DRAFT: SUBMITTED FOR REVIEW TO PLACES 2012

Limel: Local Computation with Linear Coordination

Raphaël Proust
University of Cambridge
ÉNS de Cachan
rp452@cl.cam.ac.uk

Julien Verlaguet
Facebook, Inc.
julien.verlaguet@gmail.com

Anil Madhavapeddy
University of Cambridge
avsm2@cl.cam.ac.uk

Abstract

Limel is a new systems programming language for constructing efficient, scalable data processing pipelines. Code is written in the style of ML, and the compiler statically specialises the output for execution on either multi-core systems (with call-by-reference semantics) or distributed clusters (with call-by-value semantics), or a combination of both. No modification to the source code is required to swap between calling conventions. Linear typing eliminates the need for a garbage collector, and whole-program monomorphisation means that memory values are not tagged, making the foreign function interface trivial.

The big challenge with integrating linear types in day-to-day programming languages is one of usability and expressivity. In this work-in-progress paper, we describe the Limel type system and semantics, and discuss the lessons learnt so far from applying it to various problems such as functional data structures and protocol parsing.

1 Introduction

A modern datacenter can have thousands of machines that move terabytes of data between themselves, and each host can have multiple cores that need to be filled with parallel tasks. There are usually different programming models when dealing with a distributed cluster versus a single shared-memory multicore host. A host can pass a reference to a value to another core, and rely on a runtime garbage collector to manage the lifetime of the values. The same function call to a remote machine must copy the value over the network, and any changes be manually synchronised back to the original host. Some newer multicore chips lack coherent shared memory, and blur the distinction between host and cluster computing.[1]

Conventional general-purpose languages force the programmer to make a choice between which mechanism to use for efficiency reasons, and it can be difficult to switch between shared and non-shared memory models. Haskell, for example, has sophisticated shared-memory parallelisation support [7], but switching the same program to a distributed version requires much more work [3]. In this paper, we describe a new language—Limel—that can switch between call-by-value and call-by-reference semantics at compile time (§3.2). It is based on the familiar ML syntax and semantics, with linear types to statically track value aliasing.

The focus of Limel is the *type-safe coordination of complex dataflow pipelines*, such as those found in modern datacentres. We combine linearity with monomorphization to eliminate most runtime support needed by conventional languages: there is no garbage collector and heap values map to a tag-less C memory representation (§2.2), making Limel suitable for use in constrained environments such as kernels.

2 The Limel Language

Linear types are based on linear logic [4], and results in resource-aware languages. Values cannot be discarded nor duplicated silently and must be used exactly once. This is much more restrictive than most conventional type systems.

[1] The Intel SCC is one such system. See: http://techresearch.intel.com/ProjectDetails.aspx?Id=1

```
let c0 = new_collection () in
let (e1,c3) =
  let (c1,empty) = isEmpty c0 in
  if empty then
   (1, c1)
  else
   let (e0,c2) = take_one_element c1 in
   (e0+1, c2)
in
do_something e1 c3
```

```
let c = new_collection () in
let e =
  if isEmpty ?c then
    1
  else
    take_one_element !c + 1
in
do_something e c
```

Figure 1: Linearly threading values so each is used exactly once *(left)*, and the same logic with the more succinct Limel read/write aliasing operators *(right)*

Limel is a linearly-typed ML-like language with a focus on runtime minimalism. While other languages use linear types—e.g. Clean and Mercury for tracking side effects, and Kilim [9] for message-passing over shared memory—none use them pervasively. In Limel every heap value is linear (integers and booleans are not heap-allocated and thus non-linear). Closures that hold a pointer to a linear value—e.g. through partial application—are also linear.

Figure 1 *(left)* shows a Limel fragment. Notice that no value name is ever re-used, and a value is explicitly returned if it is further needed after being passed to a function. This highlights one of the drawback of linearity: values must be threaded along the computation to allow later access to them. Figure 1 *(right)* shows a more succinct version of the same code using the Limel alias types (§2.1). Another consequence of linearity is the memory layout that can only be tree- or forest-like with roots in the stack and registers [5]. This reduces the set of data structures that can be expressed directly in Limel (§2.2).

2.1 Careful Aliasing: Safely Breaking Linearity

Our solution to the verbosity issue is inspired by Odersky's observers [8]. While observers allow scope-controlled, read-only, non-linear access to values, we also provide in-place modification access. Values are either linear or *read-aliased* (as Odersky's observers) or *write-aliased*. Write aliases can be modified in-place but not consumed nor freed. The only construct that allows aliasing is function application, by prefixing an argument with ? to mark a read alias and ! to mark a write alias.

The types are formalised in Figure 2 with judgements of the form $\Gamma \vdash e : \tau$ **leaving** $\Gamma' \, \Gamma_r \, \Gamma_w$ where Γ is the environment[2] before e, τ the type of e, Γ' the environment after e (what is left), Γ_r the set of variables that are read-aliased in e, and Γ_w the set of variables that are write-aliased in e. The type of function arguments can be plain (τ) or qualified (τ reader or τ writer). The following invariants are respected: $\Gamma' \subseteq \Gamma$, as well as $\Gamma_r \cap \Gamma_w = \emptyset$, and $\Gamma_r, \Gamma_w \subseteq \Gamma'$.

If a value is both read- and write-aliased within the same expression (as constructed by `let-in` or the ; sequencing operator) the variable is considered write-aliased. A write-alias can safely be passed to a function that expects a read-alias, since owning a write-alias to a variable has stronger constraints than holding a read-alias to it. The E_FUNCTION_RW rule forbids the partial application of aliases. In the E_FUNCTION rule, the constraint on τ makes τ a plain value or a linear closure but never a non-linear function. All functions must actually use all their arguments. Some of these rules are overly conservative, and we plan to relax them as the language matures.

[2]We consider programs after α-renaming such that each identifier is unique.

$$\frac{(i:\tau)\in\Gamma \quad \neg(\tau \text{ is linear})}{\Gamma \vdash i:\tau \textbf{ leaving } \Gamma\,\emptyset\,\emptyset}\ \text{E\_IDENT\_NONLIN}$$

$$\frac{(i:\tau)\in\Gamma \quad \tau \text{ is linear}}{\Gamma \vdash i:\tau' \textbf{ leaving } (\Gamma\setminus\{i:\tau\})\,\emptyset\,\emptyset}\ \text{E\_IDENT}$$

$$\frac{\Gamma,\{i:\tau'\}\vdash e:\tau \textbf{ leaving } \Gamma\,\emptyset\,\emptyset \quad \neg(\tau \text{ is a non-linear function})}{\Gamma \vdash \textbf{fun} i\to e:\tau'\to\tau \textbf{ leaving } \Gamma\,\emptyset\,\emptyset}\ \text{E\_FUNCTION}$$

$$\frac{\begin{array}{l}\Gamma,\{i:\tau'\}\vdash e:\tau \textbf{ leaving } (\Gamma,\{i:\tau'\})\,\Gamma_r\,\Gamma_w \\ \{i:\tau'\}=(\Gamma_r\cup\Gamma_w) \\ ((i:\tau')\in\Gamma_w)\Rightarrow(\delta=\text{writer}) \\ ((i:\tau')\in\Gamma_r)\Rightarrow(\delta=\text{reader}) \\ \neg(\tau \text{ has kind } \to)\end{array}}{\Gamma\vdash \textbf{fun} i\to e:\tau'\,\delta\to\tau \textbf{ leaving } \Gamma\,\emptyset\,\emptyset}\ \text{E\_FUNCTION\_RW}$$

$$\frac{\begin{array}{l}\Gamma\vdash e_1:\text{unit} \textbf{ leaving } \Gamma_1\,\Gamma_{r1}\,\Gamma_{w1} \\ \Gamma_1\vdash e_2:\tau \textbf{ leaving } \Gamma_2\,\Gamma_{r2}\,\Gamma_{w2} \\ \Gamma_w=\Gamma_{w1}\cup\Gamma_{w2} \\ \Gamma_r=(\Gamma_{r1}\cup\Gamma_{r2})\setminus\Gamma_w\end{array}}{\Gamma\vdash e_1;e_2:\tau \textbf{ leaving } \Gamma_2\,\Gamma_r\,\Gamma_w}\ \text{E\_SEQ}$$

$$\frac{\begin{array}{l}\Gamma\vdash e_1:\tau_1 \textbf{ leaving } \Gamma_1\,\Gamma_{r1}\,\Gamma_{w1} \\ (\Gamma_1,\{i:\tau_1\})\vdash e_2:\tau \textbf{ leaving } \Gamma_2\,\Gamma_{r2}\,\Gamma_{w2} \\ \neg((i:\tau_1)\in\Gamma_2) \\ \Gamma_w=\Gamma_{w1}\cup\Gamma_{w2} \\ \Gamma_r=(\Gamma_{r1}\cup\Gamma_{r2})\setminus\Gamma_w\end{array}}{\Gamma\vdash \textbf{let}\, i=e_1 \textbf{ in } e_2:\tau \textbf{ leaving } \Gamma_2\,\Gamma_r\,\Gamma_w}\ \text{E\_LETIN}$$

$$\frac{\begin{array}{l}\Gamma\vdash e:\text{bool} \textbf{ leaving } \Gamma'\,\Gamma_r{}'\,\Gamma_w{}' \\ \Gamma'\vdash e_1:\tau \textbf{ leaving } \Gamma_2\,\Gamma_{r1}\,\Gamma_{w1} \\ \Gamma'\vdash e_2:\tau \textbf{ leaving } \Gamma_2\,\Gamma_{r2}\,\Gamma_{w2} \\ \Gamma_w=\Gamma_w{}'\cup(\Gamma_{w1}\cup\Gamma_{w2}) \\ \Gamma_r=(\Gamma_r{}'\cup(\Gamma_{r1}\cup\Gamma_{r2}))\setminus\Gamma_w\end{array}}{\Gamma\vdash \textbf{if}\, e \textbf{ then } e_1 \textbf{ else } e_2:\tau \textbf{ leaving } \Gamma_2\,\Gamma_r\,\Gamma_w}\ \text{E\_IF}$$

$$\frac{\begin{array}{l}\Gamma\vdash e_1:\tau_1 \textbf{ leaving } \Gamma_1\,\Gamma_{r1}\,\Gamma_{w1} \\ \Gamma_1\vdash e_2:\tau_1\to\tau \textbf{ leaving } \Gamma_2\,\Gamma_{r2}\,\Gamma_{w2} \\ \Gamma_w=\Gamma_{w1}\cup\Gamma_{w2} \\ \Gamma_r=(\Gamma_{r1}\cup\Gamma_{r2})\setminus\Gamma_w\end{array}}{\Gamma\vdash e_2\,e_1:\tau \textbf{ leaving } \Gamma_2\,\Gamma_r\,\Gamma_w}\ \text{E\_APP}$$

$$\frac{\begin{array}{l}\Gamma\vdash i:\tau_1 \textbf{ leaving } \Gamma_1\,\emptyset\,\emptyset \\ \Gamma_1\vdash e:\tau_1\,\text{reader}\to\tau \textbf{ leaving } \Gamma_2\,\Gamma_{r2}\,\Gamma_{w2} \\ \neg((i:\tau_1)\in\Gamma_1)\end{array}}{\Gamma\vdash e\,?\,i:\tau \textbf{ leaving } (\Gamma_2,\{i:\tau_1\})\,(\Gamma_{r2},\{i:\tau_1\})\,\Gamma_{w2}}\ \text{E\_READ}$$

$$\frac{\begin{array}{l}\Gamma\vdash i:\tau_1 \textbf{ leaving } \Gamma_1\,\emptyset\,\emptyset \\ \Gamma_1\vdash e:\tau_1\,\delta\to\tau \textbf{ leaving } \Gamma_2\,\Gamma_{r2}\,\Gamma_{w2} \\ \neg((i:\tau_1)\in\Gamma_1) \\ (\delta=\text{reader})\vee(\delta=\text{writer})\end{array}}{\Gamma\vdash e\,!\,i:\tau \textbf{ leaving } (\Gamma_2,\{i:\tau_1\})\,\Gamma_{r2}\,(\Gamma_{w2},\{i:\tau_1\})}\ \text{E\_WRITE}$$

Figure 2: Limel typing rules.

2.2 Foreign Function Interface: Unsafely Breaking Linearity

The tree-like restriction on data structures is much more serious. Consider two threads that need to communicate in a shared memory setting: while their messages can be linear, a data structure that holds them cannot. Both threads need to explicitly access this buffer, hence the need for aliasing. It is possible to solve this case by adding a primitive to the language (with buffers exporting two distinct handles), but it highlights the need for non-linear constructs.

We currently support such constructs via a C foreign function interface (FFI), at the expense of type-safety for communication channel implementations. Approaches we are considering include the more practical forms of affine types [10] or dependent types [1], while preserving the runtime minimalism of the current Limel system.

Limel memory management is static, with the type system responsible for checking the absence of memory leaks, thus avoiding the need for a garbage collector. The idea of removing the garbage collector from the runtime stems from Lafont's Linear Abstract Machine [5].

Conventional polymorphism still requires heap values to be tagged to distinguish between them at runtime. We eliminate the need for tags by performing monomorphisation. This approach is seldom used because of the need for the whole program to be compiled in one go. Monomorphisation happens quite late in Limel however: compilation units can be separately type checked (very useful during development) and code-generation happens in one phase during link-time. This is similar to link-time optimisations in recent GCC and LLVM where, by considering the whole program, cross module inlining is made possible.

Using monomorphisation in combination with static memory management is a powerful tool for systems programming as it eliminates the need for a significant language runtime, and makes it possible for Limel values to be exchanged with C directly.

3 Limel Usage

The combination of a restrictive type system with the ease of calling foreign languages might discourage programmers from actually using Limel! We here present some use cases where a fully linear language is greatly beneficial.

3.1 Functional Linear Parsing

In a data-flow engine, the execution is directed by input data which is parsed, classified, and transmitted to the next processing block. A modern OS network stack is similar: packets are parsed and demultiplexed through the stack (e.g. TCP/IP). Limel aims to provide high-level, functional mechanisms—such as pattern matching, safety guarantees, polymorphism and higher-order functions—into the parsing realm, but with C-like performance.

I/O comes into Limel as a fresh value, and is parsed and processed directly. While low-level packet parsing is fairly straightforward, we are also building a combinator parser in Limel that should result in a high-performance parser for textual protocols, without the usual overheads of closure allocations and value copying. When more expressivity is absolutely needed, a foreign call is made into another language runtime (such as OCaml). When that runtime terminates and releases the value, the Limel program continues processing. Thus, small patches of non-linearity can exist in an otherwise linear data-flow graph, with value aliasing restricted to a local part of the computation.

3.2 Heterogenous Architecture Mapping

Call-by-value (CBV) and call-by-reference (CBR) differ from a semantics perspective—in CBR the caller and callee can influence each other by modifying the argument directly—as well as from a implementation point of view—CBR requires some shared memory to be available.[3] The most performant calling convention depends on the hardware platform a program runs on, but the different semantics prevents transparently switching from CBV to CBR. In a linear language the semantic difference between CBV and CBR is not relevant: whenever an argument is transmitted, the caller gives up the right to access it. Thus, the compiler is free to choose which calling convention is most appropriate.

When using explicit aliasing, changing the calling convention is still simple, and only CBV with a write-alias needs special handling as the modifications have to be propagated back to the caller. We are currently exploring the possibility of having "relaxed write aliases" that

[3] While it is possible to emulate shared memory over a network, this has an important performance cost.

match the `x86` relaxed memory model (where writes may be arbitrarily delayed), as this is also a better fit to distributed systems where local read caches will outpace remote writes.

A lot of the effort of building network stacks goes into maintaining high throughput with heterogeneous hardware, such as network processors. High-speed processing requires static scheduling into a fast data path for most packets, and a slower one for out-of-band processing (e.g. IP options). Limel can use linearity to safely remap processing functions onto such heterogeneous hardware without having to rewrite the source code.

This calling convention indifference was used in PacLang [2], which supports a per-architecture specialisation of packet transfers from one thread to another. While PacLang was designed for packet processing, and thus targeted at specialised network processors, we want Limel to be useful for higher-level systems such as web servers and distributed databases, as well as cloud operating systems like Mirage [6].

4 Conclusion

Linear types offer many opportunities for programming modern multicore and distributed systems. While restrictive for some problems, they allow natural encoding of others (such as data processing or some scientific computation). Once identified, these embarrassingly linear problems can benefit from a language such as Limel, especially when they need to run on heterogeneous hardware platforms. We would like to thank Scott Owens for his `ott` help, and Alan Mycroft, Simon Peyton-Jones, Don Syme and Richard Mortier for useful discussions.

References

[1] Edwin C. Brady. Idris — systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 43–54, New York, NY, USA, 2011. ACM.

[2] Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In David A. Schmidt, editor, *13th European Symposium on Programming (ESOP), part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, volume 2986 of *Lecture Notes in Computer Science*, pages 204–218, Barcelona, Spain, April 2004. Springer.

[3] Jeff Epstein, Andrew Black, and Simon Peyton-Jones. Towards Haskell in the cloud. In *Proceedings of the Haskell Symposium*, September 2011.

[4] Jean-Yves Girard. Light linear logic, 1995.

[5] Y. Lafont. The linear abstract machine. *Theor. Comput. Sci.*, 59:157–180, July 1988.

[6] Anil Madhavapeddy, Richard Mortier, Ripduman Sohan, Thomas Gazagnaire, Steven Hand, Tim Deegan, Derek McAuley, and Jon Crowcroft. Turning down the LAMP: Software specialisation for the cloud. In *2nd USENIX Workshop on Hot Topics in Cloud Computing*, June 2010.

[7] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[8] Martin Odersky. Observers for linear types. In *Proceedings of the 4th European Symposium on Programming*, pages 390–407, London, UK, 1992. Springer-Verlag.

[9] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, 2008.

[10] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 447–458, New York, NY, USA, 2011. ACM.