Linear Calculi for Programming Languages: a comparison approach

Ana Jorge Almeida, Sandra Alves, and Mário Florido

LIACC, Departamento de Ciência de Computadores
Faculdade de Ciências, Universidade do Porto
Rua do Campo Alegre, s/n, Porto, Portugal

Abstract. Here we define and implement a compilation between two linear systems, a linear calculus from David Walker (2005) and Linear Haskell (2017) and show that, although based on different semantics, there exists an equivalence between both systems. There are some difficulties to this though, given that one calculus uses a small-step semantics and the other a big-step semantics and Linear Haskell uses let-statements in addition to the terms that can be directly translated from each system. We will make clear those differences and show how to deal with them in the compilation process.

1 Introduction

A linear function consumes its argument exactly once and a linear type system gives a static correctness guarantee that a function is linear. Linear types originate in a theory of linear logic [6], where truth is a finite resource that can be used once, but not duplicated or discarded.

Several linear calculi were defined before as foundational calculi for programming languages [1–5, 7, 9–11]. In this paper we focus on two of these calculi and compare them with respect to their operational semantics: the linear calculus presented by David Walker in the 2005 chapter *Substructural Type Systems* of the book *Advanced Topics in Programming Languages* [11] and the more hype 2017 Haskell extension *Linear Haskell* [3].

In [11], David Walker presents a linear type system that ensures that we can safely deallocate an object after being used. He also defines an operational semantics that evaluates terms in an abstract machine with an explicit store using a context-based, small-step semantics.

More recently, Linear Haskell, an extension to Haskell that uses linear types [3], is build upon a big-step operational semantics with laziness in the style of Launchbury semantics [8], where terms are transformed before evaluation [3].

Here we define a compilation between these two systems and with this show that, although based on different semantics (small-step and big-step) there exists an equivalence between both systems. As an example, consider the following term of Linear Haskell: $\text{let}_\omega\ x : \alpha \to_\omega \alpha = \lambda_\omega y : \alpha.y \text{ in } < x, x >_1$ where subscripts ω means unrestricted use and 1 linear use. Our compilation translates the previous term to $(\text{lin} \ < \text{un}\ \lambda y : \text{un}\ \alpha.\ y, \text{un}\ \lambda y : \text{un}\ \alpha.\ y >)$ in the David Walker calculus, where *lin* means linear use and *un* means unrestricted use. Below are the evaluations of each term, first in Linear Haskell:

$$\frac{(\{x :_\omega = \lambda_\omega y : \alpha.y\}; < x, x >_1) \Downarrow (\{x :_\omega = \lambda_\omega y : \alpha.y\}; < x, x >_1)}{(\emptyset; \text{let}_\omega\ x : \alpha \to_\omega \alpha = \lambda_\omega y : \alpha.y \text{ in } < x, x >_1) \Downarrow (\{x :_\omega = \lambda_\omega y : \alpha.y\}; < x, x >_1)}$$

and, in the David Walker calculus:

$$\begin{aligned}&(\emptyset;\ \text{lin } < \text{un } \lambda y : \text{un } \alpha.\ y, \text{un } \lambda y : \text{un } \alpha.\ y >)\\ \to\ &(a \mapsto \text{ lin } < \text{un } \lambda y : \text{un } \alpha.\ y, \text{un } \lambda y : \text{un } \alpha.\ y >; a)\end{aligned}$$

There are intrinsic difficulties on the definition of a correct compilation, given that one calculus uses a small-step semantics and the other a big-step semantics and Linear Haskell uses let-statements in addition to the terms which can be directly translated from each system. We will make clear those differences and how to deal with them in the compilation process. All definitions were implemented in Haskell. The implementation and an extended version with auxiliary lemmas and full proofs are available in `https://github.com/anathegrey/Linear-Calculi-Translation/`.

2 Relation between Linear Haskell and David Walker calculus

Our main goal is to define a translation to show that the David Walker calculus [11], $\mathcal{W}$, and Linear Haskell [3], $\mathcal{L}$, are equivalent, which means that we want to show they both yield the same results at the end of computation, despite their differences in syntax. Therefore, we define translations between terms, types, qualifiers and heaps (environments). Below are the definitions.

Definition 1. *Translation of terms from $\mathcal{W}$ to $\mathcal{L}$.*

$x_\mathcal{L} = x$
$(q < t_1, t_2 >)_\mathcal{L} = < t_{1\mathcal{L}}, t_{2\mathcal{L}} >_{q_\mathcal{L}}$
$(\text{split } t_1 \text{ as } x, y \text{ in } t_2)_\mathcal{L} = \text{split } t_{1\mathcal{L}} \text{ as } x, y \text{ in } t_{2\mathcal{L}}$
$(q\ \lambda x : q'\ P.\ t)_\mathcal{L} = \lambda_{q'_\mathcal{L}} x : P_\mathcal{L}.\ t_\mathcal{L}$
$(t_1\ t_2)_\mathcal{L} = t_{1\mathcal{L}}\ t_{2\mathcal{L}}$

Definition 2. *Translation of terms from $\mathcal{L}$ to $\mathcal{W}$.*

$x_\mathcal{W} = x$
$(< t_1, t_2 >_\pi)_\mathcal{W} = \pi_\mathcal{W} < t_{1\mathcal{W}}, t_{2\mathcal{W}} >$
$(\text{split } t_1 \text{ as } x, y \text{ in } t_2)_\mathcal{W} = \text{split } t_{1\mathcal{W}} \text{ as } x, y \text{ in } t_{2\mathcal{W}}$
$(\lambda_\pi x : P.\ t)_\mathcal{W} = \pi_\mathcal{W}\ \lambda x : \pi_\mathcal{W}\ P_\mathcal{W}.\ t_\mathcal{W}$
$(t_1\ t_2)_\mathcal{W} = t_{1\mathcal{W}}\ t_{2\mathcal{W}}$
$(\text{let}_\pi\ a_1 : A_1 = e_1, \ldots, a_n : A_n = e_n \text{ in } t)_\mathcal{W} = t_\mathcal{W}[a_1 \mapsto e_{1\mathcal{W}}] \ldots [a_n \mapsto e_n\mathcal{W}]$

Definition 3. *Translation of S:*

$(a = q\ t, S)_\mathcal{L} = a :_{q_\mathcal{L}} P = t_\mathcal{L}, S_\mathcal{L}$, where $\Gamma' \vdash t_\mathcal{L} : P$

Definition 4. *Translation of Γ:* $(a :_\pi A = t, \Gamma)_\mathcal{W} = a = \pi_\mathcal{W}\ t_\mathcal{W}, \Gamma_\mathcal{W}$

Definition 5. *Translation of types.*

Translation of types from $\mathcal{W}$ to $\mathcal{L}$:	***Translation of types from $\mathcal{L}$ to $\mathcal{W}$:***
$q\ (q'\ P_1 \to q''\ P_2)_{\mathcal{L}} = P_{1\mathcal{L}} \to_{q'_{\mathcal{L}}} P_{2\mathcal{L}}$	$(T_1 \to_\pi T_2)_{\mathcal{W}} = \pi_{\mathcal{W}}\ T_{1\mathcal{W}} \to \pi_{\mathcal{W}}\ T_{2\mathcal{W}}$
$q\ (T_1 * T_2) = T_{1\mathcal{L}} *_{q_{\mathcal{L}}} T_{2\mathcal{L}}$	$(T_1 *_\pi T_2)_{\mathcal{W}} = \pi_{\mathcal{W}}\ (T_{1\mathcal{W}} * T_{2\mathcal{W}})$

Definition 6. *Translation of qualifiers.*
$\text{lin}_{\mathcal{L}} = 1$
$\text{un}_{\mathcal{L}} = \omega$

Definition 7. *Translation of multiplicities.*
$1_{\mathcal{W}} = \text{lin}$
$\omega_{\mathcal{W}} = \text{un}$

Let $\to$ stand for reduction in the David Walker linear calculus [11] and $\Downarrow$ stand for reduction in Linear Haskell [3].

Example 1. Let us consider the following term in the linear calculus presented in [11]:

$$t \equiv (\text{lin}\ \lambda x : \text{lin}\ (\text{lin}\ \alpha \to \text{lin}\ \alpha).\ x)\ (\text{lin}\ \lambda y : \text{lin}\ \alpha.\ y)$$

and

$$\begin{aligned} &(\emptyset; (\text{lin}\ \lambda x : \text{lin}\ (\text{lin}\ \alpha \to \text{lin}\ \alpha).\ x)\ (\text{lin}\ \lambda y : \text{lin}\ \alpha.\ y)) \\ \to\ &(\{a_1 \mapsto \text{lin}\ \lambda x : \text{lin}\ (\text{lin}\ \alpha \to \text{lin}\ \alpha).\ x\}; a_1\ (\text{lin}\ \lambda y : \text{lin}\ \alpha.\ y)) \\ \to\ &(\emptyset; \text{lin}\ \lambda y : \text{lin}\ \alpha.\ y) \end{aligned}$$

Now we will translate t to Linear Haskell:

$$\begin{aligned} t_{\mathcal{L}} &= ((\text{lin}\ \lambda x : \text{lin}\ (\text{lin}\ \alpha \to \text{lin}\ \alpha).\ x)\ (\text{lin}\ \lambda y : \text{lin}\ \alpha.\ y))_{\mathcal{L}} \\ &= (\text{lin}\ \lambda x : \text{lin}\ (\text{lin}\ \alpha \to \text{lin}\ \alpha).\ x)_{\mathcal{L}}\ (\text{lin}\ \lambda y : \text{lin}\ \alpha.\ y)_{\mathcal{L}} \\ &= (\lambda_1 x : \alpha \to_1 \alpha.\ x)\ (\lambda_1 y : \alpha.\ y) \end{aligned}$$

Before evaluating, we need to transform the term using the Launchbury syntax [3].

$$\begin{aligned} (t_{\mathcal{L}})^* &= \text{let}_1\ a_2 : \alpha \to_1 \alpha = (\lambda_1 y : \alpha.\ y)^*\ \text{in}\ (\lambda_1 x : \alpha \to_1 \alpha.\ x)^*\ a_2 \\ &= \text{let}_1\ a_2 : \alpha \to_1 \alpha = \lambda_1 y : \alpha.\ y\ \text{in}\ (\lambda_1 x : \alpha \to_1 \alpha.\ x)\ a_2 \end{aligned}$$

and, let $\Gamma = \{a_2 :_1 \alpha \to_1 \alpha = \lambda_1 y : \alpha.\ y\}$,

$$\dfrac{\dfrac{\dfrac{}{(\Gamma; \lambda_1 x : \alpha \to_1 \alpha.\ x) \Downarrow (\Gamma; \lambda_1 x : \alpha \to_1 \alpha.\ x)} \quad \dfrac{\dfrac{}{(\emptyset; \lambda_1 y : \alpha.\ y) \Downarrow (\emptyset; \lambda_1 y : \alpha.\ y)}}{(\Gamma; a_2) \Downarrow (\emptyset; \lambda_1 y : \alpha.\ y)}}{(\Gamma; (\lambda_1 x : \alpha \to_1 \alpha.\ x)\ a_2) \Downarrow (\emptyset; \lambda_1 y : \alpha.\ y)}}{(\emptyset; \text{let}_1\ a_2 : \alpha \to_1 \alpha = \lambda_1 y : \alpha.\ y\ \text{in}\ (\lambda_1 x : \alpha \to_1 \alpha.\ x)\ a_2) \Downarrow (\emptyset; \lambda_1 y : \alpha.\ y)}$$

To compare both systems we need to define, for each language, a function called *deref*, which will go through the resulting term taking the environment into account and inlines the store/heap contents in the term until it has no free occurrences of variables. Now we are able to compare both terms using the function *deref*:

$$\begin{aligned} deref_{\emptyset}(\text{lin } \lambda y : \text{lin } \alpha.\ y) = \text{let } (y, \emptyset) = deref_{\emptyset}(y) \\ \text{in } (\text{lin } \lambda y : \text{lin } \alpha.\ y; \emptyset) \end{aligned}$$

$$\begin{aligned} deref_{\emptyset}(\lambda_1 y : \alpha.\ y) = \text{let } (y, \emptyset) = deref_{\emptyset}(y) \\ \text{in } (\lambda_1 y : \alpha.\ y; \emptyset) \end{aligned}$$

where $(\text{lin } \lambda y : \text{lin } \alpha.\ y)_{\mathcal{L}} = \lambda_1 y : \alpha.\ y$ and $(\lambda_1 y : \alpha.\ y)_{\mathcal{W}} = \text{lin } \lambda y : \text{lin } \alpha.\ y$.

Finally, we proceed to the main results of our paper. In Theorem 1, we prove by induction on the length k of the reduction sequence, that if we make a computation in the David Walker calculus then, if we translate the initial pair to Linear Haskell and reduce it, the $deref_{\Gamma}$ yields the same result in both computations. By proving this theorem, we are able to show that we can simulate David Walker calculus in Linear Haskell.

In Theorem 2, we will prove the opposite simulation, also by induction on the length of the derivation tree, where we are able to show that if we make a computation in Linear Haskell then, if we translate the initial pair to the David Walker calculus and reduce it, the $deref_S$ yields the same result in both computations.

Theorem 1.
If $(S; M_{\mathcal{W}}) \rightarrow_k (S'; N_{\mathcal{W}})$ *then*

$$(S^*_{\mathcal{L}}; (M_{\mathcal{W}})^*_{\mathcal{L}}) \Downarrow (\Gamma; N_{\mathcal{L}}) \textit{ and } deref_{\Gamma}(N_{\mathcal{L}}) = deref_{S'^*_{\mathcal{L}}}((N_{\mathcal{W}})^*_{\mathcal{L}})$$

Theorem 2.
If $(\Gamma; M^*_{\mathcal{L}}) \Downarrow (\Gamma'; N_{\mathcal{L}})$ *then*

$$(\Gamma_{\mathcal{W}}; (M_{\mathcal{L}})_{\mathcal{W}}) \twoheadrightarrow (S; N_{\mathcal{W}}) \textit{ and } deref_{S}(N_{\mathcal{W}}) = deref_{\Gamma'_{\mathcal{W}}}((N_{\mathcal{L}})_{\mathcal{W}})$$

3 Conclusions

In this paper, we defined two compilations that relate two different linear calculi, the David Walker calculus [11], a linear type system which uses a small-step operational semantics, and Linear Haskell [3], an extension to Haskell, that uses a big-step semantics. We were able to show that there exists an equivalence between both systems, despite several differences in syntax and the kind of operational semantics (small-step and big-step) used, confirming the robustness of linear type theory because the choice of semantics does not fundamentally alter the behaviour of programs.

Acknowledgements This work was partially financially supported by the Artificial Intelligence and Computer Science Laboratory, LIACC, funded by national funds through the FCT/MCTES (PIDDAC) - UIDB/00027/2020.

References

1. Alves, S., Fernández, M., Florido, M., Mackie, I.: Linearity and recursion in a typed lambda-calculus. In: Proceedings of the 13th International ACM Conference on Principles and Practice of Declarative Programming (PPDP) (2011)
2. Alves, S., Fernández, M., Florido, M., Mackie, I.: Linearity: a roadmap. Journal of Logic and Computation **24**(3), 513–529 (2014)
3. Bernardy, J.P., Boespflug, M., Newton, R.R., Peyton Jones, S., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. Proc. of the 44th ACM Symposium on Principles of Programming Languages **2**(POPL) (2017)
4. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Mathematical Structures in Computer Science **26**(3), 367–423 (2016)
5. Coppola, P., Lago, U.D., Rocca, S.R.D.: Elementary affine logic and the call-by-value lambda calculus. In: Typed Lambda Calculi and Applications, (TLCA). Lecture Notes in Computer Science, vol. 3461, pp. 131–145. Springer (2005)
6. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1–101 (1987)
7. Hughes, J., Orchard, D.: Program synthesis from graded types. In: Proceedings of the 33rd European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 14576, pp. 83–112. Springer (2024)
8. Launchbury, J.: A natural semantics for lazy evaluation. In: Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL) (1993)
9. Mackie, I.: Lilac: A functional programming language based on linear logic. Journal of Functional Programming **4**(4), 395–433 (1994)
10. Maraist, J., Odersky, M., Turner, D.N., Wadler, P.: Call-by-name, call-by-value, call-by-need and the linear lambda calculus. Theoretical Computer Science **228**(1-2), 175–210 (1999)
11. Walker, D.: Substructural type systems. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages. pp. 3–44. The MIT Press (2005)