Linear Type from Linear Logic

Jen-Shin Lin

November 10, 2016

Writing programs with a language which supports linear operations has some fascinating benefits. However, providing good properties means the presence of powerful syntax. And powerful syntax equals to complicate syntax that unavoidably leads to the increacment of programming difficulty for human. Therefore one naturally will want to apply type system to a linear language as an expilict guarding mechanism. Since the linear logic provides an operational semantics [2] for using and managing resources, it's natural to apply the well-known CurryHoward correspondence and construct a linear type system from linear logic.

Linear logic was introduced via sequent calculus [1, 2] in which deductive rules are defined in a left/right flavor. It is fine for a logic system, however, when one attempt to build a type system from those deductive rules, things go south immediately. The most important reason is that, as examples below, in a L/R logic rule the connective is always introduced into the conclusion and never premises.

$$\frac{\Gamma \vdash A \qquad B, \Delta \vdash C}{\Gamma, A \multimap B, \Delta \vdash C}\ (\multimap\text{-L})$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ (\multimap\text{-R})$$

If we force to produce typing rules from L/R rules, things will become somewhat an awkward situiation. First of all, the term assignment for this kind of typing rules will be more complicated. This will immediately make those tpying rules become unsuitable for applying in safety proving.

One of the alternative choices would be translating L/R rules into introduction/elimination rules. Namely rewrite linear logic's deductive rules as in natural deduction system. The following rules are the I/E rules corresponding to those in above example. As it presets itself, with I/E rules one can easily construct and destruct terms from or back to its premises.

$$\frac{\Gamma \vdash A \multimap B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}\ (\multimap\text{-E})$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ (\multimap\text{-I})$$

In the design of our linear language, as we mentioned in the last report, the program will be allowed to be evaluated within linear mode (as default) or non-linear mode. This is done by providing a syntax-level function, `bang`, that can be used to annotate a variable or expression for being treated as a non-linear resource. To be able to evaluate program in both modes, the way to go is to seperate environment context into to parts: linear context and non-linear context. Nevertheless they cannot be entirely seperated because of the fact that they are both a part of a big abstract context in which we allow no name duplicate. To solve this problem we introduce the dual distinguished-domain, which is defined as `DualDomDist` in file: *Ctx.agda*[1]. This dual domain design can make proving easier and clearer, however, it also generate lots of requirements of proofs of related properties, which can be found in file: *Auxiliaries.agda*[1].

Besides above efforts, the most significant result here is that linear typing rules together with its corresponding language syntax, which can be found well-consistent with the work in [3], were well-defined in agda. We also attempted to use our typing rules to prove some type properties, such as structural rule, are still holded. As an example, the attempt of proving weaken rule can be found in file: *Ctx.agda*[1]. This works, however, is unfortunately incomplete because it took lots of time to deal with

[1] links to source codes are listed in the last paragraph of this article

something relative unimportant like avoiding name conflicting or trival theorem of dual domain.

In the end, it turned out that we do not actually need to rigidly prove our linear type system since our goal is take the advantage from linear type to build a reversable language and a customized type system for it. The thing we need is to show that we have an intact type system from linear logic and a well-defined language syntex containing syntax-level functions for linear operations. Therefore the result so far we have is enough for us to understand and be familiar with the notion of linear world. And, from those results, we can now move forwards to refine our langauge to provide extra power for not only linear programming but also reversiable programming.

The following are the complete links for all agda source codes.

basic syntax [https://dl.dropboxusercontent.com/u/11966021/src/LTLang/Expr.agda]

type systems [https://dl.dropboxusercontent.com/u/11966021/src/LTLang/Types.agda]

free variable namipulation [https://dl.dropboxusercontent.com/u/11966021/src/LTLang/FVar.agda]

term substitution [https://dl.dropboxusercontent.com/u/11966021/src/LTLang/Substitution.agda]

environment context [https://dl.dropboxusercontent.com/u/11966021/src/LTLang/Ctx.agda]

typing rules [https://dl.dropboxusercontent.com/u/11966021/src/LTLang/Typing.agda]

structural type rules [https://dl.dropboxusercontent.com/u/11966021/src/LTLang/Structural.agda]

aux proofs [https://dl.dropboxusercontent.com/u/11966021/src/LTLang/Auxiliaries.agda]

References

[1] Girard, J.-Y. Linear logic. *Theor. Comput. Sci. 50*, 1 (Jan. 1987), 1–102.

[2] Girard, J.-Y. Linear logic: Its syntax and semantics, 1995.

[3] Wadler, P. A taste of linear logic. In *Mathematical Foundations of Computer Science* (Gdańsk, Poland, 1993), A. M. Borzyszkowski and S. Sokolowski, Eds., Springer-Verlag LNCS 781, pp. 185–210.