The design and implementation of a variant of the OCCAM programming language

Jaime Valdemoros Gomez

2017

Abstract

We provide an implementation of a compiler and virtual machine for a subset of the original OCCAM language, with some additional features. The focus of research is on designing trust in the compiler implementation, using advanced Haskell features and proof techniques to convince ourselves that the construction is correct. While Haskell is not traditionally a proof-checking language in the way Coq, Agda or related languages are, we will explore a number of GHC-specific options that may be enabled to simulate such features.

Over the course of the project we also explore a number of *functional pearl* papers and evaluate their applications to projects larger than those usually studied in such papers.

Contents

Chapter 1

Introduction

Chapter 1 gives an introduction to the original OCCAM language, while Chapter 2 gives a specification of the variant of the language that this project aim to implement. Chapter 3 covers the design of the Virtual Machine, including the instruction set we provide and the overall design, followed by a proposed scheme for Chapter 4 forms the majority of the work, and covers the design and implementation of the Compiler, including techniques used to build trust in the correctness of the implementation. Chapter 5 deals with evaluation and testing, and the conclusions and lessons learned from the project.

1.1 Overview

The OCCAM programming language was developed at INMOS in the 1980s. It was inspired by Tony Hoares's CSP, with the key feature of making sequencing and parallelisation pf procedures fully explicit, with information being shared between processes primarily via channels. Together with the language, INMOS developed a *transputer* that would run a compiled version of this language.

Thanks to the explicit sequencing and parallel constructs, a set of interconnected transputers could then automatically distribute the workload among themselves, with communication between channels implemented via communication through links between transputers. A program could then be run concurrently on one or many transputers with little work on the part of the programmer. A single transputer would implement concurrency internally via time-slicing and a set of transputers could provide 'real' concurrency.

The aim of this project is to design and implement a virtual machine to simulate a single transputer, along with a compiler targeting this virtual machine. The instruction set of the virtual machine is loosely inspired by the instruction set of the INMOS transputer, albeit much simplified and with some changes to take advantage of the fact that we expect the virtual machine to be used on a modern computer. This implementation aims to remove some of the limitations on the original language, namely by implementing indexed repetitions of some constructs as well as mutually recursive functions and procedures.

Chapter 2

Language overview

The OCCAM language defines a *process* as being either one of several primitives, or a collection of other processes under one of several constructs [2].

2.1 Primitives

OCCAM has 6 primitives:

`<variable-refs>`	`:=`	`<expressions>`	Assignment
`<variable-refs>`	`:=`	`<f-call>`	Multiple function assignment
`<channel-ref>`	`!`	`<expr>`	Channel sending
`<channel-ref>`	`?`	`<variable-ref>`	Channel receiving
	`SKIP`		Do-nothing process
	`STOP`		Error process
`P(<arg>,`	`<arg>,`	`..)`	Procedure calls

A variable reference is either an identifier for a non-array variable, an identifier for an n-dimensional array indexed by n expressions. A channel reference similarly points to a single channel or a channel in an array. An assignment may assign the result of multiple expressions to multiple identifiers simultaneously, in which case all of the values will be computed before being any one of them is stored to its corresponding location. This means that, as long as no single location is repeated on the left-hand side, the effect of a multiple assignment is fully deterministic[1].

An OCCAM function is allowed to return any number of values of any type. Single-return functions may be used in expressions, while multi-return functions may only be used in an assignment to the corresponding number of variables. As in the case of the first form, the effect is well-defined as long as the same location is not referenced twice on the left hand side.

Given a channel, a process may use the final two constructs to write to and read from the channel. Channel inputs and outputs are *synchronised*. Given a channel output, a process will not continue until a corresponding process has accept ed the value, and vice-versa. We include a 'unit' type which may be used solely for synchronisation. On the send side, the unit value is a legal expression, while on the receiving side, we allow the special form { `ch ? ()` } to indicate the result should be discarded.

[1] This statement has a couple of potentially subtle 'gotcha's. An assignment {`a[i], i := 10, 3`} is perfectly legal as the 'location' `a[i]` is guaranteed to be computed before `i` is assigned to; on the other hand the semantics of the assignment {`a[i], a[j] := 10, 3`} are well-defined only if `i =/= j`. Since we allow arbitrary expressions in array indexes, it is not in general possible to deduce at compile-time whether such an assignment is well-formed without banning multiple uses of the same variable/array identifier altogether or requiring constant array indices.

There are two final primitives: `SKIP` and `STOP`. The first acts as a useful do-nothing process. The second raises an exception; whether this stops the current process or the whole machine is down to the implementation.

2.2 Composition constructs

2.2.1 The usual suspect

Of course, OCCAM would not be very interesting if it only allowed had single atomic processes. We may use `WHILE` loops in the same way as most languages, with the limitation that the body is a single Process (which may itself be a sequential or parallel composition). `IF` constructs are of a slightly less common form: an `IF` keyword is followed by a number of branches and processes. Each branch is checked in turn and the first one that evaluates to `TRUE` has the corresponding process run, defaulting to `STOP` if none matches. See the following for an example:

```
WHILE b
    ch ? b
```

```
IF
    b > 0
      ch ! b
    b < 0
      ch2 ? b
```

2.2.2 Sequential and Parallel composition

As described above, processes composition is explicitly sequential or parallel. Given a number of processes $P_1, P_2, ..$ or a process P_i with a free variable `i` of type `INT`, the following are legal constructs:

```
SEQ
    P1
    P2
    ..
```

```
SEQ i FOR <expr> TO <expr>
    Pi
```

```
PAR
    P1
    P2
    ..
```

```
PAR i FOR <expr> TO <expr>
    Pi
```

The indexed constructs run a number of copies of their argument in sequence or parallel with a new variable i bound to each value between the two expressions. If the first is greater than or equal to the second, this is equivalent to a `SKIP` construct.

Note that in OCCAM, indentation is important and any line 'scope' captures any lines directly beneath it that are further indented. These constructs have the obvious semantics, where in the replicated case, the variable `i` is 'fresh' and shadows any previous variable of the same name.

Parallel composition in OCCAM is *barrier synchronised*: a process that runs a number of sub-processes in parallel will not continue until all have completed.

2.2.3 Alt constructs

The most useful construct we have is the `ALT` construct. Where a query process blocks on a single channel, an `ALT` process blocks on a number of channels, non-deterministically choosing between the active ones or suspending until one of its channels becomes active. We have two additional forms of branches: a timeout branch that allows the process to unsuspend after a certain amount of time, and a `SKIP` branch that simply continues without waiting for a channel. We then allow each branch to be *guarded* by a Boolean-typed expression, with the notation `<test> & <branch>`

The non-determinism described does not make any guarantees about which channel would be chosen if there is more than one available. Instead, this is down to an implementation detail. The only guarantee we make is that if there is at least one `SKIP` branch active when the process reaches the `ALT` construct then one of these will be taken, rather than any active channel.

Unlike some modern implementations of `ALT` constructs, the OCCAM implementation only allowed branches to read from a channel, avoiding the complex (and time-consuming from a transputer perspective) process of negotiating the end state if a pair of `ALT` constructs met each other (or worse, multiple branches matching each other on a single or many `ALT` constructs simultaneously).

```
ALT i FOR <expr> TO <expr>
    i > 0 & ch[i] ? x
        process(i)
```

```
ALT
    b > 2 & ch ? x
      process(2)
    b < 2 & ch2 ? y
      SEQ
        ch3 ! y
```

Here we have two generalisations to the ALT construct. Firstly, in keeping with the SEQ and PAR constructs we add the possibility of having an indexed ALT which computes a guard and branch for every value of a given index in some range. The second is in keeping with the implementation of `ALT` constructs in some modern languages: we add the capacity to nest ALT constructs, both in unindexed and indexed form and combinations thereof. We still maintain the property that if an active SKIP branch is encountered none of the other channels would be chosen.

```
ALT
    ALT i FOR <expr> TO <expr>
      prime(i) & ch ? x
          process(i, x)
    ch ? y
      process(2, y)
```

2.3 Functions and Procedures

Both the original OCCAM and this implementation take a principled view on the role of functions and procedures: Procedures in OCCAM are more like what we usually consider subroutines, with no return values, and functions are 'pure' in that they should have no side effects and their outputs are determined only by their inputs. We can do this by disallowing all parallel constructs and non-constant variables. We do allow functions to contain STOPs since this is morally equivalent to returning a Boolean value indicating whether the calling procedure should raise an exception.

A function is made up of a sequence of declarations, on a single line, a body which is a single process and a `RETURN` value or sequence of values. The parameters of a function are made up of the keyword `VAL`, followed by a type and the name to be given to the variable.

We allow procedures to specify whether variables are passed by value (with the `VAL` keyword) or by reference (with the `VAR` keyword). When calling, a `VAL` may be any expression that evaluates to the right type, while a `VAR` must be passed a variable or array, again of the right type. We allow sub arrays to be passed as well; for example, we could declare a 5-dimensional array and

pass the 3rd index to a procedure expecting a 4-dimensional array. `CHAN` parameters are suffixed with a `!` or `?` to indicate whether they are to be used for input or output.

```
INT, BOOL FUNCTION f(VAL INT x)
    INT x, BOOL b
    IF
      <expr>
        STOP
      True
        x = 5
    RETURN x, False
```

```
PROC p(VAL INT x, VAR [][]BOOL a, CHAN []INT x?)
  SEQ
     x, b := f()
     IF
       b
         STOP
       True
         SKIP
```

A valid program is a collection of functions and procedure (given unique identifiers), including a single procedure taking no parameters and given the identifier `main`. These may be given in any order, since any function may call any other function and likewise for processes.

2.4 Channel usage

For a number of reasons, the original OCCAM implementation had a specific requirement on how to use channels: only one process may attempt to read from, and one process may attempt to write to, a channel at any one time. In combination with the restriction on ALTs to only reading channels this gives a much simplified set of possibilities to consider: instead of any number of processes trying to connect to each other in an arbitrarily sized web, we can only come across situations with either two processes trying to connect to each other, or a single process with a finite number of ALT branches, each with one or no processes at the other end.

2.5 Input and Output

For an initial proof of concept it will be sufficient to be able to read from the standard input, write to the stand output and access some sort of timer (for the timed branches). For these purposes we can stick with the OCCAM convention of providing global access to channels names `keyboard`, `screen` and `tim`. This allows us to stick with the same notation as the rest of the program while being able to convert them 'under the hood' into their specialised machine code instructions so that they aren't limited to one process at a time as they are with normal channels.

Chapter 3

Virtual Machine design

3.1 Language choice

While functional languages are best suited to the kind of data manipulation we need in the compiler, an imperative language is best suited for actually running the code. For speed and efficiency it makes sense to use a compiled language, with the additional benefit of being able to compile it on one platform and run it on a different platform[1] without needing carry over any kind of interpreter[2].

This leaves us with few widely-used languages: C, C++ and Rust. Rust's safety features and modern capabilities (immutable by default, algebraic data types, pattern matching, first class iterators) make it a strong contender against the more established languages: there is a reason Rust is the systems language *du jour* at the time of the writing.

3.2 Memory layout

The first step in designing such a virtual machine is deciding on the memory semantics. As described above, the compiler will assume the existence of an unbounded stack available to each existing process. While a common, and convenient, way of storing local variables is in local stack frames, the parallelism in the compiled program makes this harder to implement. This is because frames may not be dropped in the same order as they are created. This can be partially remedied if we know exactly how much space a process will take up, since then we can allocate that much space to a process and allow it to use stack frames internally for its function calls. However, since we would like to allow recursive and mutually recursive functions, and allow functions to be placed inside expressions, it is in general impossible to do so: finding an upper bound for the stack size at compile time would be equivalent to solving the halting problem.

Instead, we write a simple allocator/deallocator pair of functions, allowing processes to request and drop data frames at arbitrary times with `NewFrame` and `DropFrame` respectively, where `NewFrame` takes a value from the top of the stack representing the size the frame should have. These frames form a sort of tree, with processes working on their final leaf frames. Each frame contains a pointer to the previous frame, which can be accessed using the `FP` operation: this copies the frame pointer to the previous frame onto the stack. We can use `Local(n)` notation to access values on the most recent frame, otherwise using pointer arithmetic on the `FP` value

[1] in the long-term the virtual machines could be run on small single-board computers, such as the Raspberry Pi. If they could be networked to communicate between themselves we would essentially have a very similar setup to the original OCCAM transputers. In fact Rust makes it extremely easy to target such platforms.

[2] Of course while such setups exist, it seems a bit inefficient to run a language on a virtual machine that is itself interpreted in a virtual machine.

to access values on other frames further up the tree.

For simplicity, we provide `NewArr`, `NewChan` and `NewChanArr` operators, for creating arrays, channels and arrays of channels respectively, in each case leaving a relevant pointer on the stack[3]. While the only strictly necessary operation is NewChan, as data frames can be used to store array entries, it is helpful to avoid creating a large number of data frames that would need to be 'followed back' to get to a specific variable.

While frames can be dropped manually, we choose to make arrays and channels implicitly dropped when the process that created them terminates. Due to the barrier synchronisation on `PAR` branches, sub-processes can copy channel and array pointers around without worrying about the array or channel will be dropped; however, these pointers and arrays should not be sent via channels since the elements they are referring to may be dropped before they are accessed.

3.3 Instruction set

Having decided on a memory layout (and associated operators) we can now decide on the rest of the operators we want to make available. We split these into four parts: Stack operators, Memory operators, Tape operators and Process operators.

Stack operators define the usual operations on our unbounded stack. These are fairly standard so we wont spend much time on them; a full description of their operations is available with the full compiler [1].

Additionally we have three special-case stack operators: `putChr` writes a character (from the top of the stack) to the standard output without blocking, while `getChr` reads a character from the standard input. `getTime` places a value representing the current time on the stack.

We have two 'tape' operators: `JUMP` and `CJUMP`. The first takes a value from the top of the stack and jumps to that location. The second takes a value and checks whether it is non-zero; if so it performs a `JUMP` on the next value on the stack, and otherwise it simply pops off the value and discards it.

Finally we have our 'process' operators. `ENDPROC` simply terminates the process and indicates to the virtual machine that the process and its stack can be discarded. `CHANREAD` and `CHANWRITE` take a label off the stack and attempt to read from or write to it respectively, with `CHANWRITE` popping another value off the stack to write to. In both cases the virtual machine suspends the process until it can match the read/write with another process using the same channel label for the opposite operation.

To deal with parallel and alt constructors, we provide `STARTALTS` and `STARTPARS`. These don't actually have an effect[4], but do indicate to the virtual machine that it should expect a sequence of `AltBranch(n)` and `TimBranch(n)` in the first case, or `ParBranch` in the second case. All three take a value from the stack indicating to the virtual machine what position in the code the continuing process should jump to. The `n` parameter in both commands indicates to the virtual machine that it should take n values off the stack and store them away; if this branch

[3] In the case of `NewArr` and `NewChanArr`, a pointer to the array, and in the case of `NewChan`, a label indicating the identity of the channel. The array created by NewChanArr contains a label for each channel.

[4] We could equally remove these and have the compiler deduce the fact from the first `AltBranch(_)` or `ParBranch(_)` opcode, however this provides a useful sanity check.

ends up being chosen then the virtual machine should place these back on the stack[5]. Finally the `ENDALTS` and `ENDPARS` actually trigger the construct: in the first case the process suspends until at least one of the declared branches is triggered, or enough time has passed to trigger a timed branch, and in the second it suspends until all of the children described by the `PARBRANCH` commands have completed and executed `ENDPROC`.

Finally, we have one additional command. While the `ALT` commands allow an arbitrary number (including 0) of values on the stack to be stored away and returned to the process, we have made the `PARBRANCH` command weaker by not storing away any indexes at all, meaning that we can't yet implement indexed `PAR` loops. Instead we provide a `PARINDEX` command: this takes two values i and j and a program position *pos* off the stack, and runs $j - i$ processes each starting at program position *pos* with a value between i and j on its stack.

```
pub enum StackOp {
    PutChr,
    ReadChr,
    GetTime,
    NOP,
    SWAP,
    DUP,
    INC,
    DEC,
    ADD,
    MUL,
    DIV,
    MOD,
    NEG,
    EQ,
    AND,
    OR,
    NOT,
    GT,
    LIT(StackVal),
}
```

```
pub enum MemOp {
    LDW,
    STW,
    LOCAL(u8),
    FP
    NewFrame,
    DropFrame,
    NewArr,
    NewChan,
    NewChanArr,
}
```

```
pub enum TapeOp {
    Jump,
    CJump,
}
```

```
pub enum ProcOp {
    ChanRead,
    ChanWrite,
    StartAlts,
    AltBranch(u8),
    TimeBranch(u8),
    EndAlts,
    StartPars,
    ParBranch,
    EndPars,
    IndexPar,
    EndProc,
}
```

3.4 High level design

From the point of view of the virtual machine, a program is simply a sequence of operation codes, and nothing else. Each process is represented by a Worker 'object', which keeps track of its own position in the program. New workers are created when a Worker objects comes across a `ParBranch` or `IndexPar` opcode. The virtual machine will guarantee that the very first instance of `NewFrame` is at memory location 0; after than the location cannot be predicted. The reason for this is that the 'root' process, i.e. the one starting at position 0 in the program, can then deal with setting up static variables and channels, whose positions can then be statically determined at compile time so that all the other call sites can access them. Once the root process has finished setting up the static values, it is expected to immediately jump to the start of the `main` function.

3.4.1 Strongly typed state machines

We can consider the workers and channels of our program as independent state machines, communicating with each other via the virtual machine. We essentially have 3 kinds of state: two

[5]This allows for nested indexed `ALT` constructs, to determine which values the indices had at the time the branch was declared.

explicit and one implicit. A Worker representing a process has a 'Ready' state as well as a number of 'Suspended' states, indicating the reason for the Worker not being able to run: either waiting for some channel (reading or writing), waiting for its children processes to complete, or waiting for an `ALT` construct to match some branch. The Ready state can be further divided into 'Normal', 'BeginPar', 'BeginAlt'; these represent whether have started building one of these constructs. On the other hand, a channel has three states: Ready, ReadWaiting (further split into waiting on a channel or an Alt construct) and WriteWaiting. We can represent the transitions with the following diagram:

While these state systems might seem superficially the same, they are actually quite different. It would be a fundamental error on the part of the Virtual Machine to treat a process that is waiting to read from a channel as if it were ready, or to try to force a channel read on a process that is waiting for its children to terminate. However, it is a *user* error to try to read from a channel that another process is already waiting to read from; while this is still an error, it is one we should be able to handle and respond accordingly.

To this end we describe a form of *internal* and *external* states. Workers have *external* state, and we should statically check what worker state a certain Rust function is meant for. To do so, we create a new type for each state, then group them under a single *trait*. While traits are usually used to define common functions for different types, here we simply use an empty trait to make it clear what is and isn't a worker state.

```
pub struct ReadyState { .. };
impl WorkerState for ReadyState {}

pub struct ParState {
    children: ...,
}
impl WorkerState for ParState {}
```

```
pub struct AltState {
    channels: ...,
    timer: ...,
}
impl WorkerState for AltState {}
```

We can then parameterise our Worker object by the state type:

```
pub struct Worker<S: WorkerState> {
    pub id: Uuid,
    pub parent_id: Option<Uuid>,
    pub state: S,
```

```
}
```

Note that each state object contains all the information relevant to that state; we do not leak any information into the rest of the Worker that doesn't carry over from one state to the next.

Once we have defined this, we can define our other data structures and functions in terms of the relevant Worker types: the list of workers waiting to run is a `Vector` of `Worker<Ready>` objects; a channel stores away a `Worker<Reading>` or `Worker<Writing>`.

Channels, on the other hand, maintain their state internally. And operation that would feasibly be run on a `Ready` channel could, due to some user or compiler error, end up operating on (for example) a `WriteWaiting` channel. For this reason, we have channels manage their state internally, via the following enumerated type `ChannelState`.

```
enum ChannelState {
    Empty,
    ReadWaiting(WritingWorker),
    AltWaiting(AltToken),
    WriteWaiting(ReadingWorker),
}

pub struct Channel {
    ch_id: uuid::Uuid,
    state: ChannelState,
}
```

The third, implicit state is in the `Ready` state for the `Worker`s. When a `Worker<Ready>` comes across a `StartAlts` or `StartPars` operator, it disallows any further Process operations that isn't one of the relevant branches. This falls under internal state, since we still need to explicitly deal with this case if by some user or compiler error it does come up.

3.4.2 Intrusive data structures

The strength of Rust's memory model has an important limitation: when a structure contains elements, most languages allow the elements to be 'aware' of their context in some way: they can manipulate the rest of the structure, or modify it in some way. In Rust this is not possible: in order to read or manipulate the surrounding structure, the object needs a reference to the surrounding structure. In order to run a method on the object, we also need to take a (mutable or immutable) reference to it. Since Rust only allows us to take at most one mutable, or many immutable, references to any value or sub-part of a value, it is then impossible to have the two references we need at once.

The kind of structure we are describing is called an *intrusive* data structure. An example of such a structure is a double-linked list, where each node contains pointers to two other nodes and this makes up the list itself. In order to sidestep this issue, we have a few options:

- 'Pull out' the object from the data structure, replacing it with some default value, so that we now 'own' that value. This can then take in a reference to the structure and manipulate it as necessary. In this approach we must be careful to remember to return the object back into the structure if we want to keep it. This is the approach taken by the `ChannelState` below, which defines a method `take()` that pulls out the state object from whatever is holding it, leaving behind a Empty state object.

- Instead of acting directly on the structure, we can enumerate the possible actions and have the object return one of these enumerations. We then have the surrounding data structure act on this data.
- We can simply not define methods that require this, instead lifting them to the 'nearest' wrapping data structure that can actually take mutable references to the relevant objects without being itself involved.
- Rather than storing data directly, we can use a vector as a 'backing structure', and have objects store labels indicating the position in the vector rather than a pointer directly. This is useful for simulating multiple pointers, however this means we give up some of Rust's safety guarantees by essentially simulating C's pointers.

```
enum ChannelState {
    Empty,
    ReadWaiting(WritingWorker),
    AltWaiting(AltToken),
    WriteWaiting(ReadingWorker),
}

impl ChannelState {
    fn take(&mut self) -> Self {
        // This standard function swaps in the value of the second
        // argument, pulling out whatever is stored in reference
        // the first argument
        mem::replace(self, ChannelState::Empty)
    }
}
```

Chapter 4

Compiler design

4.1 Language choice

When writing a compiler, a common choice is to write the compiler in the language you are compiling (the so-called *bootstrapped* compiler) as some sort of example of the usefulness of that language. Since the OCCAM language is fairly simple, and we wouldn't be able to bootstrap our compiler anyway since we don't have an existing compiler, the next best option is to write the compiler in a functional language.

The focus in this paper is on building trust in our compiler. Functional languages make this much easier, as the almost declarative nature of function definitions make it easier to reason about correctness of our code. At this point we have a trade-off between expressive power of the type system, which we hope to use to build the aforementioned trust, and language maturity: the Glasgow Haskell Compiler has decades of research in it and is a powerful optimising compiler, however its type system doesn't have as much expressive power as a full-blown dependently-typed language such as Coq or Agda.

Ease of use and familiarity with the language, as well as being able to escape dependent types when it suits us, pushes us to use Haskell as our language. However, we will aim to simulate some aspects of dependent types via certain GHC extensions.

4.2 High level design

Our compiler is composed of four principal functions, characterised by the data-types connecting their endpoints. We will make extensive use of GADTs (described below) to restrict the 'shape' of the data-types to only programs that are 'valid' in some sense.

The first stage is parsing the input string into simple data structure. This structure aims to capture the overall structure of the input code, while being as 'loose' as possible so that as many possible inputs can be captured as possible: we can later refine the structure progressively and in doing so provide more helpful error messages for the user.

The second state involves converting our 'loose' representation into a tightly-refine data structure that aims to filter out as many incorrect programs as possible. We will provide an introduction to GADTs, which will provide most of the power we need to tightly refine our data structures.

Once we trust our structure is guaranteed to represent a well-defined program, we can set about converting it into our virtual machine's stack representation. Just as we have two repre-

sentations for our input, we do the same for our machine code representation: one will be tightly restricted to make sure our translation doesn't try to use the machine code in the 'wrong', while the other can be converted easily into the binary representation we finally use.

Finally we will go about converting our stack representation into complete binary data, This involved flattening nesting, computing label positions and finally outputting the binary data.

The success criterion for this part of the project is that the third stage should be a total function that returns a meaningful output for any possible input. This means restricting our input data structure enough that any program represented in it is 'sensible' - in our case, we expect all function and process calls to refer to functions and processes that exist, with the right number and types of parameters, and all variables and channel calls refer to variables and channels that are in scope and of the right type.

4.3 Code parsing

In this section we will describe 3 different styles of string parsing that we use in the compiler. Each one has a different strength it may use in different situations.

4.3.1 Monadic parsers

Much of the parsing within the project is done using the excellent Haskell library Parsec, which defines a set of parser combinators with a monadic instance so that it can be used with Haskell's do-notation.

The Parsec library revolves around a much more generalised version of the following, which is itself a generalised version of the presentation given by G. Hutton and E. Meijer [3] :

```
newtype Parser tok m a = Monad m => Parser([tok] -> m (a, [tok]))
```

In this scheme, a `Parser tok m a` is a function that takes an input sequence of tokens, each of type `tok`, and returns a pair of a value of type `a`, read from some prefix of the input sequence, and the remaining suffix. The result is given within a monad. When specialised to the list monad (as in the original paper), this represents a choice between different parses; when specialised to an Either a monad, we can return precise error messages about what went wrong. Since the function is covariant in `m`, if `m` is a monad then so is the parser. For monads that implement failure or choice, we can define the following:

```
token :: MonadFail m => tok -> Parser tok m tok
token t = Parser (\xs -> case xs of
                         (x : xs) | x == t -> return x
                         _ -> fail "Token does not match")

choice :: MonadPlus m => Parser tok m a -> Parser tok m a -> Parser tok m a
choice p1 p2 = Parser (\xs -> parse p1 xs 'mplus' parse p2 xs)
```

With these two simple definitions, we immediately get a powerful set of combinators: the monad instance immediately gives us access to do-notation and Functor and Applicative instances for these parsers, so we can compose and map over simple combinators to build up more complex parsers.

We make extensive use of the framework as implemented in Parsec, along with its associated lexer, in our parser for the language.

4.3.2 Pratt parsers

A simple way to parse expressions is to use a Pratt parser. A Pratt parser uses recursive descent, but unlike a number of parsers that uses grammar rules to determine how to parse the string, a Pratt parser associates semantics to tokens themselves [4]. This is very useful when a token uniquely determines the data constructor it is associated with, and is also useful when tokens have different meanings in prefix and infix positions.

[1]To see how a Pratt parser works, we ignore for a moment the existence of unary operators and parentheses, and assume a valid input is simply a sequence of numbers of identifiers, interspersed with binary operators. We define a trio of mutually recursive parsers: `unaryHelper`, `exprHelper n` and `binaryHelper n`, giving them the following invariants:

- `unaryHelper` simply consumes a single token representing a variable or literal.
- `exprHelper n` reads the remaining stream until a binary token with precedence strictly less than n is found, then reads an expression from the prefix it has read, leaving the binary token on the stream.
- Given a stream whose head is a binary operator token, `binHelper n` inspects the head, say tok, without consuming it. If we have $prec(tok) < n$ it returns `Nothing` without reading. Otherwise it continues read the remaining stream until a token with precedence greater than $prec(tok)$ is found (without consuming it), interprets what it has read so far as an expression, and returns the previous binary operator together with this new expression.

Given these invariants, `binHelper n` has an obvious implementation: read the first token then run `exprHelper` with the precedence of the token as a parameter. We assume the existence of a function `binary` that consumes a binary token and returns its precedence together with a function of two arguments which can be used to form the binary structure. We use the Parsec functions lookAhead and try to inspect the next token without consuming it.

```
binary :: Parser (Int, R.Expr -> R.Expr -> R.Expr)
binary = ...

binaryHelper :: Int -> Parser (R.Expr -> R.Expr)
binaryHelper n = (do (k, f) <- (P.lookAhead . P.try $ binary)
                     if k < n then fail ""
                              else do _ <- binary -- actually consume the input
                                      e <- exprHelper (k+1)
                                      return . Just $ flip f e)
```

Having this function, we can now define `exprHelper n`: run `unaryHelper`, then run `binaryHelper n` repeatedly, collecting up the results, until it fails to return input. Each run of `binaryHelper n` will return a binary token of strength $\geq n$ along with the expression with only greater strength binding. Once binaryHelper fails then we are done and we return the result of having collected up the results. We can see that this implementation respects the invariant we have defined.

Finally we can define our expression parser as `exprHelper 0`. This ensures that we eventually capture all the relevant binary operators.

```
exprHelper :: Int -> Parser R.Expr
exprHelper n = unaryHelper >>= loop
```

[1]The following is my own interpretation of how and why Pratt parsers work. Credit goes to [4] for a description of how to construct a Pratt parser.

```
            where
              loop :: R.Expr -> Parser R.Expr
              loop e = do m_f <- P.optionMaybe (binaryHelper n)
                          case m_f of
                            Nothing -> return e
                            Just f -> loop (f e)

expr :: Parser R.Expr
expr = exprHelper 0
```

Once we have defined a working parser, we can extend it to include parentheses, function calls and unary operators:

The unaryHelper function is extended to deal with parentheses (inside which it may simply run exprHelper 0 again), and functions and arrays (by checking whether an identifier is followed by a parenthesis or a square bracket and working accordingly).

We can then extend the binaryHelper function to deal with ternary operators, in particular the `_?_:_` form, by adding a case for the symbol `?`, running the full expression parser, then checking for `:` and continuing as before (essentially treating the segment `?<e1>:` as a single binary token). This of course assumes that `:` is not a legal token elsewhere.

The beauty of this approach is that it is essentially trivial to distinguish between left- and right-associative binary operators. The approach above is left-associative: when `binaryHelper` reads the precedence of the first operator, it increments it before passing it to `exprHelper`, thus ensuring the same operator doesn't appear in the right-hand branch, since now the next time `exprHelper` comes across that operator it will stop and return what it has found so far. If instead we wanted to make an operator right-associative, we would pass k instead of $k + 1$ to the exprHelper, thus trapping subsequent repetitions of the operator in the right-hand branch.

4.4 GADTs

As the bulk of the compiler work will involve defining appropriate GADTs and functions operating on them, we'll give a short description of what they are and how they work.

Suppose we would like to represent an expression with a custom datatype. The following algebraic datatype is probably some of the most commonly written code in Haskell:

```
data BinOp = Plus | Minus | Div | Mod

data MonOp = Negate | Not

data Expr = Lookup String       // A variable
          | B Bool              // Boolean literal
          | I Integer           // Integer literal
          | Mon MonOp Expr      // A unary operator applied to an expression
          | Bin BinOp Expr Expr // A binary operator on two expressions
          | If Expr Expr Expr   // A choice operator, <e1> ? <e2> : <e3>
```

This is a perfectly valid datatype, and allows us to express many kinds of expression quite simply. However, this has a number of drawbacks:

[2] languages that suggest otherwise are silly and should be ignored

- The definition is fairly **opaque**: Looking at each line tells us nothing about what the sub-expressions look like. Some are obvious: it (usually[2]) makes no sense to add a Boolean to and integer, so we would expect the an expression of the form `Bin Plus <e1> <e2>` to have integer-typed sub-expressions. On the other hand, it isn't clear from looking at `If <e1> <e2> <e3>` which of `<e1>` or `<e2>` (or indeed `<e3>`) forms the test and which pair forms the branches.
- Even once we have established the above, we have to be careful throughout the code not to break the conventions we have set for ourselves. Especially when working with user input, we cannot assume the expression is well-formed, and it is not clear at what stage a 'filtering process' is meant to happen.

It is possible to alleviate some of the above by separating out the constructs into separate data-type, which gives us some of the benefits we are looking for, at the cost of some code duplication in both the data-types (for example a conditional operator for both `IntExpr` and `BoolExpr`) and in the functions working on them, as for each function we'll need one to work on `BoolExpr`, one for `IntExpr`, and more for each additional type we'd like to define.

Instead, we'll turn to one of the most commonly used GHC extensions: GADTs. These will allow us to combine the benefits of the single datatype with the benefits of static separation between similar but separate 'types' of that datatype. This is done by allowing us to parameterise the datatype `Expr` into a datatype `Expr a`. Here, any valid type can be put in the place of `a`. We then provide constructor functions that describe what kind of `Expr` is constructed:

```
{-# LANGUAGE GADTs #-}
data Expr a where
    B :: Bool -> Expr Bool
    I :: Int -> Expr Int
    Add :: Expr Int -> Expr Int -> Expr Int
    If :: Expr Bool -> Expr a -> Expr a -> Expr a
  ...
```

The form of an expression is then completely determined by the type parameter: a function that takes an `Expr Bool` and matches on it can ignore the irrelevant branches:

```
evalBool :: Expr Bool -> Bool
evalBool (B b) = b
evalBool (If test e1 e2) = if (evalBool test)
                           then (evalBool e1)
                           else (evalBool e2)
```

Here, the compiler can infer which constructors can form an Expr Bool and not warn us that we have missed out the other branches, while still letting us know if we have missed a relevant branch. More generally, we can allow the return type of a function to depend on the parameter on the Expr:

```
eval :: Expr a -> a
eval (B b) = b
eval (I i) = i
eval (If test e1 e2) = if (eval test)
                       then (eval e1)
                       else (eval e2)
eval (Add a b) = (eval a) + (eval b)
```

On each branch, unpacking the datatype constructor gives us information about the type parameter `a`. The compiler can use this to determine that we are returning a value of the right

type.

4.5 Defining our expressions

While the above is very useful, it doesn't quite match up to our original specification of an expression. In particular, we have not given a way of defining variables. This is because we do not yet have a good interpretation of what type parameter an expression of the form `Lookup String` would take. At this point we have a number of possibilities:

- Define the `Lookup` constructor as `Lookup :: String -> Expr a`. This makes `Lookup str` a polymorphic value in the sense that the caller can decide what `a` is set to, so (for example) `Lookup "x"` can be used anywhere an expression is expected.
- Use a second GADT to define some form of 'indicator' `TyMarker a` that we can use to enforce what type the variable has: `Lookup :: String -> TyMarker a -> Expr a`.
- Use some form of 'context' to work out what the type of the variable will be, and parameterise the `Expr` by this context. This approach is described in more depth below.

For this compiler we take the third of these approaches.

4.5.1 De Bruijn indices

In order to represent our variables, we are going to use something similar to De Bruijn indices. De Bruijn indexing is a way of representing lambda terms in a way that removes the need for variable names. Instead, a lambda abstraction is simply the λ symbol, and a variable reference is given by an integer representing how many 'layers away' the abstraction we are looking at is.

In this scheme, if we are looking inside n lambda abstractions, any integer in $\{0..n-1\}$ would be a bound variable, and an integer in $\{n..\}$ would be a free variable. This is key in our implementation: we can declare variables ('abstract out' in λ-calculus terms), while unbound variables are still made clear.

4.5.2 The DataKinds GHC extension

Before we get to defining variables, we will take a look at what exactly we can put into a GADT type parameter. As with most functional languages, values have types. What we can do is consider the 'types of types', generally called a '*kind*'. In some dependently-typed languages, such as Coq or Agda, this extends to an infinite hierarchy of universes; however, in Haskell this stops at *sort*[3] representing the type of kinds, which contains the unique value *BOX*, which is the *sort* of any Haskell *kind*.

Fortunately, we don't have to worry about this, and we only need to look at types and kinds. With the **DataKinds** GHC pragma, Haskell automatically promotes any suitable type to a kind, and its constructors to type constructors. Thus for example the following declaration, which represents natural numbers,

```
{-# LANGUAGE DataKinds #-}
data Natural = Zero | Succ Natural
```

[3] In Haskell, *sort* is then a member of itself, so the infinite hierarchy seen in other languages is made finite in Haskell. Due to a variation of Russell's paradox in type theory, called Girand's paradox, this means Haskell's type system is actually inconsistent. Of course this doesn't actually matter, since Haskell's type system was never consistent to begin with.

also generates generates the following:

```
Natural :: BOX
'Zero :: Natural
'Succ :: Natural -> Natural
```

We can then use a slightly different form of GADTs to indicate what *kind* each of its type parameters should belong to. This allows us to define the standard dependent-language example of arbitrary fixed-length vectors with the length encoded in the type (note that the *Type* is the Haskell name for the kind of our usual builtin and programmer-defined types and needs to be explicitly imported):

```
import Data.Kind(Type)

data Vect :: Natural -> Type -> Type where
    VNil :: Vect 'Zero k
    VCons :: k -> Vect n k -> Vect ('Succ n) k
```

This also extends to Haskell lists: enabling the extension gives us the type `'[]` and the type constructor `(':) :: k -> [k] -> [k]`, where `k` is any kind. Given a sequence of types $T_1, T_2, ..T_n$ of kind K, the type `[T_1 ': T_2 .. ': T_n]` has kind `[K]`.

This allows us to define the second major data structures that the compiler makes use of:

```
data HList (f :: k -> Type) :: [k] -> Type where
  HNil :: HList f '[]
  HCons :: f ty -> HList f tys -> HList f (ty ': tys)
```

Where HLists usually define heterogeneous collections of a fixed length and sequence of types, this variation allows us to describe a list of This is a generalisation of the usual HList construct for homogeneous lists to allow us to specify that the contents are values whose types include a certain type constructor. For example, given the above definition of `Expr :: Type -> Type`, a value of type `HList Expr [T1, T2, T3]` is a sequence composed of and `Expr T1`, an `Expr T2` and an an `Expr T3`.

4.5.3 Initial definitions for our expressions

Having gone through this, we can finally make our initial complete definition for expressions. We begin by defining a datatype that represents a De Bruijn index but which 'pulls out' a specific type from a list:

```
data ListPointer :: [k] -> k -> Type where
  Last :: ListPointer (t ': ts) t
  Prev :: ListPointer ts t -> ListPointer (t2 ': ts) t
```

Note that by using polymorphic types `ts` and `t2` we make no restrictions on how many or what other variables are 'in scope'. We define the types we will allow in our OCCAM language:

```
data BaseType = UnitTy | BoolTy | CharTy | IntTy
data VarType = Base BaseType | Arr VarType
```

Since we want to work with arrays and not just base types, we generalise the De Bruijn index slightly to allow 'peering in' to an array:

```
data Location :: [VarType] -> VarType -> Type where
  Var :: ListPointer vars ty -> Location vars ty
```

```
  Index :: Location vars (Arr ty) ->
           Expr vars 'IntTy ->
           Location vars ty
```

This allows us to not only point to a variable in our environment (using the `Var` constructor) but also, given a pointer to an array and an integer-typed expression, we can get a pointer to an entry in that array. Finally we parameterise our `Expr` type above by the context it 'expects' to be in and this gives us enough to work with to include variables:

```
data Expr :: [VarType] -> BaseType where
    Lookup :: Location context (Base ty) -> Expr context ty
    B :: Bool -> Expr context 'BoolTy
    I :: Int -> Expr context 'IntTy
    Add :: Expr context 'IntTy -> Expr context 'IntTy -> Expr context 'IntTy
  ...
```

Note that the definition of `Lookup` requires that the `Location` value points to a base type and not to an array value.

An evaulator function for this datatype can then ensure it has access to some sort of context, and use the type to assert that there is 'sufficient' context of the right form to evaluate the expression:

```
data Literal :: VarType -> Type where
    BoolLit :: Bool -> Literal 'BoolTy
    IntLit :: Int -> Literal 'IntTy
    ArrLit :: [Literal ty] -> Literal (Arr ty)

data Context :: [VarType] -> VarType -> Type where
    CNil :: Context '[]
    CCons :: Literal ty -> Context tys -> Context (tys ': tys)

evalPointer :: Context tys -> ListPointer tys a -> Literal a
evalPointer (CCons v _) Last = v
evalPointer (CCons _ vs) (Prev p) = evalPointer vs p

evalLoc :: Context tys -> Location tys a -> Literal a
evalLoc c (Var p) = evalPointer c p
evalLoc c (Index p expr) = case (evalLoc c p) of
                             ArrLit lits -> lits !! (eval c expr)

eval :: Context tys -> Expr tys a -> a
eval c (Variable pointer) = evalLoc pointer
  ....
```

Note that in all the functions, the restrictions on the type parameters means that we only need to give definitions for the cases that can exist. In the second branch of the `evalLoc` definition, we know that the pointer `p` has as its second type parameter `Arr ty` for some `ty`, as otherwise we could not have constructed an `Index p expr` value. This means that the recursive step will give us an `ArrLit` back and and we can use a single case match to get out the original list.

By doing this we have avoided polluting our code with filling in the other cases with assertions or returning errors. Even better, the GHC compilers knows this as well and doesn't tell us there is a branch missing. If in some future we decide to construct another value that also has form `Arr _`, the compiler will realise there is now a branch missing an ask us to fill it in

- this is far better than the usual need to go looking manually for the assertions we would be hitting and filling in the new cases, if we weren't using GADTs.

4.6 The singleton framework

The following idea comes from Richard A. Eisenberg and Stephanie Weirich's paper [5].

At times matching on a GADT won't be quite enough to tell us everything about the included type. Other times a function might be polymorphic only in the return value and we want to specify what the type instance would be. In both of these scenarios, we are trying, at the term level, to inspect the type that a polymorphic function has been instantiated to. While this is not in general possible, given our since we can easily enumerate the types we are working in, there is a way to do so for our `VarType` definition.

A singleton type is a type with only one value. Consider the following declarations:

```
data SBaseTy :: BaseTy -> Type where
  SUnit :: SBaseTy 'UnitTy
  SBool :: SBaseTy 'BoolTy
  SChar :: SBaseTy 'CharTy
  SInt :: SBaseTy 'IntTy

data SVarTy :: VarTy -> Type where
  SBase :: SBaseTy ty -> SVarTy ('Base ty)
  SArr :: SVarTy ty -> SVarTy ('Arr ty)
```

This gives us a one-to-one correspondence between the types in the kind of BaseTy and the constructors of SBaseTy, and similarly for VarTy (We ignore the existence of $\perp$ for now). We can use these in a number of ways:

- When a polymorphic function needs to inspect the 'form' of type, and that type is of kind BaseTy or VarTy, we can add an SBaseTy or SVarTy, with the same type parameter to the parameters of the function (being careful to use the **same** type parameter: the compiler will not warn you if you use a different type parameter, in which case the caller would be free to pass in any singleton value at all and the code would happily compile). The code calling the function would usually be able to deduce what the argument it is passing in 'looks like' (in the sense that they know what type it has and can provide an appropriate singleton) or can pass the requirement up in its own parameters.
- We can now provide our first really form of 'dependent-like' programming. We define the following helpful GADT[4]. Given a kind `ty` and some type constructor `sing` (expected to be a singleton constructor, but cannot be enforced), and a second type constructor `k`, a `TyWrapper sing k` contains a value of type `k a` for some type `a`. While we could 'hide' types inside a GADT in a similar fashion before[5], it is much easier to simply define a singleton type and reuse this.

```
data TyWrapper (sing :: ty -> Type) (k :: ty -> Type) :: Type where
  TyWrap :: sing a -> k a -> TyWrapper mark k
```

[4]While I am sure it has been independently discovered elsewhere, the idea for this specific helper and the generalisation to arbitrary kinds developed over the course of this project.

[5]Consider the constructor `TyWrap :: Typeable a =>a -> TyWrapper`; the Typeable class allows us to recover information about `a` at run-time

4.7 Stack language representation

Now that we have some essential machinery out of the way, we can start to see the full power of our GADTs when translating our expression trees to machine code.

Since machine code is far less structured than expression trees, we are going to want to restrict the types even further to maintain some amount of control over its structure. To this end we parameterise the GADT by the 'shape' of the start state and end states of the stack. For each operation, we assert at the type level what state we expect the top of the stack to be in, and what state it would be left in. We then provide a composition operator `(:.)` that takes a section of stack code whose end state matches the start state of another and composes them in the obvious way. Like the expression GADT, we parameterise it by the available free variables:

Before we can do so, however, we need a bit more machinery at the type level. While polymorphic types in GADTs allow us to essentially 'copy across' type information from on structure to another (for example, consider the way `Location var ty` tells an `Expr var` where to find a value of type `ty`), there isn't much more we can do to manipulate values, apart from wrap them in a given constructor or pattern match on them.

To remedy this, Haskell allows us to define **Type families**, which define relations between types.
Here we take the usual recursive definitions for Append, Map and Repeat and simply lift them to the type level[6].

```
type family Append (list1 :: [k]) (list2 :: [k]) :: [k] where
    Append '[] list2 = list2
    Append (x ': xs) list2 = x ': (Append xs list2)

type family Map (f :: k1 -> k2) (list :: [k1]) :: [k2] where
    Map f '[] = '[]
    Map f (x ': xs) = f x ': Map f xs

type family Repeat (x :: k) (n :: Nat) :: [k] where
     Repeat x 'Zero = '[]
     Repeat x ('Succ n) = x ': Repeat x n
```

These functions become very useful as we make translations between Expressions and Stack code. For example, an array pointer will (in the implementation) be actually be a pointer together with a number of values indicating to us the dimensions of the array. To avoid having to keep track of a list of pointers, expected to be pointing to a contiguous locations, we instead treat this as atomic at the storage level, and create a `CVar` constructor whose translation to machine code will be expected to unpack the 'atomic' data onto the stack.

Before we continue, there is a small caveat in using type families. In order to define a function with polymorphic types, the GHC compiler needs to convinced that given an actual applied type it can deduce the values of the type variables. Consider the following function:

```
exampleFunc :: HList SVar xs -> HList SVar zs ->
               HList SVar (Append xs ys) -> HList SVar (Append zs ys) ->
               Bool
```

In this case, the compiler would like to be able to deduce what the value of `ys` actually is, so

[6]Note that the Singleton library (based on the paper [5]) can do this sort of work for us, as well as define curried versions of these functions (using defunctionalization mechanically).

that it can continue with its type checking and refinement at the call site. However, to do so it needs to know that Append is injective, so that it knows there is exactly one type `ys` for any given signature satisfying the restrictions above. Of course, Append is not in fact injective (in two variables), but given its result and the first parameter we could deduce the second. While GHC does do a small amount of injectivity analysis (via user annotations), it doesn't yet have this functionality. For this reason, quite often we'll need to provide markers or phantom variables to help the compiler along.

Now that we have the above, we can define an auxiliary type family, along with our stack code representation[7].

```
type family Dimension (v :: T.VarTy) :: Nat where
   Dimension ('Base _) = 'Zero
   Dimension ('Arr ty) = 'Succ (Dimension ty)

data StackEntry where
  Ref :: VarTy -> StackEntry
  Val :: BaseTy -> StackEntry

data StackCode :: [T.VarTy] -> [StackEntry] -> [StackEntry] -> Type where
  CNil     :: StackCode vars stack stack
  CSwap    :: StackCode vars (a ': b ': stack) (b ': a ': stack)
  CDup     :: StackCode vars (a ': stack) (a ': a ': stack)
  (:.)     :: StackCode vars a b -> StackCode vars b c -> StackCode vars a c

  CPlus    :: StackCode vars ('Val 'IntTy ': 'Val 'IntTy ': stack)
                             ('Val 'IntTy ': stack)
  CNeg     :: StackCode vars ('Val 'IntTy ': stack)
                             ('Val 'IntTy ': ts)
  CEq      :: StackCode vars ('Val t ': 'Val t ': stack)
                             ('Val 'BoolTy ': stack)
  CNot     :: StackCode vars ('Val 'BoolTy ': stack)
                             ('Val 'BoolTy ': stack)
  CCompGT  :: StackCode vars ('Val 'IntTy ': 'Val 'IntTy ': stack)
                             ('Val 'BoolTy ': stack)

  CLit     :: E.Literal ('Base ty) ->
              StackCode vars stack ('Val ty ': stack)

  CIf      :: StackCode vars stack1 stack2 ->
              StackCode vars stack1 stack2 ->
              StackCode vars ('Val 'BoolTy ': stack1) stack2
  CWhile :: StackCode vars stack ('Val 'T.BoolTy ': stack) ->
            StackCode vars stack stack ->
            StackCode vars stack stack

  CVar     :: T.ListPointer vars ty ->
              StackCode vars stack ('Ref ty ':
                                    Append (Repeat ('Val 'IntTy)
                                                   (Dimension ty))
                                           stack)

  CLookup  :: StackCode vars ('Ref ('Base ty)) ': stack)
                                     ('Val ty ': stack)
```

[7]The idea for this representation comes from [6], where Conor Thomas McBride demonstrates how to use lists in datatype parameters to enforce stack code restrictions. I extended this with the De Bruijn indices to account for the fact that our machine code requires more than just stack operations.

```
  CStore   :: StackCode vars ('Ref ('Base ty)) ': 'Val ty ': stack) stack
```

A few of these have a few oddities that we will try to lay out. The first set are essentially straightforward: `CDup` takes a stack with at least one element, and returns a stack which has two elements of the same type at the top of the stack[8]. `CIf` and `CWhile` aren't quite stack sequences: they still keep their branches separate, since we don't yet have the concept of a label or jumping. `CIf` takes two sets of stack code (with the same behaviour, i.e. that start and end in the same kind of stack states) and returns a stack code that takes the same stack start type, plus a Boolean, and returns the right stack type. `CWhile` acts similarly; we enforce the additional restriction that each loop must leave the stack in the same form that it started in, and that the test simply places an additional Boolean value on the stack.

Now that we have set up most of our machinery, the actual translation from Expressions to stack code is fairly mechanical. For each definition, the compiler is checking behind the scenes that we do not underflow the stack (since no constructor can 'remove' anything from an empty stack), and that the code we 'glue' together using `(:.)` has matching types. Thanks to type unification by GHC and that we have left the rest of the stack polymorphic, GHC can automatically infer the start and end states of joining two sets of code even if they have different requirements on the stack, as long as these are compatible.

Looking at the type signature of `compileExpr`, we have essentially given ourselves a recursive call invariant and enforced it at the type level: compiling an Expr returns code that does nothing to the stack but add another value on top. The invariant on compileLocation is even more useful: given a pointer to an n-dimensional array we leave on the stack a pointer to the start of the array and a list of the integers of length equal to the dimension of the array (corresponding to the sizes of each subarrayi level). Since Expression variables are restricted to pointers to base types, which have dimension zero, GHC can infer that the result on the stack is simply the reference with no additional integers. This is exactly the kind of condition we would write in a Hoare triple: we set up the precondition (CVar pointer places the values on the stack), we run the series of statements, preserving the invariant, and when it exits the combination of the invariant and the exit condition give us the property we were looking for.

```
compileExpr :: Expr stores calls ty ->
               StackCode stores calls stack ('Val ty ': stack)
compileExpr (E.Add e1 e2)          = compileExpr e1
                                     :. compileExpr e2
                                     :. CPlus
compileExpr (E.Mul e1 e2)          = compileExpr e1
                                     :. compileExpr e2
                                     :. CMul
compileExpr (E.Neg e)              = compileExpr e
                                     :. CNeg
compileExpr (E.And [])             = CLit $ E.BoolLit True
compileExpr (E.And [e])            = compileExpr e
compileExpr (E.And (e : e2 : es))  = compileExpr e
                                     :. CIf (CLit $ E.BoolLit True)
                                            (compileExpr $ E.And (e2 : es))
compileExpr (E.Not e)              = compileExpr e
                                     :. CNot
compileExpr (E.CompareEQ e1 e2)    = compileExpr e1
```

[8]Note that we are careful not to say that `CDup` actually duplicates the value. A bad implementation may decide to do strange things like increment one of the values if the value is an integer. However as this is polymorphic in the top type of the stack, we can deduce that the only sensible implementation is to duplicate the top value. Philip Wadler's Theorems for Free [7] explores this idea in detail.

```
                                    :. compileExpr e2
                                    :. CEq
compileExpr (E.CompareGT e1 e2)   = compileExpr e1
                                    :. compileExpr e2
                                    :. CCompGT
compileExpr (E.CompareGE e1 e2)   = compileExpr (E.Not $ E.CompareGT e2 e1)
compileExpr (E.Literal a)         = CLit a
compileExpr (E.Variable location) = compileVarLocation location :. CLookup
compileExpr (E.If eb e1 e2)       = compileExpr eb
                                    :. CIf (compileExpr e1) (compileExpr e2)


compileLocation :: E.Location vars ty ->
                     StackCode vars stack ('Ref ty ':
                                           Append (Repeat ('Val 'IntTy)
                                                          (Dimension ty))
                                                  stack)
compileLocation (E.Var pointer) = CVar pointer
compileLocation (E.Index loc index) = compileLocation loc
                                    :. CSwap
                                    :. compileExpr index
                                    :. CMul
                                    :. CIndexV
```

4.8 Proving type equality

When writing code involving recursive type families, it is beneficial to have recursive calls be on the same parameter as the recursion in the definition of the type family. However, this is not always possible, and in such cases the GHC compiler may not be able to infer that the value you are giving it, of a certain type, matches the type of the value it expects. This came up twice throughout this project:

- At one point we needed to recurse in the second argument of a structure that used `Append` in its type. At the base case, GHC cannot infer that `Append xs '[] == xs`.
- When dealing with channels, we wanted the concept of 'casting' a channel type to a read or write type. Channels are given types `ChanType = ChanArray ChanType | ChanInput BaseType | ChanOutput BaseType | ChanBoth BaseType`, and clearly an instance of `'ChanBoth ty` can be used wherever a `'ChanInput ty` is required. However, GHC can't infer that the dimension of a, say an array of arrays of `'ChanInput` is the same as a `cast` of an arrays of arrays of `'ChanBoth`.

A way to solve this is given by Richard Eisenberg in [8]. We begin by defining a pair of datatypes

```
data PropEq :: k -> k -> * where
  Refl :: PropEq x x

type EnumerableEquality (a :: k) (b :: k) = Maybe (PropEq a b)
```

Here, a value `Refl` of type `PropEq T1 T2` is an assertion that the types `T1` and `T2` are equal. We can combine this with our singletons framework from above to either generate proofs of equality, or to, or to test equality and return a value of `Maybe (PropEq a b)`. For example, the following is a proof that `Append xs '[] == xs`:

```
proveAppend :: HList k ts -> PropEq (Append ts '[]) ts
proveAppend HNil = Refl
proveAppend (HCons _ rest) = case proveAppend rest of Refl -> Refl
```

This is a simple inductive proof, encoded in the type system. Note that in order to construct the `Refl` value at runtime, the code will run scan through the `HList` every time you use `ProveAppend`. In a fully dependently typed langauge, termination would be checked at compile time so that type-checking would be sufficient to convince the compiler that such a value does indeed exist and that it can elide the actual running of `ProveAppend`. Haskell, however, allows functions to not terminate, so that for example the following is well-typed:

```
proveRidiculous :: PropEq Int Char
proveRidiculous = proveRidiculous
```

4.9 Extending to Functions and Processes

Now that we have quite a bit of machinery available to us, we can look at extending the translation from our Expression GADT into a translation from all our Process an Function GADTs. While this ends up being a significant amount of work, most of which involves implicitly proving to the compiler that we've satisfied the type requirements that we have set by setting up helper functions with the right conditions, academically it is not much more complex than what this report has already covered.

One aspect we do want to cover is how we manage the implicit Worker state that was touched on in section 3.4.1. We note that there are certain operation codes that must be performed in a specific order: in particular, `TimeBranch` or `AltBranch` command may only appear after a `startAlts` and before an `EndAlts` command. In order to ensure that we do follow this convention, we tag the `StackCode` construction with an additional pair of type parameters, indicating which 'implicit state' the machine code starts and ends at:

```
data WorkerState = Ready | Alt [StackEntry] | Par | Done

data StackCode :: [VarType] ->
                  [StackEntry] -> [StackEntry] ->
                  WorkerState -> WorkerState ->
                  Type where
  CNil     :: StackCode stores calls tys tys state state
  CSwap    :: StackCode stores calls (a ': b ': rest) (b ': a ': rest) state state
  CDup     :: StackCode stores calls (a ': rest) (a ': a ': rest) state state
  ...

  CAltBranch :: SCast ch ('ChanInput ty) ->
                SNat n ->
                StackCode stores calls (Append (Repeat ('Val 'IntTy) n)
                                               ('Val ty ': tys))
                                       tys2
                                       'Ready 'Ready ->
                StackCode stores calls (Append (Repeat ('Val 'IntTy) n)
                                       ('Ref ('Chan ch) ': tys))
                                       tys
                                       ('Alt tys2) ('Alt tys2)

  ...
```

This begins to demonstrate the difficulty in making absolutely everything explicit: the types start to become very unwieldy, and small changes to the interface require significant changes throughout the module. The `CAltBranch` constructor takes a possible branch, that starts with n integers on the stack. It then becomes a `StackCode` that pops n values off the stack, as well as a channel reference, and stays in type `Alt tys2`. The reason we parameterise the `Alt` constructor is that each of the branches must leave the stack in the same form: this form is given by the parameter of `'Alt`.

At this point we have two possible approaches to actually changing the final pair of parameters so we can actually can use these constructors. In the `Alt` case, we simply provide a constructor `CAlts` which takes as a value the whole sequence generating the `Alt` constructors, expecting the later conversion to know to insert the right tokens on either side, while in the (simpler) `CPar` case, we simply create starting and ending contructors that change the state accordingly.

```
CAlts :: StackCode vars tys tys ('Alt tys2) ('Alt tys2) ->
         StackCode vars tys tys2 'Ready 'Ready

CStartPars :: StackCode vars tys tys 'Ready 'Par
CEndPars :: StackCode vars tys tys 'Par 'Ready
```

In order to eventually describe all kinds of variable, functions and processes in scope, the GADT definition ends up getting a bit more complicated:

```
data BaseTy = UnitTy | BoolTy | CharTy | IntTy
data ChanTy = ChanInput BaseTy | ChanOutput BaseTy | ChanBoth BaseTy | ChanArr ChanTy
data VarTy = Base BaseTy | Arr VarTy

data StoreTy = Chan ChanTy | Var VarTy | Ref BaseTy

data FuncTy = FuncTy [BaseTy] [BaseTy] -- parameterised by inputs and outputs

data ProcArg = ProcConst BaseTy | ProcChan ChanTy | ProcVar VarTy
data ProcTy = ProcTy [ProcArg] -- parameterised simply by arguments

data CallTy = Func FuncTy | Proc ProcTy

data StackCode :: [StoreTy] -> [CallTy] ->
                  [StackEntry] -> [StackEntry] ->
                  WorkerState -> WorkerState ->
                  Type where
  ...
```

4.10 Exporting our StackCode

4.10.1 The state monad

While our `StackCode` representation is already largely quite flat, there are still a few branches left in some of the constructors. These are precisely the points where labels are needed: since we didn't yet have access to such labels it was impossible to flatten the structure any further. In order to do so now, while avoiding label clashes, we will work inside the State monad to give us a fresh supply of labels.

We first define a simple datatype to represent our operation codes. Since we will be changing representations of our labels, from markers to fixed positions, we can make our lives easier

by parameterising the Opcode structure by the type of the label then deriving a Functor instance so we can map over the structure with little boilerplate.

```
{-# LANGUAGE DeriveFunctor    #-}
data Op a = NOP | SWAP | DUP | INC | DEC | ADD | MUL | DIV | MOD
          | NEG | EQ | AND | OR | NOT | GT | Lit(Int)
          | LDW | STW | LOCAL(Int) | FP
          ...
          | Jump | CJump | LabelLit(a)
          | KEY | SCR | TIME deriving (Show, Functor)
```

We can then do our conversion in the state monad. We define a function that gives us fresh variables within the monad, and then request a new label from it when we need one:

```
import qualified Control.Monad.State.Lazy as ST

type Label = String

freshLabel :: ST.State Int Label
freshLabel = ST.modify (+1) >> ST.get >>= (return . show)

-- An Operation code with any number of labels
data TaggedOp a = T [Label] (Op a)
```

In order to deal with frame declarations and function calls, we can augment the signature for `convert` to take a sequence of representations of frames, and a vector indicating what label each function or procedure has. The function/procedure vector is an easy lookup. For the frames, we need to walk up the frame list, counting down the list pointer until we know how many frames we have to follow and what position in the frame we need to look at.

```
data Frame :: [T.StoreTy] -> Type where
  NilFrame :: Frame '[]
  ConsFrame :: T.HList T.SStoreTy tys1 -> Frame tys2 -> Frame (T.Append tys1 tys2)

convert :: Frame stores ->
           C.Vect (Length calls) Label ->
           U.StackCode stores calls a b st1 st2 ->
           ST.State Label [TaggedOp Label]
convert c _ (U.CVar pointer) = return $ varInfo c pointer

varInfo :: Frame tys -> T.ListPointer tys ty -> [TaggedOp Label]
varInfo f loc = [T [] FP] ++ (helper f loc)
                where
                  helper :: Frame tys -> T.ListPointer tys ty -> [Tagged]
                  helper (ConsFrame (T.HCons ty rest) _) T.Last =
                    -- we've found the frame, calculate what position of the frame
                    -- we're at
                    let pos = frame_pos rest
                        n = dimen ty
                    in simples [Lit (pos + n), ADD]
                       ++ (concat $ take n $ repeat (simples [DUP, LDW, SWAP, DEC]))
                       ++ (simple LDW)
                    where
                      frame_pos :: T.HList T.SStoreTy tys1 -> Int
                      frame_pos = ..
                  helper (ConsFrame (T.HCons _ rest) next_frame) (T.Prev p) =
```

```
        -- count down the contents of the frame until it is empty
        helper (ConsFrame rest next_frame) p
    helper (ConsFrame T.HNil prev_frame) pos =
        -- we've run out of frame - load the next one and carry on
            [T [] LDW]
             ++ helper prev_frame pos
```

4.10.2 Tying the knot

Now that we have a sequence of operations tagged with labels, it is time to fill in the the jumps! While usually this would take two passes over the data (one to collect up the label positions and another to write then to the appropriate places), we can in fact take advantage of Haskell's laziness and do this in a single pass[9]. The idea is simple: we write a function that takes a lazy map, and which builds up a new map as it walks over the sequence. As we walk over the list, we add `(label -> position)` pairs that we come across to our new map. We also, at each step, use our Functor instance to lookup values in `LabelLit` in the map we received as a parameter.

Finally, we use a bit of Haskell wizardry, applying a technique called **tying the knot**[9]: take our function, and feed its output map back into its input. Thanks the the power of laziness, it turns out that when we used fmap, the lookup wasn't evaluated until it is needed - which is only at the end when we later force the computation to output the code. This means that the map doesn't need to have been built until after we've run through the whole list - at which point we have enough information for it to be built!

```
import qualified Data.Map.Lazy as Map

compute_labels :: [TaggedOp Label] -> [TaggedOp Int]
compute_labels ops = let (done_map, result) = looper done_map -- magic
                     in result

looper :: Map.Map Label Int -> [TaggedOp Label] -> (Map.Map Label Int, [TaggedOp Int])
looper done_map [] = helper 0
                     where
                       helper :: Int -> [TaggedOp Label] ->
                                 (Map.Map Label Int, [TaggedOp Int])
                       helper _ [] = (Map.Empty, [])
                       helper n (T labels op : rest) =
                          let (new_map, rest_mapped) = helper (n+1) rest
                              mapped_new_map = foldr Map.insert new_map labels
                              mapped_op = fmap ((Map.!!) done_map) op
                          in (mapped_new_map, mapped_op : rest_mapped)
```

[9] Credit is due to a paper that shows how to replace each value in an integer tree with the minimum value of that tree in a single pass; unfortunately after much searching I been unable to find it again.

Chapter 5

Conclusions

5.1 Evaluation and testing

In terms of requirements carried out, the project ended up growing far larger than I expected, and programming it left minimal time for extensive testing. I tested a range of programs on the compiler and machine which worked as expected - these can be found in the program directory [1].

I would argue that writing the project was itself a test of the methodologies described in this report. The aim of of the project was to explore techniques that build trust in the compiler. Time and again over the course of programming the compiler, I was sure absolutely sure that I had translated something correctly, only to realise after spending an extended period of time fighting the compiler that one of my assumptions was wrong or that I had mis-coded an interface in a different part of the file. To this extent, while the barriers meant that every task took (on average) around three or four times longer than I initially expected, and a lot more code, this is what you come to expect when you have to prove that your code works, in a slightly more formal way than usual. In as far as I *trust* my code, I have much more faith in the parts of the compiler that are tightly bound by the restrictions I have placed through the type system than in equivalent code where I would have spent a lot more time testing instead.

That's not to say, of course, that testing wouldn't have had its place. I would have benefited immensely from forcing myself to write a framework to test the code end-to-end, if only to look through the code and realise what parts I was missing and how I could circumvent them. 'Testing-by-compiling' meant I often used placeholder error functions for functions I hadn't yet implemented, and I would have benefited from some better methodology in keeping track of what sections I had and had not implemented. Of course, I also cannot extend the same trust that I have in the core of the compiler out the the edges: the parser, the translator from the Raw AST to the parameterised GADT, and the function that turns the `StackCode` into binary all have far fewer restrictions. If I wanted to put the same amount of emphasis on these, I'd likely have to drop Haskell altogether and use a full-blown dependently-typed language.

While a lot can be said about how the methodology meant that I felt I couldn't test the code part-way until it was finished, that mistake of course lies on my shoulders as well. In that way this has been a great learning experience: clearly the design process should have had more of a focus on incremental design and setting up achievable milestones.

5.2 Conclusions and further research

In conclusion, I felt the project was an overall success. The aim was to explore how we can use techniques from dependently-typed languages in a Haskell program, and in this sense I certainly got a feel for how such programming needs to be structured to satisfy the compiler's requirements. In addition I managed to implement all the compiler features I was aiming for: replicated arbitrary-sized indexing, nested `ALT` constructs, mutually recursive functions. In practice the additional safety meant writing a lot of extra code that lays out explicitly why the assumptions made are correct. While a stronger type system in theory means the compiler can make more optimisations, this might be offset by the large amounts of extra code produced to satisfy the compiler.

There is a number possible directions to take carrying on from this project. Currently it feels quite unergonomic to program with dependent types (or more precisely dependent-like types) in Haskell. As Haskell evolves and pulls in more features from that world, it would be useful to see how the ergonomics of this progresses. A tangentially related aspect of this is the need to constantly define new datatypes and associated helper functions to describe the requirement that you want on the structure. Going from freely using lists, folds, maps and the like to resorting to defining 'helper' functions all the time to walk up and down structures feels like a step in the wrong direction; perhaps as more people use these sorts of structures we might develop a good set of generalisations that can be applied in a variety of situation. This already happened here when necessity forced me to generalise the HList to parameterise it by some constructor, then realising it worked quite nicely in different places; much like lists and list functions evolved in functional programming, the same might happen for related operations in the dependent world.

Bibliography

[1] Jaime Valdemoros Gomez, *Compiler and virtual machine code listings* https://github.com/JaimeValdemoros/4th-year-project

[2] Pountain, Dick, and David May. *A tutorial introduction to OCCAM programming* McGraw-Hill, Inc., 1987.

[3] Hutton, Graham, and Erik Meijer, *Monadic parsing in Haskell*, Journal of functional programming 8.04 (1998): 437-444.

[4] Crockford, Douglas. *Top down operator precedence* Beautiful Code: Leading Programmers Explain How They Think (2007): 129-145.

[5] Eisenberg, Richard A., and Stephanie Weirich. *Dependently typed programming with singletons.* ACM SIGPLAN Notices 47.12 (2013): 117-130.

[6] Conor Thomas McBride. *Agda-curious?: an exploration of programming with dependent types.* Proceedings of the 17th ACM SIGPLAN international conference on Functional programming (ICFP '12).

[7] Wadler, Philip. *Theorems for free!* Proceedings of the fourth international conference on Functional programming languages and computer architecture. ACM, 1989.

[8] Eisenberg, Richard A. *Dependent types in Haskell: Theory and practice.* arXiv preprint arXiv:1610.07978 (2016).

[9] Haskell wikipedia page *Tying the knot* "https://wiki.haskell.org/Tying_the_Knot"