Scheme2Beam

Harlan Kringen and Zach Sisco

CS 263 – Spring 2020

1 Introduction

The λ-calculus is a foundational model of computation which has inspired functional programming languages like Lisp and Scheme. It is well understood and we feel confident in using it in our research. However, there is an analogous model of computation situated in the parallel and concurrent world. We have recently become interested in this mirror world, which has led us to the π-calculus, and subsequently to Erlang, a language very much inspired by the π-calculus.

A language's runtime environment is a critical component in understanding this analogous computation model since much of what constitutes parallel and concurrent behavior takes place in real time, during program execution. To gain a better understanding of the runtime environment we decided to write a translation from a λ-calculus-like language to a π-calculus-like language, which directly addresses the concurrent world. Concretely, this translation is a source compiler taking Scheme to Erlang, which optionally autogenerates a parallelized version of the source code.

2 Contributions

In this project, we wrote a source compiler translating a subset of the Scheme language into a language that runs on the Erlang VM, or BEAM. We also developed an IR pass for our compiler which attempts to parallelize code based on a translation given by Robin Milner [1], mapping the λ-calculus to the π-calculus. Our compiler targets not Erlang directly, but an intermediate language in the Erlang toolchain known as Core Erlang. This language proved to be a simpler translation target than Erlang itself, and there is a tradition of building languages on top of Core Erlang, such as Elixir. Our source compiler is written in OCaml and relies on only one external library, totaling a very manageable 700 lines of code. We wrapped our compiler in a simple command line tool called S2B which can dumps to file the Core Erlang, which can then be readily compiled to BEAM bytecode. After we finished writing the source compiler, we saw our optional parallelization pass as "completing the circuit" shown in Figure 1. We were familiar with Robin Milner's theoretical translation, and we knew we had concrete implementations of the constituent parts, so we simply wanted to see if we could provide the concrete implementation of the translation as a whole.

3 Background

Erlang is a programming language made for designing fault-tolerant systems. It comes with built-in primitives for reasoning about parallelism and concurrency. To understand how to write a source compiler that targets the Erlang VM, we give a brief description of its compiler toolchain.

At a high level, the flow of the Erlang compiler is from source code to BEAM bytecode. Erlang source code is first transformed into an abstract syntax tree called the Erlang *abstract format*. The Erlang abstract format is then converted to *Core Erlang*, an intermediate representation (IR) used by the compiler for optimizations and program analyses. (See Section 4.2 for more discussion on Core Erlang.) After the Erlang compiler performs optimizations and

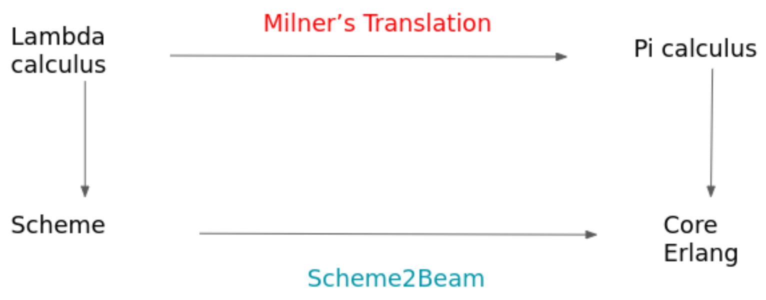

Figure 1: Diagram of translations from λ-calculus to π-calculus, from Scheme to Erlang, and where our project fits in.

other program transformations over Core Erlang, it finally compiles down to BEAM bytecode.

There is no official specification of Core Erlang. The only reliable reference is the latest source code in the Erlang compiler. We relied on this closely for specifying Core Erlang in OCaml. There are about 30 syntactic constructs that make up Core Erlang. Since we are writing a source compiler for Scheme, we actually did not need to implement every single construct, only the ones necessary for expressing Scheme. In the end, we ended up implementing 20 of the 30 language constructs.

4 Implementation

4.1 Scheme as a Source Language

As a descendent of the Lisp family of languages, Scheme employs a syntax style known as *s-expressions* which constitutes a simple, formal grammar consisting of atomic tokens and lists of atomic tokens (recursively). This format makes parsing the raw string format of Scheme relatively straightforward and also fits well within the strengths of the OCaml language.

While the tokenizing phase gives us a simple abstract syntax tree, we must imbue this format with a semantics closer to our target language. Before we can generate concrete Core Erlang, we need our own version of abstract Core Erlang. To this end, we created our own representation of abstract Core Erlang using OCaml algebraic data types. Because of this, the code generation became a simple recursive tree traversal.

Parsing Scheme into our internal representation required towing the line between bare parsing and actual evaluation. The Erlang language exposes some facets of the AST to the user in the concrete syntax, such as requiring the number of function parameters to be specified directly in the definitions. This required the parsing phase to maintain a type of symbol table in which we could store more detailed information about the code during parsing and code generation. We anticipate that there are unique gains in breaking this parsing phase out into possible further passes that perform symbolic execution or some other optimization.

This parsing traversal is largely responsible for generating concrete Core Erlang and it constitutes the main utility of our compiler. We do however, as we explain in subsequent sections, perform another IR pass, modifying the Core Erlang AST in an attempt to parallelize the original source program.

4.2 Core Erlang as a Target Language

Most high-level language constructs in Erlang source code compile down to sequences of function definitions and applications, case statements, guards, and let bindings in Core Erlang. Being a smaller language (but still higher level than BEAM bytecode), Core Erlang is an ideal IR for our source compiler. Other language designers have noted this compatibil-

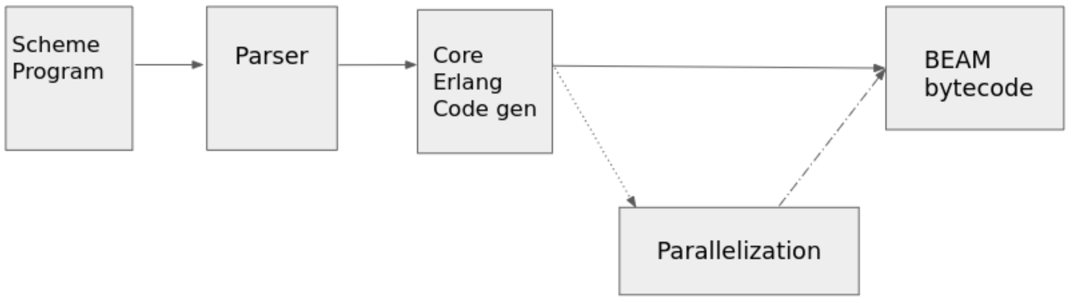

Figure 2: Source compiler process flow.

ity as well, such as Elixir, which compiles its source language to Core Erlang.

As we have referenced, Core Erlang has two syntactic representations: *concrete* and *abstract*. For actually performing program transformations over Core Erlang, the compiler uses the abstract representation, which specifies Core Erlang as a data type over all syntactic keywords and constructs in the language. The compiler uses the concrete representation to emit human readable Core Erlang.

The backend of our source compiler must handle both representations of Core Erlang, the abstract syntax to map parsed Scheme to constructs in Core Erlang, and the concrete syntax to emit valid Core Erlang. As an example, Listings 1 and 2 show a factorial function written in Scheme and its corresponding concrete Core Erlang. The overall pipeline of our compiler can be see in Figure 2.

4.2.1 Generating Concrete Core Erlang

To generate Core Erlang, we relied mostly on header comments in the Core Erlang source code, and our own hand-written Erlang examples that we compiled down to Core Erlang. Although not perfect, this informed us how to format concrete Core Erlang in order to be accepted by the Erlang compiler.

Given that there is no formal specification of Core Erlang, we were limited in how we verified the correctness of generated Core Erlang. The best assurances we have is that our unit tests are accepted by the Erlang compiler, successfully compile down to BEAM bytecode, and can be run on the BEAM.

```
(define (factorial n)
  (if (< n 1)
      1
      (* n (factorial (- n 1)))))
```

Listing 1: Factorial function written in Scheme.

```
module 'factorial' ['factorial'/1]
  attributes []
'factorial'/1 = fun (_n) ->
    case <> of
      <> when call 'erlang':'<'(_n,1)
          ->
        1
      <> when 'true' ->
        call 'erlang':'*'(_n,apply
            'factorial'/1(call
            'erlang':'-'(_n,1)))
    end
end
```

Listing 2: Scheme factorial function translated to Core Erlang.

4.3 Functions as Processes

At this point, our compiler can take Scheme code and run it on the BEAM; however, we have not yet addressed Erlang's primitives for concurrency. Vanilla Scheme cannot express these directly so we need to take some cues from the theory of concurrent systems. As we mentioned in the introduction, the π-calculus is an analogue to the λ-calculus that expresses computations in terms of processes communicating over named channels, essentially swapping the abstraction of a "function" for the abstraction of a "process." Robin Milner showed, surprisingly, that there is a way to transform non-parallel λ-calculus expressions, such as our Scheme source code, to a parallel version in the π-calculus [1]. Since it has the same basic features and properties, we will substitute the π-calculus for Core Erlang.

To implement the translation, we developed an optional pass over the Core Erlang AST. The translation itself works by rewriting the three rules behind the λ-calculus into corresponding expressions in the π-calculus. The three rules concern: (1) closed terms, which can be thought of as atomic, primitive expressions; (2) lambda abstractions, which can be recognized as function definitions; and finally (3) applications, which are the result of substituting lambda expressions into parameters named in an application.

A translation from these into π-calculus is not entirely obvious. In most treatments there is little time spent on motivating the intuition for the π-calculus expressions, and working them out on paper is essential. The takeaway is simply that these three rules become sequences of processes communicating over named channels. Closed terms can be thought of as sending their value as a channel. Abstractions can be seen as receiving data on a channel and then rebroadcasting it on another channel. Applications are responsible for setting up multiple processes to scaffold the actual reduction of expressions.

We instituted the translation pass as a walk over the Core Erlang AST, matching against atomic terms, abstractions and applications, and inserting the send and receive code based on the translation. For instance, having parsed the AST, we know how many functions there are and of what arity. Given that the translation splits function applications into a few parallel processes, we know we need to generate top-level spawn functions and store their process ids. We know abstractions and applications require sending information on channel names, so we traverse to those points in the code and insert the sends and receives to the appropriate process ids.

While the above explanation is somewhat cursory, we give an example translation of the identity function into its parallel version in Listing 3 and Listing 4. The second listing shows the multiple spawn calls needed to create processes, as well as how the functions were translated into code that sends and receives their values. Compiling this code to BEAM bytecode and running it on the BEAM results in the number 3117 as output.

```
(define (id x) x)
(define (run) (id 3117))
```

Listing 3: Identity function written in Scheme.

```
module 'id2' ['id'/0,'run'/0]
  attributes []
'id'/0 = fun () ->
    receive
      <{X,R}> when 'true' ->
        do
        call 'erlang':'!'(R,X)
        apply 'id'/0()
      after 'infinity' ->
        'true'

'run'/0 = fun () ->
    let <Lhs_id> =
      call 'erlang':'
        spawn'('id2','id',[])
    in
      let <_recv@2> =
        fun () ->
          receive
            <X> when 'true' ->
              X
            after 'infinity' ->
              'true'

      in
        let <Rhs_id> =
          call
              'erlang':'spawn'(_recv@2)
        in
          call 'erlang':'!'
            (Lhs_id,{3117,Rhs_id})
end
```

Listing 4: Scheme identity function parallelized according to Milner's π-calculus encoding and translated to Core Erlang.

5 Discussion

In reflecting on the project as a whole there a few things worth noting. First, the choice to use OCaml as our source compiler language resulted in several trade-offs. Erlang appears to be a natural choice for writing a source compiler for a language to Erlang's VM. The biggest advantage is that the source compiler does not need to specify and reimplement Core Erlang. Language designers can directly use the Core Erlang source code files from the Erlang compiler in their implementation. From a language runtime perspective, this choice also simplifies the runtime environment since users only rely on the Erlang ecosystem for development.

On the other hand, we chose OCaml because it is a language we are both familiar with—especially for writing compilers, interpreters, and program analyses. Given the time frame for the project, we felt we would be most productive using a language we are more familiar with, rather than adding more ramp-up time to learn how to write a compiler in Erlang. From a learning perspective, having to specify and implement the abstract and concrete representations of Core Erlang in OCaml taught us a lot about the internals of Erlang and was an unexpected benefit.

As for the second half of our project, implementing Robin Milner's translation, there were two main difficulties. The first challenge was understanding the "why" of how it works. We were only able to validate a small example used in the original paper, namely the identity function. Part of this difficulty results in the way Robin Milner employs syntactic substitution over expressions intermixed in the actual reduction semantics. It quickly became too complicated to verify for larger examples without implementing a full-blown π-calculus interpreter. Nevertheless, we found Vasconcelos [2] to be an excellent resource for learning about the language and its translation. The second difficulty concerned the actual implementation of the pass in OCaml. We came up with a few different ways of organizing how processes are spawned and how channel names are generated and transferred. It is not clear we chose a scalable solution and this could be worth revisiting.

6 Future Directions

Our project mainly demonstrates two observations. The first is that writing a source compiler targeting the BEAM is fairly painless. While the Core Erlang error messages are opaque, debugging non-compliant Core Erlang code never required too much effort. Moving from Scheme was, as expected, a breeze, given its easily parseable syntax and its functional programming semantics. The second observation contrasts strongly with the first, however, in that we found it to be very difficult to follow Robin Milner's translation from the λ-calculus to the π-calculus. Going forward we would like to gain confidence in our approach and then translate more complicated programs into π-calculus versions in Core Erlang.

To this end we would like to explore the actual behavior at runtime of our parallelized versions. Intuitively, turning a sequential program into multiple processes sending and receiving data bears a conceptual resemblance to the continuation passing transform. It would then be interesting to see if there is any impact on performance, or if the translation highlights actual opportunities for more traditional parallelization. Incorporating what we've learned into our PL toolboxes is a distinct ongoing interest of ours.

References

[1] R. Milner. Functions as processes. *Mathematical structures in computer science*, 2(2):119–141, 1992.

[2] V. T. Vasconcelos. The call-by-value λ-calculus, the secd machine, and the π-calculus. Technical Report TR–00–3, Department of Informatics, University of Lisbon, May 2000.