Discussion: Benchmarking Binding

Stephanie Weirich
sweirich@cis.upenn.edu

1 How should we represent binding?

Implementations of type systems, logics and programming languages must often answer the question of how to represent and work with binding structures in terms. While there may not be a one-size-fits-all solution, I would like to understand the trade-offs that various approaches to binding make in terms of execution speed and simplicity of implementation.

Binding structures are epitomized in the lambda calculus by the abstraction term, $\lambda x.e$, that binds the variable x within the body e. All occurrences of x in this body are references to this bound variable. Other forms of binders include quantifiers (e.g. $\forall$,$\exists$,Π,Σ,ν), recursive definitions, pattern matching, etc.

To that end, I have been constructing a benchmark platform using the Haskell programming language and have been gathering multiple implementations of binding for comparison and experimentation.[1] I propose to use this repository as a focus for a discussion about the following questions related to aggregating and comparing implementations of binding.

1.1 What should a benchmark suite contain?

We don't just representating lambda terms, we also want to work with that representation efficiently. Therefore, the goal of a benchmark suite should be to evaluate different implmentations of the same operations. But which ones?

The operations that I think are important include:

- Conversion to / from a "raw string" representation
- Comparing for alpha-equivalence
- Capture-avoiding substitution
- Free variable set calculation

Designing a uniform and informative way to evaluate implementations of these operations is challenging. One issue is that the operations have different types with different binding representations. For example, if an implementation of capture-avoiding substitution requires freshening, it may require additional arguments to be sure that the new names chosen are fresh.

But, more fundamentally, are these the right operations to begin with? Do implementers use capture-avoiding substitution outside of teaching students about programming language theory? Should we add others to this list, such as normalization, alpha-invariant ordering, or hashing? Certain representations may also require additional functions, such as shifting (changing the scope of a term), freshening (renaming free variables), or instantiation (replacing a binding variable with a term) when working with terms, how do those interact with our benchmarks? Should we only measure larger operations, such as type checking or source-to-source translations? How can we do that while still comparing a large number of implementations, being sure that they all implementing the same operation?

1.2 What optimizations can we evaluate?

There are ways to optimize an operation like capture-avoiding substitution that are independent of the binding representation chosen. For example, if we know that a variable does not appear in a term, we can terminate the substitution operation early. But tracking the free variables of a term comes at a cost. Where is the trade off?

Alternatively, if we can suspend substitution operations inside terms, we can fuse multiple traversals together, which makes a difference in, say normalization, which requires repeated uses of substitution. Furthermore, if substitution is interleaved with another function over lambda terms, say $[\![a]\!]$, then instead of computing $[\![a\{b/x\}]\!]$ it might be more efficient to compute $[\![a]\!]\{[\![b]\!]/x\}$ instead, when substitution commutes with the operation.

Are there other optimizations that apply to capture-avoiding substitution? To the other operations? How can we evaluate their effectiveness?

1.3 What about library support?

Several libraries exist to support Haskell programmers in implementing binding including `unbound-generics`[2], `bound`[3], and others. These libraries simplify working with binding.

The `unbound-generics` library uses a locally nameless representation. The `bound` library uses a variant of de Bruijn indices based on nested datatypes. Other libraries are inspired by nominal logic. What representations can we use for such libraries? How can we make them both easy to use and efficient? What is the abstraction cost for introducing such a library? Can we design libraries that minimize this cost?

1.4 What is used in practice?

Above, I've provided my answers to these questions to make it clear the kinds of discussion that I'm looking for. However, I'm keen to hear the experiences of others, hence the discussion format.

[1] https://github.com/sweirich/lambda-n-ways. This repository is based on a draft paper by Lennart Augustsson, but extends his four original implementations of the untyped lambda calculus with many additional variations.

[2] https://hackage.haskell.org/package/unbound-generics

[3] https://hackage.haskell.org/package/bound