L³: A Linear Language with Locations

Greg Morrisett¹, Amal Ahmed¹, and Matthew Fluet²

¹ Harvard University
{greg, amal}@eecs.harvard.edu
² Cornell University
fluet@cs.cornell.edu

Abstract. We explore foundational typing support for strong updates — updating a memory cell to hold values of unrelated types at different points in time. We present a simple, but expressive type system based upon standard linear logic, one that also enjoys a simple semantic interpretation for types that is closely related to models for spatial logics. The typing interpretation is strong enough that, in spite of the fact that our core calculus supports shared, mutable references and cyclic graphs, every well-typed program terminates.

We then consider extensions needed to make our calculus expressive enough to serve as a model for languages with ML-style references, where the capability to access a reference cell is unrestricted, but strong updates are disallowed. Our extensions include a thaw primitive for temporarily re-gaining the capability to perform strong updates on unrestricted references.

1 Introduction

The goal of this work is to explore foundational typing support for strong updates. In type systems for imperative languages, a strong update corresponds to changing the type of a mutable object whenever the contents of the object is changed. As an example, consider the following code fragment written with SML syntax:

1. let val r = ref () in
2.   r := true;
3.   if (!r) then r := 42 else r := 15;
4.   !r + 12
5. end

At line 1, we create a ref cell r whose contents are initialized with unit. At line 2, we change the contents so that r holds a bool. Then at line 3, we change the contents of r again, this time to int. In spite of the fact that at different program points r holds values of different, incompatible types, there is nothing in the program that will cause a run-time type error. This is because subsequent reads of the reference are type-compatible with the immediately preceding writes.

¹ We assume that values are represented uniformly so that, for instance, unit, booleans, and integers all take up one word of storage.
Unfortunately, most imperative languages, including SML and Java, do not support strong updates. For instance, SML rejects the above program because it requires that reference cells hold values of exactly one type. The reason for this is that tracking the current type of a reference cell at each program point is hindered by the potential for aliasing. Consider, the following function:

1. fun f (r1: int ref, r2: int ref): int =
2.   (r1 := true;
3.     !r2 + 42)

In order to avoid a typing error, this function can only be called in contexts where \( r1 \) and \( r2 \) are different ref cells. The reason is that if we passed the same cell for each formal argument, then the update on line 2 should change not only the type of \( r1 \) but also the type of \( r2 \), causing a type error to occur at line 3.

Thus, any type system that supports strong updates needs some control over aliasing. In addition, it is clear that the hidden side-effects of a function, such as the change in type to \( f \)'s first argument in the example above, must be reflected in the interface of the function to achieve modular type-checking. In short, strong updates seem to need a lot of technical machinery to ensure soundness and reasonable accuracy.

Lately, there have been a number of languages, type systems, and analyses that have supported some form of strong updates. The Vault language [1,2] was designed for coding low-level systems code, such as device drivers. The ability to track strong updates was crucial for ensuring that driver code respected certain protocols. Typed Assembly Language [3,4] used strong updates to track the types of registers and stack slots. More recently, Foster and Aiken have presented a flow-sensitive qualifier system for C, called CQUAL [5], which uses strong updates to track security-relevant properties in legacy C code.

Vault, later versions of TAL, and CQUAL all based their support for strong updates and alias control on the Alias Types formalism of Smith, Walker, and Morrisett [6]. Though Alias Types were proven sound in a syntactic sense, we lacked an understanding of their semantics. Furthermore, Vault, TAL, and CQUAL added a number of new extensions that were not handled by Alias Types. For instance, the restrict operator of CQUAL is unusual in that it allows a computation to temporarily gain exclusive ownership of a reference cell and perform strong updates, in spite of the fact that there may be unknown aliases to the object.

In this paper, we re-examine strong updates from a more foundational standpoint. In particular, we give an alternative formulation of Alias Types in the form of a core calculus based on standard linear logic, which yields an extremely clean semantic interpretation of the types that is directly related to the semantic model of the logic of Bunched Implications (BI) [7]. We show that our core calculus is sound and that every well-typed program terminates, in spite of the fact that the type system supports first-class, shared, mutable references with strong updates. We then show how the calculus can be extended to support a combination of ML-style references with uncontrolled aliasing and a restrict-like primitive for temporarily gaining exclusive ownership over such references.