CSCI-B629 Writeup

Spenser Bauman

May 6, 2015

1 Truffle Interpreter

This portion of the project consisted of making an interpreter for a Scheme like language using the Truffle DSL. Truffle consists of several components which handle the details of tree rewriting in an AST interpreter. This document will attempt to outline the major components and how they are used to implement this interpreter. The basic implementation strategy was based off of a Truffle tutorial[1].

My initial strategy was to write an AST interpreter and then transition it over to using Truffle, hoping that the modifications could be made gradually – similar to an RPython interpreter. The transitioning took more work that was expected, as Truffle requires all AST nodes to be capable of performing tree rewriting.

1.1 Truffle Annotations

The Truffle DSL consists largely of a set of Java annotations. These annotations then generate many of the class members for managing tree rewriting and specialized nodes from a class specification. The user specifies which components of an AST node correspond to child nodes, and largely allows the user to ignore the low level details of tree rewriting.

The biggest challenge with using Truffle is learning to phrase optimizations as tree rewriting. For instance, this interpreter uses tree rewriting to speed up variable lookup by rewriting variable nodes which traverse the environment to a version with a cached environment frame.

The annotations also generate the class definitions for the specialized versions of a node. Manually implementing each class would require much more work, resulting in several times more class definitions in the implementation.

1.2 Environment Management

To facilitate type specialization of environment accesses (and communicate this information to Graal), Truffle defines several `Frame` types. During parsing,

[1] `https://cesquivias.github.io/blog/2014/10/13/writing-a-language-in-truffle-part-1-a-simple-slow-interpreter/`

the user must generate a `FrameDescriptor` which specifies the shape of the environment frames. Lexical scoping is obtained by linking frames into a list like structure and storing them on closures.

This approach is somewhat more restrictive than RPython, requiring up front processing and exclusive use of the Truffle API. That said, Truffle frames contain slots for unboxed values, so the end result is unboxing equivalent to how Pycket unboxes values in environment frames – without manual intervention.

1.3 Procedure Calls

The most difficult functionality to implement was procedure calls. Truffle has a notion of a "call target", which is created when building a closure, wrapping up the code for the function. Similarly, application nodes include a call node which is responsible for invoking the call target after.

Truffle includes several types of call nodes, depending on how procedure calls work in the language. Some call nodes may be better optimized by Truffle and Graal, at the cost of more implementation work. The interpreter currently uses the `IndirectCallNode`, as it is the easiest to use.

One difficulty with implementing Scheme on top of Java as an AST interpreter is the implementation of tail calls. Application nodes in tail position will allocate a Java stack frame, so a reliable Scheme implementation needs a work around. When an application in tail position occurs, a `TailCallException` is thrown, which wraps the call target and arguments. Function applications not in tail position consist of loops which catch tail call exceptions and execute the wrapped function.

This is an inelegant way to implement tail calls, but is necessitated by the structure of Truffle interpreters. This is definitely a case where less constrained nature of RPython interpreters wins out.

1.4 Conclusion

While the performance of the interpreter has gotten much better by using Truffle, it is still over an order of magnitude slower than Pycket on simple benchmarks. Truffle adds in the logic needed to use the Graal JIT, which provides huge performance improvements of the standard JVM, though I have not yet managed to perform serious benchmarking, and I have only performed a few optimizations.

At this point, I believe it is possible to produce an interpreter competitive with Pycket using Truffle, though more work may be required to achieve similar functionality – I have no idea how to go about implementing features like `call/cc` though escape continuations may not be too difficult. (R)Python's metaprogramming facilities are more usable than Java's, which greatly simplifies a lot of the implementation work of Pycket. On the other hand, the fact that Truffle (including the automatic code generation) works with Eclipse is a big plus in favor of the Truffle approach.

2 Partial Evaluator

This part of the project made use of the `pgg` partial evaluator by Mike Sperber et al. `pgg` is a partial evaluator for the Scheme 48 system. The general goal of this project was to explore how well the partial evaluator could remove the interpretation overhead of a simple definitional interpreter when specializing on a program. The results were mixed, as the partial evaluator seems to be rather sensitive to the code structure and some of the hints provided by the partial evaluator.

The partial evaluator did not perform as well on the interpreters I devised as it did on their example interpreters. The example interpreter that came with `pgg` typically had all of the environment management elided in the resulting code, which I have not been able to manage.

That said, `pgg` managed to eliminate most of the overhead from the interpreter on simple programs. All of the overhead of dispatching on the AST structure was elided from the final result.

My initial attempts were straightforward definitional interpreters using a list of pairs for the environment representation. Depending on the program, often resulted in environment management and AST dispatch in the residual code – typically anything with a `lambda` would produce this behaviour. Two things improved the initial versions of the code: primitive annotations and capturing variables rather than passing them around. The `pgg` system has some annotations that tell it how to treat primitive operations like `apply` and `cons`. Annotating `cons` as pure, for instance, improved the residual code for environment management. The proper annotations for `apply` allow `pgg` to convert uses of the `apply` function to normal function calls when the arity is known statically.

The `pgg` documentation suggested that using custom data types, rather than `cons` cells, might work better for an environment representation, but that had little effect on the residual code. One annotation that improved the resulting code (when it did not result in non-termination of `pgg`) was to define the interpretation function without memorization, which should be safe when the functions termination depends only on static data. This annotation typically resulted in the partial evaluator exploring error paths in environment lookup, though those branches are dead code.

In an attempt to remove the overhead of environment traversal, I produced a manually staged version of the interpreter. To do so, the environment is split into static and dynamic portions The interpretation function then takes the input expression and static environment and produces a function from the dynamic environment to the result. Staging the interpreter allowed the partial evaluator to remove symbol comparisons from the residual code, though not the traversals of the dynamic environment.

Overall, this seems like an unreliable method for implementing a high performance compiler from an interpreter. It is difficult to assess ahead of time how the partial evaluator will behave and little diagnostic information to address the problem It is difficult to assess ahead of time how the partial evaluator will behave and little diagnostic information to address the problem.