

Comp 617 Proposal

Ed Westbrook

This proposal deals with four potential research papers: staged monad transformers; native code performance numbers for the examples in the document “A Gentle Introduction to Multi-Stage Programming, Part II”, or GIMSP-II; constructive real numbers for modeling hybrid semantics; and encoding variable bindings as data. These are discussed below.

1 Staged Monad Transformers

In previous work [3], it was shown how modular interpreters can be written using monad transformers. A *monad* is a representation of some notion of computation. For example, continuation passing forms a monad, as does mutable state. A *monad transformer* is a construction that builds a more complex monad out of another monad, adding on some notion of computation to the existing monad. For example, the above-mentioned paper demonstrates how to build a monad transformer that adds continuation passing to an existing monad.

I propose here a paper that stages the aforementioned previous work. Specifically, the modular interpreter written in that work will be written here in Meta-OCaml Concoction, and will then be staged. The goal is to study the interplay between monads and staging, specifically looking to exploit potential speedups that could be brought with staging.

2 Native Code Performance Numbers for GIMSP-II

Domain-specific languages, or DSLs, have seen increasing use in the software engineering community. DSLs can be implemented in one of two ways, with either an interpreter, which symbolically interprets a DSL, or a compiler, which translates programs written in a DSL to some existing language or to native code. This choice represents a trade-off: interpreters are easier to write, but compilers are more efficient. In the GIMSP-II paper [7], a “middle ground” is proposed, called staged interpreters. Staged interpreters are interpreters, so are easier to write than compilers, but staging allows the interpreter itself to generate and compile special-purpose code for each program it interprets, leading to greater efficiency. Performance numbers collected in the GIMSP-II paper demonstrate that staging does indeed offer greater efficiency. The data collected there, however, considered only the case where the interpreter and the code it creates by staging are compiled to byte-code. I propose to run similar experiments on the same programs with compilation to native code. This could potentially demonstrate a greater speedup, but could also be used to directly compare staged interpreters with compilers, hopefully validating the claim that staged interpreters can effectively be used in place of compilers for DSLs.

3 Constructive Real Numbers for Hybrid Semantics

As more technology is dependent upon embedded controllers, there is a growing call to model and predict the behavior of these embedded controllers in real environments. Recent work [1] has suggested one approach to this problem, which involves the combination of a discrete model, to describe the embedded software, and a continuous model, to describe the physical environment of the embedded controller. This approach to combined discrete and continuous models is called *hybrid semantics*. One potential difficulty with this approach is that the continuous model cannot be directly modeled or represented inside a computer itself. This is because the continuous model uses real numbers, of which there are uncountably many, and computers

can only represent countable sets. To solve this problem, I propose to study *constructive* formalizations of real numbers. These are countable formalizations using recursive functions, and so can be represented by computer. One starting point will be the book by Sanin on the subject [6].

4 Encoding Variable Bindings as Data

Consider the task of writing software, such as a compiler, interpreter, or static analysis tool, that operates on programs themselves as data. In order to operate on programs, such software must *encode* programs as some datatype `program`. For example, if the given software is to operate on Java programs, there must be some element of the `program` datatype which corresponds to the identity function

```
int id (int x) { return x; }
```

on the type `int`.

The `id` function given above includes an instance of what is called a *variable binding*. Specifically, `id` is said to *bind* the variable `x` inside its body. Variable binding has the following three properties:

1. **Freshness:** `x` is a new thing, distinct from any other element of the programming language;
2. **α -equivalence:** the name `x` itself is irrelevant, meaning the program

```
int id (int y) { return y; }
```

is identical to the definition given above; and

3. **Scoping:** Variables such as `x` cannot appear outside a binding for them, so, for example, the function

```
int foo (int x) { return y; }
```

is ill-formed, unless it is nested inside a function with argument `y`.

Encoding variable binding in such a way as to satisfy these three properties is known to be a difficult problem. For example, using the character string ‘`x`’ to encode the variable `x` does not satisfy α -equivalence. This is because such a character string is not equal to the string ‘`y`’.

My proposal related to this topic is two-fold. First, I will present on known techniques for encoding variable binding, including Higher-Order Abstract Syntax [5], Nominal Logic [2], deBruijn Indices, and the Locally Nameless approach [4]. Second, I will implement and present a new approach to encoding variable binding, based on ideas from my doctoral dissertation [8]. The implementation will take the form of a language, called Nominal Meta-OCaml, which adds a construct called a *ν -abstraction* to the Meta-OCaml language. ν -abstractions can then be used to encode variable binding, as they have the above three properties by definition. Variable binding is thus had “for free” in Nominal Meta-OCaml.

References

- [1] O. Bouissou and M. Martel. A hybrid denotational semantics for hybrid systems. In *17th European Symposium on Programming (ESOP '08)*, 2008.
- [2] M. Gabbay and A. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
- [3] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd Symposium on Principles of Programming Languages*, 1995.

- [4] C. McBride and J. McKinna. I am not a number: I am a free variable. In *Proceedings of the Haskell Workshop*, 2004.
- [5] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation*, 1988.
- [6] N. A. Sanin. *Constructive Real Numbers and Constructive Function Spaces*. AMS, 1968. Available online at http://www.ams.org/online_bks/mmono21/.
- [7] W. Taha. A gentle introduction to multi-stage programming, part ii. Available from author by email.
- [8] E. Westbrook. *Higher-Order Encodings with Constructors*. PhD thesis, Washington University in Saint Louis, 2008.