Ben Greenman October 13, 2015
CS 7480 Summary Substructural Type Systems

A substructural type system contains types for which some (or all) of the standard context-manipulation lemmas do not hold. This week's papers focused on two rules—COMPRESSION and WEAKENING—and ultimately studied the 4 possibilities that arise when these rules may be toggled across types in the language. Additionally, the chapter by Walker briefly considered types where neither COMPRESSION, WEAKENING, or EXCHANGE hold [3].

The practical motivation for this line of work is tracking resource use and allocation. Examples of programming features enabled by these type systems are strongly-updatable references, race-free threads, and reference counting. At first glance it is surprising that a type system can effectively reason about these diverse features, but the proof is in the papers. However these type systems required very careful design and would be difficult to program in. There are too many annotations!
(Since I didn't finish the papers, I'll just give "greatest hits" rather than strengths.)

Strengths

- *L3* presents the semantics of strong updates in terms of linear types [2]. Well-typed programs are guaranteed to terminate and have a well-typed semantic interpretation.
- Someone deserves credit for introducing the swap operator. At least, Walker convinced me that swap is very useful. His entire chapter was very motivating—the wide range of examples was impressive.
- *Substructural State* gives many examples (thankfully!) describing the intricacies of mixing substructural types, strong references, and polymorphism [1].
- These papers have encouraged me to read Jesse Tov's thesis on the next long weekend (Thanksgiving?).

Weaknesses

- Besides Walker's *ordered* types, none of the papers talk about toggling the EXCHANGE rule. For example, affine types without EXCHANGE would be necessary for modeling big-bang programs, which run in a fixed event loop, but skip rendering for some frames for performance reasons. I am curious to see what the 8-point lattice would look like, but I can only imagine how compilicated it would be after reading *Substructural State*.
- None of the papers consider general principles for adding new kinds of substructural types. The lattice is fixed at the start and never extended.

It is clearly a difficult research problem to work out the correct interactions between substructural types, so even if the design process remains manual a consistency-checking tool that explores the possible combinations in a given lattice for invariant violations would be a great help.

- Rust's success controlling data races is very compelling, but while reading these papers I did not get a strong sense of why a (complicated) type system like this would be an improvement over dedicated syntax for resource control like Python's `with` statement. The syntax guarantees that the resource is closed or lock released, and this guarantee is visually obvious from the code. In contrast these papers' heavily-annotated typed languages seem only easy for a compiler to reason about.

References

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *ICFP*, 2005.

[2] Greg Morrisett, Amal Ahmed, and Matthew Fluet. A step-indexed model of substructural state. In *TCLA*, 2005.

[3] David Walker. Substructural type systems. In *ATAPL*, 2005.