$

PROOF: By (eval-readRef-anon)

$$\langle 2 \rangle 5. heap' \sim_{\text{heap}} binding'$$

PROOF: By (IH), $heap \sim_{\text{heap}} binding$, extended with equivalent maps.

$$\langle 2 \rangle 6. out = out'$$

PROOF: By $\langle 2 \rangle 1$ and by $\langle 2 \rangle 3$

$$\langle 2 \rangle 7. M' \sim_{\text{mem}} mem'$$

PROOF SKETCH: As in the *newRef* case, the memory equivalence step requires exploring the Clight derivation tree further by first applying rule (20). In particular, the execution of the r-value requires more work.

$$\langle 3 \rangle 1. code.global, code.local \vdash r_1, M \xrightarrow{E} (b, 0)$$

PROOF: By definition of *code.local* at $\langle 2 \rangle 2$

$$\langle 3 \rangle 2. r_2 = ref\ type\ r_2$$

PROOF: By definition of $v_2 = r_2$, by $\langle 2 \rangle 4$

$$\langle 3 \rangle 3. code.global, code.local \vdash c_{r_2}, M \xrightarrow{C} x\ \text{with } x \sim_{\text{mem}(M, mem)} mem(l)$$

PROOF SKETCH: c_{r_2} compiles to distinct terms depending on the type *type* of r_2 . Hence, the following case analysis on this type:

$$\langle 4 \rangle 1. \text{CASE: } type = pointerT\ (pointerT\ type'\ m')\ m$$

PROOF SKETCH: Based on the case assumptions above, we can unroll the Clight derivation tree with rules (1), (2), and (8) up to values that, by the induction hypothesis, are known to be equivalent. Then, we simply show that this equivalence is preserved along the derivations.

$$\langle 5 \rangle 1. t_{r_1} = pointerT\ type'\ Read$$

PROOF: By Case Assumption $\langle 4 \rangle 1$, and by definition of $\text{unfoldPointerType}(\cdot)$

$$\langle 5 \rangle 2. c_{r_2} = *r_2$$

PROOF: By definition of c_{r_2} , by definition of v_2 , by Case Assumption $\langle 4 \rangle 1$, and by definition of toC_e on read pointer

⟨5⟩3. $code.global, code.local \vdash r_2, M \xrightarrow{c} ptr(b_{r_2}, 0)$ with $b_{r_2} \sim_{mem(M, mem)} l$

PROOF: By (IH) ⟨0⟩:2 and CLight rule (1)

⟨5⟩4. $code.global, code.local \vdash *r_2, M \xrightarrow{c} (b_{r_2}, 0)$

PROOF: By ⟨5⟩3 and CLight rule (2)

⟨5⟩5. $p_{r_2} = loadval(type(r_2), M, (b_{r_2}, 0))$

PROOF: By ⟨5⟩4 and CLight rule (8)

⟨5⟩6. $code.global, code.local \vdash *r_2, M \xrightarrow{c} p_{r_2}$

PROOF: By ⟨5⟩5 and CLight rule (8)

⟨5⟩7. Q.E.D.

PROOF: By ⟨5⟩6 and $p_v \sim_{mem(M, mem)} mem(l)$

⟨4⟩2. CASE: $type = pointerT\ type'\ m$

PROOF SKETCH: The proof is similar to the previous case ⟨4⟩1 with the exception that it follows rules (1) and (8).

⟨5⟩1. $t_{r_1} = type'$

PROOF: By Case Assumption ⟨4⟩2, and by definition of $unfoldPointerType(\cdot)$

⟨5⟩2. $c_{r_2} = r_2$

PROOF: By definition of c_{r_2} , by definition of v_2 , by Case Assumption ⟨4⟩2, and by definition of toC_e on reference of base type

⟨5⟩3. $code.global, code.local \vdash r_2, M \xrightarrow{c} (b_{r_2}, 0)$ with $b_{r_2} \sim_{mem(M, mem)} l$

PROOF: By (IH) ⟨0⟩:2 and CLight rule (1)

⟨5⟩4. $p_{r_2} = loadval(type(r_2), M, (b, 0))$

PROOF: By ⟨5⟩3, and CLight rule (8)

⟨5⟩5. $p_{r_2} = load(type(r_2), M, b, 0)$

PROOF: By ⟨5⟩4, by definition of \mathcal{A} , applied on r_2 , passed by value

⟨5⟩6. $code.global, code.local \vdash r_2, M \xrightarrow{c} p_{r_2}$

PROOF: By ⟨5⟩3, by ⟨5⟩5, and CLight rule (8)

⟨5⟩7. Q.E.D.

PROOF: By ⟨5⟩6 and $p_{r_2} \sim_{mem(M, mem)} mem(l)$

⟨3⟩4. $M' = M[b \mapsto (0, sizeof(typeof(r_2)), 0 \mapsto x)]$

⟨4⟩1. $M' = storeval(typeof(r_2), M, (b, 0), x)$

PROOF: By ⟨3⟩1, by ⟨3⟩3, and by CLight rule (8)

⟨4⟩2. $M' = store(typeof(r_2), M, b, 0, x)$

PROOF: By ⟨4⟩1, by definition of \mathcal{A} , applied on c_{r_2} , passed by value

⟨4⟩3. Q.E.D.

PROOF: By ⟨4⟩2, and by definition of $store(\cdot)$

⟨3⟩5. Q.E.D.

PROOF: By (IH), $M \sim_{mem} mem$, by ⟨2⟩1, by ⟨3⟩4, and by definition of \sim_{mem}

⟨2⟩8. Q.E.D.

PROOF:

- By ⟨2⟩5, $heap' \sim_{heap} binding'$
- By ⟨2⟩6, $out = out'$
- By ⟨2⟩7, $M' \sim_{mem} mem'$

⟨1⟩3. CASE: $s = writeRef\ r\ e$

PROOF SKETCH: This proof adopts the by-now standard technique: we evaluate the $writeRef$ statement and also compile/execute it. We easily show that both outcomes are similar and \sim_{heap} is preserved. A little more work is required to show that \sim_{mem} is also preserved.

LET:

- $v_r = \mathbb{E}[r]$
- $v = \mathbb{E}[e]$
- $mem' = mem[r \mapsto v]$

$$\langle 2 \rangle 1. \left(\begin{array}{l} mem, \\ heap, \\ writeRef \ r \ e \end{array} \right) \Rightarrow \left(\begin{array}{l} Normal, \\ mem', \\ heap \end{array} \right)$$

PROOF: By (eval-writeRef-anon)

LET:

- $c_r = toC_e(r)$
- $c_e = toC_e(e)$
- $t_e = \text{typeof}(e)$
- $dc_e = \text{deref}(t_e, c_e)$

$$\langle 2 \rangle 2. \left(\begin{array}{l} binding, \\ writeRef \ r \ e \end{array} \right) \xrightarrow{c} \left(\begin{array}{l} binding, \\ [\text{stmts: } c_r = dc_e] \end{array} \right)$$

PROOF: By (compile-writeRef)

LET:

$out' = Normal$

$$\langle 2 \rangle 3. code.global, code.local \vdash (c_r = dc_e), M \xrightarrow{C} Normal, M' | \epsilon$$

PROOF: By Clight derivation rule (20)

$$\langle 2 \rangle 4. v_r = ref \ type \ l \ \text{with } l \in dom(mem)$$

PROOF: By (eval-writeRef-anon)

$$\langle 2 \rangle 5. heap \sim_{heap} binding$$

PROOF: By (IH)

$$\langle 2 \rangle 6. out = out'$$

PROOF: By $\langle 2 \rangle 1$ and by $\langle 2 \rangle 3$

$$\langle 2 \rangle 7. M' \sim_{mem} mem'$$

PROOF SKETCH: As usual, to solve this goal, we have to go deeper in Clight derivation tree. In particular, we have to execute the right-hand side of the assignment after having followed rule (20).

$$\langle 3 \rangle 1. code.global, code.local \vdash c_r, M \xrightarrow{E} (b_r, 0)$$

PROOF: By (IH) $\langle 0 \rangle 1$, and CLight rule (1)

$$\langle 3 \rangle 2. code.global, code.local \vdash dc_e, M \xrightarrow{C} x \ \text{with } x \sim_{mem(M, mem)} v$$

PROOF SKETCH: As in the *newRef* case, we might write a base type or a pointer. Because these two cases do not compile to the same code, we have to handle them separately.

$$\langle 4 \rangle 1. \text{CASE: } t_e \text{ is an integer or float type}$$

PROOF SKETCH: In this case, both Clight and FoF evaluate dc_e to the same value. The proof is trivial, by application of Clight rules (5) or (6).

$$\langle 5 \rangle 1. v \text{ is an integer or a float}$$

PROOF: By Case Assumption $\langle 4 \rangle 1$

$$\langle 5 \rangle 2. c_e = v$$

PROOF: By $\langle 5 \rangle 1$ and definition of toC_e applied on a base type

$$\langle 5 \rangle 3. dc_e = v$$

PROOF: By $\langle 5 \rangle 2$, by Case Assumption $\langle 4 \rangle 1$, and definition of $\text{deref}(\cdot, \cdot)$

$$\langle 5 \rangle 4. code.global, code.local \vdash dc_e, M \xrightarrow{C} v$$

PROOF: By $\langle 5 \rangle 3$, and CLight rule (5) or (6)

$$\langle 4 \rangle 2. \text{CASE: } t_e \text{ is a pointer}$$

PROOF SKETCH: This case requires further Clight derivation unrolling. However, the derivation path depends on whether v_0 is in *Read* or *Available* mode. Therefore, we make another case analysis.

$$\langle 5 \rangle 1. v = ref \ type \ r_e$$

PROOF: By Case Assumption $\langle 4 \rangle 2$

$$\langle 5 \rangle 2. \text{CASE: } type = pointerT \ type' \ Read$$

PROOF SKETCH: At this stage, we know enough, thanks to the case assumptions, to unroll the Clight derivation tree with rules (1) then (8) until two equivalent Clight and FoF values are reached. Then, we show that the derivations preserve this equivalence.

$$\langle 6 \rangle 1. c_e = *r_e$$

PROOF: By $\langle 5 \rangle 1$ and definition of $\text{deref}(\cdot, \cdot)$

⟨6⟩2. $dc_e = r_e$
 PROOF: By ⟨6⟩1, by Case Assumption ⟨5⟩2, and by definition of toC_e

⟨6⟩3. $\text{code.global}, \text{code.local} \vdash r_e, M \stackrel{\mathcal{E}}{\Leftarrow} (b, 0)$ with $b \sim_{\text{mem}(M, \text{mem})} v$
 PROOF: By (IH), and CLight rule (1)

⟨6⟩4. $p_e = \text{loadval}(\text{type}(r_e), M, (b, 0))$
 PROOF: By ⟨6⟩3 and CLight rule (8)

⟨6⟩5. $p_e = \text{load}(\text{type}(r_e), M, b, 0)$
 PROOF: By ⟨6⟩4, and by definition of \mathcal{A} , applied to a type passed by value

⟨6⟩6. $\text{code.global}, \text{code.local} \vdash dc_e, M \stackrel{\mathcal{E}}{\Leftarrow} p_e$
 PROOF: By ⟨6⟩3, by ⟨6⟩5, and CLight rule (8)

⟨6⟩7. Q.E.D.
 PROOF: By ⟨6⟩6, and $p_e \sim_{\text{mem}(M, \text{mem})} v$

⟨5⟩3. CASE: $\text{type} = \text{pointerT } \text{type}' \text{ Available}$
 PROOF SKETCH: This case is similar to ⟨5⟩2, following rules (1), (9), and (8).

⟨6⟩1. $c_e = r_e$
 PROOF: By ⟨5⟩1, by Case Assumption ⟨5⟩3, and definition of toC_e

⟨6⟩2. $dc_e = \&r_e$
 PROOF: By ⟨6⟩1 and definition of $\text{deref}(\cdot, \cdot)$

⟨6⟩3. $\text{code.global}, \text{code.local} \vdash r_e, M \stackrel{\mathcal{E}}{\Leftarrow} (b, 0)$ with $b \sim_{\text{mem}(M, \text{mem})} v$
 PROOF: By (IH), and CLight rule (1)

⟨6⟩4. $\text{code.global}, \text{code.local} \vdash \&r_e, M \stackrel{\mathcal{E}}{\Leftarrow} \text{ptr}(b, 0)$
 PROOF: By ⟨6⟩3, and CLight rule (9)

⟨6⟩5. $\text{code.global}, \text{code.local} \vdash dc_e, M \stackrel{\mathcal{E}}{\Leftarrow} \text{ptr}(b, 0)$
 PROOF: By ⟨6⟩2, by ⟨6⟩4, and CLight rule (8)

⟨6⟩6. Q.E.D.
 PROOF: By ⟨6⟩5, and $\text{ptr}(b, 0) \sim_{\text{mem}(M, \text{mem})} v$

⟨3⟩3. $M' = M[b \mapsto (0, \text{sizeof}(\text{typeof}(r)), 0 \mapsto v_r)]$

⟨4⟩1. $M' = \text{loadval}(\text{typeof}(r), M, (b, 0), x)$
 PROOF: By ⟨3⟩1, by ⟨3⟩2, and by CLight rule (20)

⟨4⟩2. $M' = \text{load}(\text{typeof}(r), M, b, 0, v_r)$
 PROOF: By ⟨4⟩1, and by definition of \mathcal{A} , applied to type passed by value

⟨4⟩3. Q.E.D.
 PROOF: By ⟨4⟩2 and by definition of $\text{store}(\cdot)$

⟨3⟩4. Q.E.D.
 PROOF: By (IH), $M \sim_{\text{mem}} \text{mem}$, by ⟨3⟩3, and by ⟨2⟩1

⟨2⟩8. Q.E.D.
 PROOF:

- By ⟨2⟩5, $\text{heap}' \sim_{\text{heap}} \text{binding}'$
- By ⟨2⟩6, $\text{out} = \text{out}'$
- By ⟨2⟩7, $M' \sim_{\text{mem}} \text{mem}'$

Bibliography

- [1] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics*, 2005.
- [2] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhanian, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS? In *HotOS*, May 2009.
- [3] Kevin Bierhoff and Chris Hawblitzel. Checking the hardware-software interface in Spec#. In *PLoS '07: Proceedings of the 4th workshop on Programming languages and operating systems*, pages 1–5. ACM, 2007.
- [4] Richard S. Bird. Unfolding pointer algorithms. *J. Funct. Program.*, 11(3):347–358, 2001.
- [5] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transaction on Computer Systems*, 2(1):39–59, February 1984.
- [6] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. *Formal Methods*, pages 460–475, 2006.
- [7] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 2009. Accepted for publication, to appear.
- [8] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.
- [9] David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. *Theorem Proving in Higher Order Logics*, pages 167–182, 2008.
- [10] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. *Programming Languages and Systems*, pages 520–535, 2007.
- [11] Pierre-Evariste Dagand. The design and implementation of reliable operating systems, 2009. <http://perso.eleves.bretagne.ens-cachan.fr/~dagand/fof/biblio.pdf>.
- [12] Pierre-Evariste Dagand. Filet-o-Fish, 2009. <http://perso.eleves.bretagne.ens-cachan.fr/~dagand/fof/FiletoFish.pdf>.
- [13] Pierre-Evariste Dagand. Fugu, 2009. <http://perso.eleves.bretagne.ens-cachan.fr/~dagand/fof/Fugu.pdf>.
- [14] Pierre-Evariste Dagand. Hamlet, 2009. <http://perso.eleves.bretagne.ens-cachan.fr/~dagand/fof/Hamlet.pdf>.
- [15] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. *Verified Software: Theories, Tools, Experiments*, pages 99–114, 2008.
- [16] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, October 2000.
- [17] Jeremy Gibbons. Calculating functional programs. *Algebraic and coalgebraic methods in the mathematics of program construction*, pages 149–201, 2002.

- [18] Paul Hudak. Building domain-specific embedded languages. *ACM Computer Survey*, page 196, 1996.
- [19] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.
- [20] Graham Hutton and Joel Wright. Compiling exceptions correctly. *Mathematics of Program Construction*, pages 211–227, 2004.
- [21] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 275–288. USENIX Association, 2002.
- [22] Simon P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.
- [23] Leslie Lamport. How to write a proof. *Amer. Math. Monthly*, 102(7):600–608, 1995.
- [24] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1):42–54, January 2006.
- [25] Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, July 2008.
- [26] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.
- [27] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.
- [28] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95*, pages 333–343. ACM Press, 1995.
- [29] J. Liedtke. On micro-kernel construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250. ACM, 1995.
- [30] Fabrice Méry, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: an IDL for hardware programming. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 17–30. USENIX Association, 2000.
- [31] George C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5):83–94, 2000.
- [32] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to programmers - taxonomies and characteristics of comments in operating system code. In *Proc. Int'l Conf. on Software Engineering*, pages 331–341, May 2009.
- [33] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, 1998.
- [34] Martin Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, Massachusetts Institute of Technology, March 1999.
- [35] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, June 2008.
- [36] Martin Strecker. Formal verification of a Java compiler in Isabelle. In *CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, pages 63–77, 2002.
- [37] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

- [38] Wouter Swierstra. *A Functional Specification of Effects*. PhD thesis, University of Nottingham, November 2008.
- [39] Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 25–36. ACM, 2007.
- [40] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 207–222. ACM Press, 2003.
- [41] Philip Wadler. The expression problem, 1998. Accessed at <http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt>.
- [42] Maurice Wilkes and Roger Needham. *The Cambridge CAP computer and its operating system (Operating and programming systems series)*. North-Holland Publishing Co., 1979.