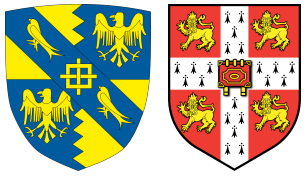
High-level languages for low-level programming

Raphaël Proust

Magdalene College
University of Cambridge



Thesis proposal

followed by

First year report

2013-06-28

In the first year of my PhD, my research focus has shifted from "using linear types to get rid of the garbage collector" to a more important issue: "why can't we use high-level programming languages for low-level programming?". The first part of this document sketches a thesis proposal on this topic (including a detailed explanation of the problem I am trying to solve and a literature review). The second part lists some activities I have pursued during my first year and details my plans for the near future.

Part 1: Thesis proposal

The following is a skeleton version of my PhD thesis as I see it now—obviously subject to major changes.

Chapter 1: Introduction

Computer Science consists of creating abstractions. In the system community there are file systems (to abstract the details of storage management), network stacks (the communication), processes (CPU sharing), &c. In the Programming Language (PL) community we find garbage collectors (GC) (to abstract memory-management), co-routines (concurrency), functions (stack management), reflection (dynamic reconfiguration), loops (control-flow), algebraic data types (ADT) (memory layout), &c. A PL that offers many abstractions is called *high-level*. Similarly we call code that relies on many system abstractions *high-level* code—that is, code that lives high up in the abstraction stack, of which bare metal is the lowest level.

Runtimes Many PL abstractions are provided using *runtime support code* (or simply *runtime code*, or *runtime*; I will use "execution-time" or "dynamic" to talk about the time during which the program is executed). For PLs that compile down to native code, the runtime is code that is meant to be linked against all binaries produced by the compiler. In interpreted and byte-code based PLs the runtime is bundled in the interpreter or the VM.

In many PLs, the distinction between the runtime and the standard library is tenuous. Libraries are included in a PL distribution to avoid code (e.g. sorting routines, widely used data-structure primitives) being written again and again by different programmers. Libraries are collections of functions called explicitly by programmers. On the other hand, runtime code is included to provide some PL features (e.g. memory-management, preemptively-scheduled threads). It is generally written in lower-level PL—most of the time it cannot be self-hosted. Use of runtime code is not explicitly visible in the source code of a program; although someone familiar with a given PL can often make guesses on its usage.

The distinction is further blurred when we consider compatibility and portability code such as OS-agnostic I/O libraries. Compatibility and portability code generally consist of a thin layer that merely translates function calls in the PL into the appropriate system call. It merely "aligns" the current program onto the system it is executed on. Although it adds a layer of indirection, I do not consider it part of the runtime of a PL.

Abstractions Abstractions have a cost. File systems and network protocols are generic and not tailored to one particular application. We cannot expect an abstraction to be optimal with respect to every application that uses it. Similarly on the PL side one cannot expect a general purpose PL to provide the exact abstractions needed for one particular application. These abstractions too have a cost: they reduce performances or impose a certain style to the programmers. This cost is often paid whether the abstractions are actually used or not (e.g. tail-calls are not optimised in Python even when debugging functionalities are not used).

However, abstractions are useful. Chief among their strengths is genericity. It makes code portable (across systems that ship the required set of abstractions) and coders more efficient (by allowing them to apply the same technologies to different projects). It is generally admitted that the portability of C is one of the reasons UNIX got so much traction—even though switching to (what was at the time considered) a high-level PL was controversial. Increasingly numerous and higher abstractions are necessary to effectively build ever-more-complex systems.

Abstraction clash Currently, low-level code (i.e. code that runs in the lower layers of system abstractions, code that runs close to the metal; e.g. code for file systems, network stacks, process schedulers), is written in low-level PLs, in one low-level PL in particular: C[1,2]. The reason C is a good PL for such tasks is twofold: it offers various abstractions (e.g. functions, `unions`, `structs`), *and* these abstractions are provided in a way that does not impede fine-grain control over the dynamic behaviour of programs[3] (e.g. the dynamic layout of `unions` and `structs` in C are simple and very easy to predict whereas ADT in Haskell or ML are not).

In low-level code, it is necessary to take care of some details. In fact, code that takes care of execution-time details is *by definition* low-level code (i.e. code that does not rely on abstractions). It is important to note that low-level code is *necessary*: in order to provide an abstraction for a part of the system, a programmer needs to deal with all the details of this part of the system manually.

[1] C was once considered high-level, it is now on the very bottom of the PL heap just atop assembly.

[2] There are projects of network stacks in higher-level PLs (e.g. FoxNet [1], Mirage [2]); however, these are not widely deployed, use some C internally, and suffer from performance issues.

[3] For additional details, see "the costs of abstraction" talk on http://suckless.org/conference.

When trying to write low-level code in a high-level PL, a very salient problem a programmer is confronted with is the memory representation of data. In most high-level PLs some form of boxing or tagging is used to differentiate values at runtime. The amount of tagging and boxing values are subject to varies from one PL to the next. In a dynamically typed PL with automatic casting and overloading (e.g. Javascript, Ruby) the runtime chooses the appropriate operation to perform for each dynamic application of a given operator. In such a PL, values need to embed enough information for the runtime to reconstruct type information. On the other hand, for the runtime of a statically typed PL without overloading (e.g. ML), type information is not necessary during execution.

For runtime code to work, the compiler changes the shape of values. This prevents the programmer from controlling the memory representation of the values its code handles. Being unable to control the data layout of values is a problem when one needs to interface with the outside world where a specific format is expected (e.g. when preparing network packets for a standard protocol, when formatting data to fit a file-system's specification, &c.).

Consequences Being unable to use a high-level PL to write low-level code is problematic. Low-level PLs offer very little protection against human mistakes programmers are bound to make. It makes OSs unsafe, insecure (buffer-overflows can be prevented through the use of adapted PL abstractions), difficult to maintain (to the point where we use high-level PLs to build tools such as Coccinelle[4] to manage evolvability of big C programs), &c.

I believe that

1) Abstractions are vital to code (for code safety, coders productivity, &c.), and
2) PLs that give control to the programmer are necessary (for file-system code, data OSI layer code, &c.).

Tackling the problem The first part of my attack plan is to create a high-level PL that needs no runtime code[5]. As mentioned previously, runtime code forces the compiler to meddle with the memory representation thus preventing the programmer from controlling it. Removing the need for runtime code tackles the memory representation issue at its source. Note that portability code (including I/O libraries) and general purpose libraries are not problematic for low-level code.

Although inventing a new PL is tempting, I believe there are more benefits to be reaped from removing the runtime code of an existing, established, mature PL:

[4] Coccinelle: http://coccinelle.lip6.fr/.

[5] It is worth noting that C actually has a runtime (libc maintains stateful components such as `errno`), albeit one small enough that it does not intrude on the memory representation of values. A C-like runtime is acceptable for low-level code.

a) some work has already been done by others on the mature PL;
b) adoption (for others and myself) of a known PL's dialect is easier;
c) it forces me to face some research challenges by attacking a "real world" case.

ML is a good candidate because

a) it has been a focus of the PL research community for some time,
b) both FoxNet[1] and Mirage[2] have been written in ML (more specifically SML and OCaml),
c) runtimes for ML variants are reasonably simple,
d) it will be easier to port findings to other ML variants, and
e) it is a PL I would like to write (low-level) code in.

Thus I plan on writing a RunTime-Free ML (RTFML), studying the challenges that are raised along the way, and formalising parts of it.

Chapter 2: Technical background

Reading papers ranging in focus from "type systems" and "compilation" to "networking" and "operating systems" made me understand both the underlying issues that are caused by the use of low-level PLs in system code (lack of safety, verbosity, lack of security), and the fundamental reasons high-level PLs are unfit for system programming (unpredictability, lack of control over memory representation).

Low-level systems using high-level PLs

The FoxNet project [1] aimed at writing a complete network stack in SML. It showed the limits of high-level PLs for the writing of low-level system code. On the one hand, the use of abstractions such as closures and modules (with functors and interfaces) makes code simple, well-organised, readable, and safe. On the other hand, the authors acknowledge a significant amount of time is spent making the code GC-friendly and compensating for the lack of native arrays and unboxed integers.

The Mirage project [2] is a framework that compiles application code directly into OS kernels. The project provides a set of libraries (for networking, file-systems, &c.) linked against the application code during the synthesis of an image. Produced images can be run directly on top of the Xen hypervisor.[6] The project is directly related to my PhD for two essential reasons. Firstly, Mirage removes several layers of system abstractions (run-levels, processes, threads, &c.)

[6] Xen project: http://www.xenproject.org/.

from the execution in order to run the application *as* the OS. This removing of abstraction layers is similar to my PL endeavour. Secondly, because the project is written in OCaml, the synthesised OSs include some runtime code. Mirage is thus both a good project to study runtime code and its effect on low-level code, and also a good candidate to evaluate my findings against.

In both projects the GC is considered to have a negative impact. However, contrarily to popular belief, the GC pauses are not the main problem caused by the GC. The tagging and boxing—needed by the GC to make some sense out of execution-time data—is pointed out by the authors as the main issue.

GC-less memory management

The Lively Linear Lisp dialect [3] showed it is possible to perform garbage collection statically. This PL is a dialect of Lisp that uses strict linear typing to remove the need for garbage collection during execution. Under linear typing discipline, values must be used exactly once. The type system tracks value usage at compile-time, which makes it possible to manage the memory without involving a GC. More precisely, at each use of a value, its memory cell is either mutated in-place or added to a free-cell list. Either case is safe because the value is never to be used again, which is statically ensured by linearity. Additionally, because values *must* be used *exactly* once, memory cannot leak. In Lively Linear Lisp the purpose of linear types is to increase performance (no marking pass, in-place mutability).

Quasi-Linear Types [4] alleviate some of the overly conservative constraints of strict linear types. The type system distinguishes values that are used once (i.e. linearly) from the values that are used repeatedly. This information is inferred from the program source and no annotation is needed at all. A mixed memory-management strategy is applied: linear values are statically managed, non-linear values are garbage collected. Equipped with such a type system, the programmer could choose how much effort he would put into linearising his program.

Region based memory-management [5] is a mechanism in which memory is managed by allocating values in a given region and freeing the whole region as one action. The original region system uses a LIFO approach in which regions are deallocated in reverse order of their allocation. This method of managing memory can sometime cause memory overuse as some deallocations are delayed by other regions being in use. Further work explored the concept of linear regions [6] and showed it is possible to loosen the restrictions on the use of regions. In particular, by threading capabilities associated to each region, it is possible to subsume several region-related memory-management strategies into a single, sound system. This system is used by the Cyclone[7] PL in combination with a traditional, optional GC.

[7] Cyclone is a safe dialect of C: http://cyclone.thelanguage.org/.

Other languages with similar goals and features

ATS[8] (Applied Type-System) is a PL that gives complete control over many aspects of the dynamic behaviour of program (including data representation). Combined with the abstractions it provides (ADTs with pattern matching, first-class functions, &c.), it makes ATS a good fit for the problem I am trying to solve. However, programming for low-level systems in ATS requires the use of a theorem-prover. This is a feature I do not want in RTFML.

Rust[9] provides a mixture of programming paradigms and memory safety. Memory management is achieved through a mixed model where some values are handled by the GC while other are taken care of at compile-time. While a subset of the features of Rust resemble those envisioned for RTFML, the goals are quite different.

Typed Assembly Language[10] (TAL) is a variant of assembly with type annotations and primitives for memory management. TAL is related to RTFML in that it tries to bridge the gap between high-level and low-level PL. TAL takes the dual approach: bring high-level constructs to the bottom of the PL heap.

Chapter 3: Taking back control over memory representation

Bits for the GC

One of the reason ML (and other PLs such as Haskell, Scheme, Lisp) uses tags and boxing is the GC. The GC needs to distinguish different kinds of values to differentiate pointers from immediate values. Thus words (in particular integers) are tagged, structures have a header that makes it possible to identify their shape, and some values are boxed. The GC forces value representation to be uniform; thus the compiler meddles with the memory representation of values.

It is possible to provide tag-free GC with conservative GC technologies. In such a setting, the GC considers every word to be a pointer. However, conservative GCs do not necessarily collect all that can be collected and still induce unpredictability by dynamically introducing arbitrary pauses. These are not problems in user-space applications but are best avoided given our goals.

RTFML features *static* automatic memory management. It provide programmers with a memory abstraction that does not impede low-level programming. In particular, the memory representation is not set by the compiler to suit a dynamic GC. The exact mix of mechanism that will be needed by RTFML to carry this objective is still a matter to be researched. By drawing from both region and linear typing (and other strategies) I believe it is possible to achieve this goal

[8] The ATS Programming Language: http://www.ats-lang.org/.
[9] Rust, a safe, concurrent, practical language: http://www.rust-lang.org/.
[10] Typed Assembly Language: http://www.cs.cornell.edu/talc/.

and eliminate the GC. Maintaining the expressivity of ML under the linearity and region disciplines is also challenging.

Bits for polymorphism

Another part of the runtime code in ML is dedicated to polymorphism. In OCaml, tags are used to dynamically provide polymorphic equality, comparison, hashing, and marshalling. In Haskell, polymorphic functions receive an additional parameter: a dictionary in which to dynamically look for the appropriate monomorphic code. While the former is too intrusive for low-level code, the latter mostly impacts performance.

It is possible to avoid polymorphism impacting the dynamic behaviour of a program altogether. Using whole program monomorphisation a compiler can transform a polymorphic program into a monomorphic one by analysing every call point of every function. Such an analysis requires the entire program to be known, which impedes separate compilation. In a way, it blurs the distinction between compilation and linking—the phase that traditionally resolves the interactions between different compilation units.

Whether whole program monomorphisation is a viable solution for a high-level system PL is yet to be determined.

Chapter 4: Better tags

While tags introduced by the compiler for the use of the runtime code is harmful to low-level code, being able to use one's own tags is essential. They can be used to discriminate unions or manage memory through reference counting. Consider the following extract from a typical Linux's `/usr/include/netinet/in.h` (where the padding length computation has been scrapped):

```
/* Structure describing an Internet socket address.  */
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;                 /* Port number.  */
    struct in_addr sin_addr;            /* Internet address.  */

    /* Pad to size of `struct sockaddr'.  */
    unsigned char sin_zero[<some formula>];
  };

/* Ditto, for IPv6.  */
struct sockaddr_in6
  {
```

```
    __SOCKADDR_COMMON (sin6_);
    in_port_t sin6_port;        /* Transport layer port # */
    uint32_t sin6_flowinfo;     /* IPv6 flow information */
    struct in6_addr sin6_addr;  /* IPv6 address */
    uint32_t sin6_scope_id;     /* IPv6 scope-id */
  };
```

In `/usr/include/sys/un.h`, a `sockaddr` structure is defined for UNIX sockets, and other such structures are defined in different parts of the Linux sources. In all these, the macro `__SOCKADDR_COMMON` expands to a tag field that makes it possible to differentiate dynamically the types (and sizes) of the subsequent fields. In C this tagging of `struct`s in a `union` is idiomatic. The corresponding idiom in ML (and Haskell and other high-level PLs) is to use ADTs. ADTs provide safety guarantees at a price: the programmer cannot control the memory layout of its values.

In order to provide both bit-level control over memory layout and safe, opaque access, RTFML features a different ADT system. The following code illustrates this.

```
type sockaddr =
        | V4 of { tag = tag_v4; port: sin_port; addr: sin_addr; }
        | V6 of { tag = tag_v6; port: sin_port6;
                flowinfo: flowinfo; addr: sin_addr6; }
        with automatic_padding
```

The type declaration includes information about the exact content of the `tag` fields (indicated by the use of the `=` (equal) sign instead of `:` (colon)) and padding. Thus the programmer can communicate to the compiler the shape the data is meant to have. And, in turn, the compiler can translate opaque accesses to fields in the data-type into offsets in memory. Match operations on values of type `sockaddr` are compiled to tests over the tag field and field accesses to direct memory read with offset (e.g. on a MIPS architecture loading a field at offset 2 into register `$t0` is achieved by `lw $t0,4($t1)`).

Other forms of tags can be designed (e.g. encoding the length of a particular field) to allow the programmer to encode more information about the shape of values in the type system. As an interesting illustration of the use of smart tags, consider the following type for UTF-8 runes; it reuses the tags already present in the UTF-8 standard and makes parsing and printing free operations.

```
type rune =
        | R1 of { tag = 1b0; content: bit[7]; }
        | R2 of { tag = 3b110; content_a: bit[5];
                tag = 2b10; content_b: bit[6]; }
        | R3 of { tag = 4b1110; content_a: bit[4];
```

```
                tag = 2b10; content_b: bit[6];
                tag = 2b10; content_c: bit[6]; }
        ...
```

This example uses a syntax for binary tags of specific bit-length: `3b110` is a 3-bit long field containing the bits `110`. (The concrete syntax of RTFML is not yet fixed.) Also note that some record have multiple `tag` fields. This reflects the UTF-8 standard and its self-synchronising property. Because tag fields are not meant to be accessed, this is not a problem. Indeed, the variants (here `R1`, `R2`, `R3`...) are syntactically sufficient for `match` constructs as demonstrated by the following code.

```
let int32_of_rune =
        | R1 of { content; } ->
                let i = BitArray.make 32 0 in
                BitArray.blit
                        content 0
                        i (32 - BitArray.length content);
                (i : int32)
        | R2 of { content_a; content_b; } ->
                let i = BitArray.make 32 0 in
                BitArray.blit
                        content_a 0
                        i (32 - BitArray.length content_a);
                BitArray.blit
                        content_b 0
                        i (32 - BitArray.length content_a
                                - BitArray.length content_b);
                (i : int32)
        | R3 of { content_a; content_b; content_c; } ->
                ...
        ...
```

It is dangerous in such settings to try to make the PL's type system as powerful as possible, getting enticed into ever-richer dependent types. The aim of RTFML is practical and thus I will not delve too far into refinements of the type system.

Chapter 5: Applications/evaluation

Looking at the use that is made of the runtime-free PLs (assembly, C) and PLs with a small runtime (Rust, C++), we can evaluate RTFML on similar cases: runtimes, network code, fast variable-length encoding libraries, &c.

- A new runtime for OCaml: the OCaml runtime mostly deals with data layout—checking for particular headers to determine the shape of a given

value and taking appropriate action. In RTFML the data layout can be specified and can be made to match existing data-structure. Thus it is possible to work on OCaml values as RTFML values.

- Inter-PL bindings: this has to do with changing data layout and moving and copying memory.
- Parser-less UTF-8 library: by encoding the UTF-8 layout in the type system, both scanning and outputting code-points become simple.
- Parser-less, zero-copy network stack: more difficult than UTF-8 encoding is the description of network packets in the type system. While it is not clear whether a complete description is possible in a simple type system, RTFML type system might still be helpful to describe important invariants of the network stack.

While the design of RTFML is goal oriented, it is worth noting that some design choices have important side-effects. Linear typing is known to be useful for helping to deal with concurrency [7, 8], integrating side-effects in purely functional settings [9, 10], and enabling some optimisations. These additional features of RTFML could also be studied.

Chapter 6: Conclusion

This section will be written in two years' time.

Part 2: First year report

I spent a part of the first year of my PhD reading papers and discussing with colleagues. This shaped my perception of the problem exposed in the thesis proposal. A more detailed report follows and a second section lists on going work.

Past activities

Here are listed activities I pursued during the first year of my PhD.

Reading

Other papers—than those listed in the Technical Background section—I have been reading have topics that range from

- type system and in particular their use for cache-friendliness and better concurrency: [11–13], to
- garbage collection and other memory management strategies: [14, 15], to
- other uses of linear typing: [16].
- And miscellaneous papers and other writings on PL, compilation, hardware, and the plan9 OS.

I was appointed to review papers for the EuroSys "Shadow PC". Having never seen a PC function, this insight into the reviewing and selection processes was interesting. EuroSys is a system conference and soliciting papers related to networking, file-systems, and operating system safety. These areas are connected to the "low-level programming" end of my PhD.

I have also been interested in the ideas relayed by the suckless community[11]. It contributed to my view of how software should be coded and how PLs should be used.

Lab life

I have been involved in OCaml Labs, mostly discussing ideas related to concurrent garbage collection and syntax extension. I also helped review the "real world OCaml" book and contributed lightly to the general eco-system (integrating TypeRex[12] in Vim and Acme, reporting bugs in opam[13]).

I have supervised the following courses:

[11] `suckless.org`, software that sucks less: http://suckless.org/philosophy.
[12] TypeRex, the OCaml Programming Studio: http://www.typerex.org/.
[13] OCaml Package Manager: http://opam.ocamlpro.com/.

- Compiler Construction, Optimising Compilers,
- Semantics, Types, and
- Concepts in Programming Languages.

All of these are relevant to my PhD and served as useful reminders of courses I have followed previously at other universities. Semantics and Types are both central to the tracks I want to pursue. Compiler Construction and Optimising Compilers will both be useful when I write the first draft of my compiler.

I organised the visit by Jonathan Protzenko and Francois Pottier to CPRG. They gave a talk about Mezzo (a PL they are working on). More interestingly for me, we had a long discussion (also present were Alan Mycroft and Anil Madhavapeddy) about the similarities and differences between Mezzo and RTFML. While pursuing a different goal, some of the technologies they use are very similar, notably encoding some dynamic properties of the heap in the type system.

Training

I attended the 2013 edition of "École des Jeunes Chercheurs en Programmation" organised by INRIA. Of particular interest to me were courses on *why3* and *coq* which I might make use of to formalise (parts of) RTFML. Other courses included work on *scala*, *pharo*, and *hop*, and about static analysis and formalisation of security protocols.

Miscellaneous

I have been a member of the MCR committee of Magdalene College and was recently appointed to the executive position of Secretary.

I am currently building a network of Raspberry Pis[14] (RPi) with friends of mine. Each RPi is stored at a friend's house, connected to the internet. The aim of the network is to provide distributed data-backup and sharing/collaborating space. Backed-up data is available both at one's own place (on one's own RPi) (thus providing a local cache) and at others (thus providing off-site backup). Challenges include: restoring end-to-end connectivity (punching through NATs, having a stable addressing mechanism, &c.) and setting up versioning, distribution, and encryption of files. Physically storing your data at friends' makes more sense than "in the cloud" for privacy concerns. Once deployed we want to document the inner workings and setting-up process of the network.

Present and foreseeable future

Ongoing work and plans for the near future are listed here with time estimates.

[14] Raspberry Pi: http://www.raspberrypi.org/.

Prototype compiler

I have been working on a prototype compiler for RTFML for a few months, thus far focusing on architectural concerns (i.e. intermediate representations (IR) and phases). I will soon write a prototype implementation, after which I will go through a phase of incremental build by fixpointing the following design process:

a) writing a toy example,
b) adding support for any missing features to the compiler for the toy example to compile, and
c) tidying up the code.

Taking Hofstadter's law[15] into account this phase should extend to October. This incomplete compiler will then be improved into a demo-able state. This last step can be done by writing a few small-yet-real-life examples (e.g. UTF-8 handling). This second phase does not deal with the core part of the compiler but all the details around it. It will likely take as much time.

I have always been curious about compiler design and have recently been playing with the idea of UNIX-ier compilers (i.e. compilers that follow more closely the UNIX philosophy). In such a compiler, each pass would be like a UNIX filter: one pass should do one thing (and do it well), passes should be composable, all intermediate representations (IR) should share a common set of features[16] (e.g. an IR should be parsable, printable, and interpretable).

Interesting questions on the matter include

- How to not get too verbose with the plethora of intermediate representations?
- How to keep to the DRY (Don't Repeat Yourself) philosophy in such a compiler?
- What performances can we expect from such a compiler?
- What granularity is optimal? (I.e. what is "one thing" that a pass does.)
- How to make independent passes commute?

Because these are tangential questions I will not dwell on these matters for too long during my PhD.

"Better tags" formalisation

Another endeavour I will be focusing on during the upcoming summer is formalising the "Better tags" mentioned in the thesis proposal. Having a type system

[15] Hofstadter's Law: "It always takes longer than you expect, even when you take into account Hofstadter's Law".

[16] Just as text is a uniform representation of the content that is pushed through pipes in UNIX-y systems, the IRs should have some form of uniformity.

based on bit-level representation of data seems a useful feature for low-level programming. An abstract from a yet-to-be-written paper follows.

> Different PLs offer different level of control over memory representation. Languages such as C offer great control but impose great responsibilities on the programmer. At the other end of the spectrum, PLs such as Java, ML, and Haskell offer very little control (most values are boxed, integers are tagged, &c.) but guarantee some form of memory safety. Aside this spectrum is ATS, which offers the best of both worlds but requires the programmer to go through a theorem prover to ensure the correctness of its program. We believe there is space in the middle for a PL that allows fine-grained control over memory representation and guarantees safety without the hassle of a theorem prover.
>
> We present a type system that focuses on bit-level memory representation. Borrowing ideas from dependent typing we encode enough information in the representation of values to have complete memory safety. Finally we sketch the interaction of our type system with different memory-management technologies.

This part is very synergistic with the aforementioned compiler prototyping as the bit-level type system is to be included in RTFML from the start. Early experiments with the IRs of the compiler were performed with a first-draft, non-formalised version of this type system. For this reason, the paper will probably also be ready around October.

Taglessness papers

Work on removing the GC from ML being central to my PhD, it will also be the topic of at least one paper. Similarly, evaluation of the viability of whole-program monomorphisation will need to be studied at some point and should result in academic writings. These papers are subject to a less precise time line.

References

[1] Biagioni, E. et al. 2001. A Network Protocol Stack in Standard ML. *Higher Order Symbol. Comput.* 14, 4 (dec. 2001), 309–356.

[2] Madhavapeddy, A. et al. 2013. Unikernels: library operating systems for the cloud. (2013), 461–472.

[3] Baker, H.G. 1992. Lively linear Lisp: "Look Ma, No Garbage!" *SIGPLAN Not.* 27, 8 (aug. 1992), 89–98.

[4] Kobayashi, N. 1999. Quasi-linear types. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1999), 29–42.

[5] Tofte, M. et al. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation Journal.* 17, (2004), 245–265.

[6] Fluet, M. et al. 2006. Linear Regions Are All You Need. *In Proc. ESOP'06* (2006), 7–21.

[7] Srinivasan, S. and Mycroft, A. 2008. Kilim: Isolation-Typed Actors for Java. *Proceedings of the 22nd European conference on Object-Oriented Programming* (Berlin, Heidelberg, 2008), 104–128.

[8] Richard, R.E. et al. 2004. Linear Types for Packet Processing. *In Proceedings of the 13th European Symposium on Programming* (2004), 204–218.

[9] Clean: I/O on the Unique World: http://clean.cs.ru.nl/download/html_report/CleanRep.2.2_4.htm _Toc311797987.

[10] Mercury: I/O library: http://www.mercurylang.org/information/doc-release/mercury_library/io.html io.

[11] Mycroft, A. 2012. Isolation types and multi-core architectures. *Proceedings of the 2011 international conference on Formal Verification of Object-Oriented Software* (Berlin, Heidelberg, 2012), 33–48.

[12] Reynolds, J. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. (2002), 55–74.

[13] Boyland, J. et al. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. *Proceedings of the 15th European Conference on Object-Oriented Programming* (London, UK, UK, 2001), 2–27.

[14] Optimizing Allocation and Garbage Collection of Spaces: http://home.pipeline.com/~hbaker1/OptAlloc.html

[15] Jim, T. et al. 2002. Cyclone: A Safe Dialect of C. *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2002), 275–288.

[16] Berdine, J. et al. 2002. Linear Continuation-Passing. *Higher Order Symbol. Comput.* 15, 2-3 (sep. 2002), 181–208.

Colophon

Thanks are due to friends (especially the non-tech-savy ones) and colleagues for linguistic and technical feedback.

This document was edited in markdown format in `acme`, and, driven by `mk`, processed succesively by `pandoc` and `pdflatex`. Crests on the title page are from Wikimedia Commons under Creative Commons Attribution-Share Alike license.