VRIJE
UNIVERSITEIT
BRUSSEL

Graduation thesis submitted in partial fulfilment of the requirements for the degree of de Ingenieurswetenschappen: Computerwetenschappen

DISTRIBUTING REACTIVE HAAI

Haai Virtual machine

Hans

June 2024

Promotors: prof. dr. Wolfgang De Meuter Supervisor: Bjarno Oeyen

sciences and bioengineering sciences

Abstract

We are looking into the distribution of a purely reactive programming language Haai.
Context . **inquiry** . **Approach** . **Knowledge** . **Grounding** . **Importance**

Contents

Chapter 1

Introduction

This thesis steps into the world of distributed reactive programs. Reactive programming revolves around the concept of reacting on streams of data. In reactive programming the program reacts on a stream of data. The stream of data travels trough the reactive program to exit in an altered state. The reactive programming language used is Haai (Oeyen et al., 2024), a purely reactive language consisting of only reactors.

1.0.1 Statement of the problem

Distributing software has multiple well known issues.

1.0.2 Definition of terms

Pure reactive language Distribution

1.0.3 Literature review

1.0.4 Remaining chapters

This is the introduction (Oeyen et al., 2022a) and so on

Chapter 2

Haai

We are making use of the novel purely reactive language Haai. Haai is reactive all-the-way-trough. As such the only construct available for expressing computations is a reactor. A reactor has sources or inputs and sinks or outputs. For each time the sources are updated the reactor will react and produce a new sink value. The number of sources and sinks are variable per reactor definition. In Haai there are no functions only reactors. This allows us to build reactive programs without functions. (Oeyen et al., 2024)

Two common problems encountered within reactive programming are by design eliminated. These two problems aer called the *Reactive Thread Hijacking Problem* and the *Reactive/Imperative Impedance Mismatch.*(Vonder et al., 2020) Two problems that arise from mixing reactive code with sequential code in a two-layered reactive programming language.

Reactive Thread Hijacking Problem

This happens in a two-layered reactive programming language when the reactive runtime or top layer calls a function in the bottom layer and that call never returns. The host has hijacked the control and the reactive runtime will not be able to regain control.

Reactive/Imperative Impedance Mismatch

When the reactive runtime in a two-layered reactive program used different side effects by calling functions on the first layer. It might be so that the order in which they are executed is not as expected. Creating faults in the programs that might not be easy to detect.

2.1 Haai instruction set

After compiling a Haai program into its bytecode everything is based around this instruction set. The instruction set defines the set of low-level operations that the virtual machine can execute. Each instruction corresponds to a specific operation. the Haai virtual machine presented in the next chapter is build around this instruction set.

[**I-ALLOCMONO, reactor**] Allocate memory for the given reactor.

[**I-LOOKUP, signal**] Lookup the value for signal in a signal table.

[**I-SUPPLY, from, destination, index**] Supply the value found in from to destination at index.

[**I-REACT, memory_location**] Apply the reaction required at memory_location.

[**I-CONSUME, memory_location, index**] Move the value form REACT to the run-time-memory.

[**I-SINK, rti_location, sink_index**] Remove value from run-time-memory and store in the reactor sink.

2.2 Reactors

In Haai the only construct available for expressing computations is a reactor. Such a reactor consists of the following parts. There are the sources and the sinks, they can be seen as what goes in and what comes out of the reactor Or the sinks will hold the altered value that was found in the source. The reactor has two possibilities to react on the values found in the source. The reactor can use a user defined reactor or a native reactor. For example in the Haai virtual machine we have native reactors for basic arithmetic. The memory size of a reactor is predefined by the number of instructions it needs to run.

2.2.1 Reactors memory

The in size fixed memory block of a reactor exists of the following four parts: sources, deployment-time-memory (dtm), reaction-time-memory (rtm) and sinks.

Chapter 3

Haai virtual machine

The Haai virtual machine (Hvm) is developed in the Elixir computer language. elixir runs on top of the Erlang Virtual Machine called BEAM. The BEAM virtual machine provides features such as lightweight processes, scheduling, memory management, and fault tolerance. This will allow us to explore the possibilities for working with the Haai computer language in a distributed environment.

For the development of this virtual machine inspirations was found in a different virtual machine for Haai called Remus. Remus i a virtual machine that has been carefully designed to be usable for running reactive programs in low-powered computing environments.(Oeyen et al., 2022b)

3.1 Example bytecode program

After parsing the bytecode that was compiled form the original Haai code with the Haai compiler into an Elixir readable format of nested lists a program could look like this. In this example we see two user defined reactors named consonance and note_length followed by the main reactor that functions as the main reactor to be executed.

```
co_nl = [
[:consonance, 1, 1,
[
    ["I-ALLOCMONO", :multiply]
],
[
    ["I-LOOKUP", :ci],
    ["I-SUPPLY",["%RREF",1],["%DREF",1],1],
    ["I-SUPPLY",["%SRC",1],["%DREF",1],2],
    ["I-REACT",["%DREF",1]],
    ["I-CONSUME",["%DREF",1],1],
    ["I-SINK",["%RREF",5],1]]
],

[:note_length, 1, 1,
[
    ["I-ALLOCMONO", :divide],["I-ALLOCMONO", :multiply]
```

```
],
[
    ["I-SUPPLY",60000,["%DREF",1],1],
    ["I-SUPPLY",["%SRC",1],["%DREF",1],2],
    ["I-REACT",["%DREF",1]],
    ["I-LOOKUP",:lm],
    ["I-SUPPLY",["%RREF",4],["%DREF",2],1],
    ["I-CONSUME",["%DREF",1],1],
    ["I-SUPPLY",["%RREF",6],["%DREF",2],2],
    ["I-REACT",["%DREF",2]],
    ["I-CONSUME",["%DREF",2],1],
    ["I-SINK",["%RREF",9],1]]
],

[:main, 2, 2,
[
    ["I-ALLOCMONO", :consonance],["I-ALLOCMONO", :note_length]
],
[
    ["I-SUPPLY",["%SRC",1],["%DREF",1],1],
    ["I-REACT",["%DREF",1]],
    ["I-SUPPLY",["%SRC",2],["%DREF",2],1],
    ["I-REACT",["%DREF",2]],
    ["I-CONSUME",["%DREF",1],1],
    ["I-SINK",["%RREF",5],1],
    ["I-CONSUME",["%DREF",2],1],
    ["I-SINK",["%RREF",7],2]
]
]
]
```

The above is an example of bytecode that is send to the Haai virtual machine. This bytecode is a compilation of the following Haai program defining two user defined reactors and joining them together within the main reactor. The main reactor could be seen as the 'main' function in C. The starting point or the call that kicks off the programs execution.

```
(
(defr (consonance f)
      (out (* ci f)))

    (defr (note_length bpm)
       (def q (/ 60000 bpm))
            (out (* lm q)))

    (defr (main f bpm)
       (out (consonance f))
       (out (note_length bpm)))
)
```

3.2 Native reactors

The Haai virtual machine makes use of native reactors to perform simple arithmetic operations like addition, division, multiplication and subtraction. Native reactors are recognized by the Hvm.

Chapter 4

Distributed Haai

The example program is explained in more detail in the use case chapter. The same program or reactor can be run many times. The distributed setup we make exists out of a number of elixir nodes that all send there messages to one sound server. The bigger the distributed network is the more messages that are send to the sound server. The sound server it self has virtually no limit to receive messages. The main parts of the distribution are taken care of by the underlying Beam virtual machine. One main reason to develop the Hvm on this platform is to be able to distribute in a solid and proven way.

4.1 Actor model

The actor system we use is Erlang which implements the processes actor model. (De Koster et al., 2016) These actors are modeled as processes and can define there receive primitive to specify what messages it responds to. Thanks to the isolated turn principle the actor model guarantees to be free of data races and deadlocks by design.

Chapter 5

Use case, sound

Sound or the musical expression of notes is the use case for this thesis. On two incoming streams of data reactors will react. One stream exists out of numbers representing frequencies to be played by the synthesizer. The second stream are numbers that represent the tempo in beats per minute, from that tempo durations for sounds can be computed by the reactor. Those durations are the time a sound plays until it disappears, secondly that same duration will be the time one reactor iteration takes. The reactor loops over and over but each loop has a defined duration, this duration is virtually equal to the duration calculated by the reactor.

The synthesizer that produces the sound is loaded inside of sound server named SuperCollider. Supercollider is an open-source platform for audio synthesis and algorithmic composition. Developed by James McCartney in the late 1990s (Wilson et al., 2011), it has since become a powerful tool in the fields of music technology, computer music, and sound art. Supercollider provides a flexible and expressive environment for creating and manipulating sound in real-time, making it an invaluable resource for both research and artistic exploration. Some main features of super collider include:

Audio Synthesis: Supercollider offers a wide range of synthesis techniques, including additive, subtractive, granular, and physical modeling synthesis. Users can create complex sounds by combining these techniques and modulating parameters in real-time.

Real-time Processing: One of Supercollider's key strengths is its ability to process audio in real-time. This makes it suitable for live performances, interactive installations, and other time-sensitive applications.

Algorithmic Composition: Supercollider provides tools for generating music algorithmically, allowing users to create compositions based on mathematical algorithms, rulesets, or generative processes.

Integration with External Hardware: Supercollider can interface with external MIDI controllers, audio interfaces, and other hardware devices, enabling users to incorporate physical instruments and sensors into their sound projects.

Community and Documentation: Supercollider has a vibrant online community of users who share code, tutorials, and resources. Additionally, comprehensive documentation is available, including tutorials, reference guides, and examples to help users learn and master the software.

Academic use of SuperCollider

SuperCollider has been and is actively used in academia, for example with the following purposes:

Research in Music Technology: Supercollider is widely used in academic research projects exploring topics such as digital signal processing, human-computer interaction, and machine learning for music. Researchers leverage its flexibility and programmability to prototype new algorithms, experiment with novel synthesis techniques, and investigate the perceptual and cognitive aspects of sound.

Composition and Sound Design: Composers and sound designers in academia use Supercollider to create innovative works that push the boundaries of traditional music and sound art. Its ability to generate complex and evolving textures, as well as its support for algorithmic composition, makes it a valuable tool for exploring new sonic territories.

Teaching and Learning: Supercollider is increasingly incorporated into music technology and computer music curricula at universities and colleges worldwide. It provides students with hands-on experience in sound synthesis, programming, and digital audio processing, helping them develop technical skills and creative problem-solving abilities.

Interactive Installations and Performances: Academic institutions often host exhibitions, concerts, and multimedia installations that feature interactive audiovisual experiences. Supercollider is frequently used to create the interactive soundscapes and generative audio environments that form an integral part of these events.

Conclusion

Bibliography

De Koster, J., Van Cutsem, T., & De Meuter, W. (2016). 43 years of actors: A taxonomy of actor models and their key properties. *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 31–40. https://doi.org/10.1145/3001886.3001890

Oeyen, B., De Koster, J., & De Meuter, W. (2022a). Reactive programming on the bare metal: A formal model for a low-level reactive virtual machine. *Proceedings of the 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, 50–62.

Oeyen, B., De Koster, J., & De Meuter, W. (2022b). Reactive programming on the bare metal: A formal model for a low-level reactive virtual machine. *Proceedings of the 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, 50–62. https://doi.org/10.1145/3563837.3568342

Oeyen, B., De Koster, J., & De Meuter, W. (2024). Reactive programming without functions. *The Art, Science, and Engineering of Programming*, *8*(3), 11. https://doi.org/10.22152/programming-journal.org/2024/8/11

Vonder, S. V. d., Renaux, T., Oeyen, B., Koster, J. D., & Meuter, W. D. (2020). Tackling the awkward squad for reactive programming: The actor-reactor model [ISSN: 1868-8969]. In R. Hirschfeld & T. Pape (Eds.), *34th european conference on object-oriented programming (ECOOP 2020)* (19:1–19:29). Schloss Dagstuhl–Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2020.19

Wilson, S., Cottle, D., & Collins, N. (2011). *The supercollider book*. The MIT Press.