Compilation of a CSP-like Language

Candidate Number: 1024701
Word Count: 9623

Fourth Year Project Report
Honour School of Computer Science - Part C
Trinity 2021

Abstract

We create a small, portable implementation of a subset of Occam, a parallel programming language based on CSP, that concentrates on the process algebra parts of the language. We accomplish this by implementing a virtual machine based on the Inmos transputer and a full-stack compiler that targets its instruction set. We discuss the different methods of memory allocation, and analyse the possible ways to introduce recursion to our language.

Acknowledgements

I am deeply indebted to my supervisor for his guidance throughout the project and expert advice about design decisions. I would also like to thank my friends and family for supporting me throughout my work for this project, especially in times as trying as these.

Contents

1 Introduction

Parallel processing computer systems underwent a revolution in the 1980s, with the United Kingdom at the forefront. During this period, as traditional CPU designs were reaching a performance plateau, researchers and industry alike began to look towards parallel execution as the next great milestone in computing.

One significant step in this direction would be undertaken by Inmos, a British semiconductor firm based in Bristol. In 1983, employee David May, together with consultant Robert Milne, issued the first design of the *transputer*, a series of microprocessors intended for parallel computing. The transputer heralded multiprocessing, the separation of independent software processes onto physically independent computing units, as the natural evolution of multitasking. With their simple yet bold architecture that facilitated concurrent programming, the series initially showed great promise and led to a metaphorical revolution in the hardware industry.

That same year, May and Richard Taylor, under the supervision of C.A.R. Hoare, would develop the parallel programming language *occam* to serve as the native language for the transputer. As a middle point between a programming language and a design formalism, it provided programmers with simultaneously powerful and rigorous ways to define the behaviour of concurrent programs. occam was well suited for its task, being used both to describe the structure of a system in terms of connected transputers and to program the individual transputers themselves.

In 1989, Chris Jesshope [1] identified the transputer revolution as the convergence of two extremely active (at the time) streams of research that had their roots in the 1960s. The first of these was the advent of the microprocessor. The second was the development of parallel computer architectures and methods by which they can be effectively used in application development. On the theoretical side, groundwork had also been laid by C.A.R. Hoare with the introduction of the process algebra CSP (communicating sequential processes) in 1978, used for rigorously describing patterns of interaction in concurrent systems.

Ultimately, software development lagged behind the rapid developments in hardware technology, and advances in traditional computing meant that by the early 1990s Inmos had halted the production of transputers. While the transputer did not achieve the revolutionary success that Inmos had hoped for, it did provoke new ideas in computer architecture, several of which have re-emerged in different forms in modern systems. In particular, the model of cooperating concurrent processors which it pioneered lives on in the world of supercomputing.

1.1 Motivation

Owing to its close ties with the formal process algebra CSP, occam provides programmers with very natural and powerful ways to describe the interaction of concurrent processes. This not only makes it a great programming language to design such systems, but also a fantastic didactic tool. Unfortunately, the baggage of the original context that it was designed in makes it less suited for teaching purposes. For one, it is meant to compile onto a transputer, but transputers have only seen hobbyist use (and that occasionally) since Inmos stopped producing them in the 1990s. Secondly, while its didactic value is given by the fundamental framework for concurrent processes that it provides, occam contains many features that are superfluous to this.

In addition to this, the choice to perform the allocation of activation records statically means that no official language specification has ever supported recursion, although there have been extensions (notably occam-pi) which do. Recursion is orthogonal to the rest of the language, and enabling it broadens the range of programs that can be expressed in the language. Additionally, it is a good exercise in language design.

This project aims to remedy some of the drawbacks described above and to provide a modular, thorough and well-made educational tool that spans the entire language, from lexical analysis to running machine code. It achieves this by isolating the core of occam - the implementation of the process algebra CSP through the different language constructs - and decoupling it from old architecture by emulating the transputer.

1.2 Contributions

We contribute the following:

- An informal specification of a simplified version of occam that concentrates on the process algebra parts of the language.
- A virtual machine based on the Inmos transputer.
- A compiler and assembler for our language that target this virtual machine.
- An analysis of the types of memory allocation, the way they are used in occam, the constraints that recursion imposes upon the types of allocation that may be used, and some possible ways to implement it.

1.3 Outline

Section 2 goes over knowledge that is fundamental to understanding the context of the project. In Section 3 we describe in detail the virtual machine that we have created, the intermediate representations of code, as well as how the bytecode is assembled. Section 4 goes over the lexical analysis, parsing, and code generation for our simplified version of occam. In Section 5, we explore a possible implementation of procedures for our language, and how we might go about enabling recursion for these procedures. We test our full-stack solution in Section 6, and conclude with Section 7. Extra information that is not strictly necessary to understand the project is presented in the appendices, in Section 8.

2 Background

2.1 Transputer

As their name, a portmanteau of *transistor* and *computer*, suggests, transputers were meant to constitute building blocks for larger systems by connecting to each other via their four links to form a *farm*. Within these distributed systems, each transputer provided services, like input or output, to the greater whole. Each chip could hold a fixed number of processes and came with its own integrated scheduler, thus removing the need for process scheduling at the high level of the system.

Transputers share a common instruction set designed for simple and efficient compilation. Instruction sizes are fixed at one byte long, and each instruction is divided into two equal parts. The most significant four bits are a function code, while the least significant four bits are a data value. This representation provides for sixteen functions, each with data values ranging from 0 to 15. Two of the functions, *pfix* and *nfix*, are used to incrementally construct operands of up to the word size of the processor. Thirteen of the functions are used to encode the most important operations (in Inmos' judgment) performed by a compiler executing a high-level language. The final one, *opr*, causes its operand to be interpreted as the code of an operation to be executed. Operations are instructions that have no direct arguments, but instead push and pop values from the evaluation stack. Operations allow for an essentially unbounded number of instructions to be encoded, but with the caveat that higher codes are costlier to load.

The classical transputer has six main registers - three *evaluation stack* registers **A**, **B** and **C**, an operand register, a workspace pointer and an instruction pointer. Additionally, they contain additional registers for specialist purposes like process scheduling and counting ticks. Pushing a word onto the evaluation stack causes the values of **A** and **B** to move down, and likewise popping one causes the values of **B** and **C** to move up.

2.2 Occam

Building on Hoare's earlier CSP process algebra, occam was intended (May & Taylor [2]) as a middle point between a programming language and a design formalism. As such, an occam program may be read as either a sequence of instructions for a machine to execute, or a predicate in an extension of predicate calculus about which mathematical deductions can be made. Through this double nature, occam lends itself particularly well to automated verification and enables programmers to take a process-calculus approach to solving problems.

Since it first appeared in 1983, occam has undergone several revisions. The last official version, occam 2.1, was released in 1994. Version 3 was planned, and a reference manual was distributed for community comment, but it was ultimately never fully implemented in a compiler. Parallel to this, the University of Kent has been developing a competing standard, occam-pi, since 1996. This variant of the language is improved through the addition of several ideas from pi-calculus, and is implemented by the Kent Retargetable Occam Compiler (KRoC).

An occam program is a collection of concurrently executing *processes* that act upon *variables* and communicate via *channels*. Each process inside an application describes the behaviour of a particular aspect of the implementation, and each channel describes a connection between two processes [3].

The simplest occam processes are *primitive* processes, which provide some fundamental, indivisible service to the program. We recognise five types - SKIP, STOP, assignment, input and output. Semantically, SKIP does nothing, while STOP loops forever. An assignment attributes the value of an expression to a variable, and input and output communicate values between processes through channels.

Alternatively, processes may be composed from other processes through the IF, CASE, WHILE, SEQ, PAR and ALT constructs. We do not implement CASE since it can be simulated through the IF construct. IF and WHILE behave identically to their counterparts in sequential languages. SEQ takes a list of processes and runs them sequentially. PAR also takes a list of processes, but runs them in parallel. Finally, ALT gates a list of alternatives behind boolean expressions that are evaluated at run-time and channel inputs. It waits for one of the alternatives to become ready (when its boolean evaluates to true and its channel is ready for communication), then selects and runs this alternative.

While this is all the background knowledge required to understand this project, the official specification of occam is much lengthier. The full specification for version 2.1 can be found in the reference manual [3], including code examples.

2.3 Procedures & Types of Allocation

As a general rule (Scott [4]), memory in programs may either be allocated *statically* at compile-time or *dynamically* at run-time. At the level of individual objects (which conceptually can include activation records and program text, as well as objects defined inside the programming language), statically allocated objects are given an absolute address at compile-time that is retained throughout the program's execution. On the other hand, dynamically allocated objects are given a new, separate address each time they are instantiated. The main advantage of static allocation is efficiency, since it is performed at compile-time and requires no additional overhead. Dynamic allocation is relatively expensive, with costs ranging from stack manipulation to system calls.

At the program level, Vasseur & Dunkels [5] identify that static allocation of the entire program allows for the memory footprint to be known at compile-time. A program that makes use of dynamic allocation, however, trades this predictive power for the ability to respond to the actual memory load that the system requires. One particular kind of dynamic allocation is *stack allocation*, where objects are allocated and deallocated in a last-in, first-out order. Stack allocation is usually used for subroutine calls and returns and naturally supports recursion. Other distinguished kinds of dynamic allocation are *heap allocation* and the hybrid *dynamic allocation from static memory pools*.

occam implements three instances of the procedure abstraction - the process, the procedure, and the function. Broadly, a procedure is a named process, while a function is a procedure that can return a value. Cooper & Torczon [6] explain that calling a procedure creates a distinct instance, or *activation*, of the procedure. Each instance is associated with control information (the caller, the return address) and data (local variables, passed arguments, additional temporaries). The compiler defines a data structure called an *activation record*, collecting this information into private blocks of contiguous storage. In principle, each activation record is associated with a single instance of a single procedure, and every procedure call generates its own activation record.

It is the choice of each *language specification* whether to store activation records statically or dynamically (if they do, it is generally on the stack, where they become known as *stack frames*). Scott [4] notes that if a language specification permits recursion, then static allocation of local variables is no longer an option, since the number of instances of a variable that may need to exist at the same time is both unknown at compile-time and conceptually unbounded. We can extend this notion to static allocation of activation records, since a recursive function by definition has at least one parameter, the total number of individual arguments is unbounded across a recursive execution, and each argument must be stored either in the parent method's activation record or the child's.

Inmos decided (Hull [7] & Zarea-Aliabadi [8]) to enforce static memory allocation for occam progams in the interest of minimising execution time and defending against running out of memory space. In particular, they sought to minimise the run-time overhead of the parallel construct [9]. As a trade-off, explicit recursion was forbidden, and there has not been any official version of occam that supports it. As further restrictions, the number of elements of an array must be known at compile time, and so must the number of processes created by a parallel replicator [9].

3 Assembly & Interpreting

3.1 Interpreter

We implement a virtual machine based on the Inmos transputer. For the most part, our design will closely align with the official specification of the instruction set [10]. Our interpreter uses the same registers as a transputer, features separate process and instruction memories and has a native word-length of 64 bits. It also contains a scheduling mechanism and automatically time-slices processes on jumps. Throughout this section, we use the terms *virtual machine* and *interpreter* interchangeably. We denote the contents of memory at address i by $M[i]$. We also identify the name of a register with its contents.

3.1.1 Registers

The virtual machine has six main registers named **A, B, C, I, W** and **O**, as well as four special registers **qHead**, **qTail**, **tickCount** and **elapsedCount**.

Registers **A**, **B**, and **C** form the *evaluation stack* of the interpreter. In that sense, instructions may load words into **A**, which pushes the previous value of **A** into **B** and the previous value of **B** into **C**. They may also consume up to three values from the stack, resulting in the value of **A** becoming the previous value of **B** and the value of **B** becoming the previous value of **C**. Instructions may do both, in which case they pop some values off the stack, and push a value back on. Register **I** holds a pointer to the current instruction, and **W** holds a pointer to the workspace of the currently running process. The register **O** is used by the *pfix* and *nfix* instructions to construct larger operands.

3.1.2 Instruction Set

Instructions have a fixed length of one byte, meaning that the instruction set is completely independent from the word-length of the virtual machine. This has a historical basis, in that Inmos produced transputers with varying word-lengths, and it was intended to be easy to retarget code to any other chip in the series. The four most significant bits are a function code, and the four least significant bits are a value. This allows for sixteen different functions, each with sixteen different possible values. We present the full instruction set in the table below. Almost every time, after the code and value nibbles of an instruction have been separated, the contents of **O** are shifted up four places and or-ed with the instruction value. We call the value of **O** after this has taken place the *operand* of the instruction.

Function Code	Abbreviation	Description
0x0	pfix	Prefix
0x1	nfix	Negative prefix
0x2	opr	Operate
0x3	ldl	Load local
0x4	stl	Store local
0x5	ldlp	Load local pointer
0x6	ldc	Load constant
0x7	ldpi	Load pointer to instruction
0x8	eqc	Equal to constant
0x9	j	Jump
0xA	cj	Conditional jump
0xB	ldnl	Load not local
0xC	stnl	Store not local
0xD	ldnlp	Load not local pointer
0xE	call	Call
0xF	ajw	Adjust workspace

Informally, *ldlp* pushes the address **W** + **A** onto the evaluation stack. *ldl* and *stl* push (and respectively pop) onto the evaluation stack the contents of the memory at the address calculated by *ldlp*. Conversely, *ldnlp* increments **A** by **O**, while *ldnl* and *stnl* manipulate the memory at the address created by *ldnlp*. *ldc* loads **O** into **A**. *eqc* compares **A** with *O*, pushing the result onto the evaluation stack. *ldpi* adds the **I** to **A**. *j* is the usual, and *cj* only executes if **A** is 0. The *call* and *ajw* instructions are used to implement procedures, but are not implemented for the purposes of this project.

Operations are other processor functionalities that are called through the *opr* instruction. When such an instruction is executed, its operand is taken to mean the code of the operation to be performed. There is enough space in the value nibbble of *opr* to encode the most common sixteen operations in a single instruction, although higher codes need to be built through *pfix* and thus cost more instructions to encode. Operations pop a numer of values off the evaluation stack, do some work, then optionally push a value back. In effect, the operation mechanism allows for an unlimited number of functions, with the caveat that the higher the code of a function, the more time it takes to perform it. We present the full operation set, numbering nineteen operations, in the table below.

Code	Abbreviation	Description
0x00	noop	No operation
0x01	rev	Reverse
0x02	gt	Greater than
0x03	add	Addition
0x04	sub	Subtraction
0x05	mul	Multiplication
0x06	div	Division
0x07	rem	Remainder
0x08	startp	Start process
0x09	endp	End process
0x0A	inword	Input word from channel
0x0B	outword	Output word to channel
0x0C	enbs	Enable guard w. SKIP
0x0D	enbc	Enable guard w. channel
0x0E	diss	Disable guard w. SKIP.
0x0F	disc	Disable guard w. channel.
0x10	alt	Start ALT
0x11	altwt	Wait for an alternative to become ready
0x12	altend	End ALT

The full formal specification can be found in Appendix A. *noop* does nothing, but we are required to have it since the interplay between prefix generation and offsets in the assembly file means that on occasion, we will be required to add in extra instructions that achieve nothing. Since its purpose is padding out groups of instructions, the size of its operand must also be encodeable in a single byte. *rev* swaps **A** and **B**, and is used heavily in expression evaluation. *gt*, *add*, *sub*, *mul*, *div* and *rem* all pop **A** and **B** off the stack and push a new value corresponding to their output.

3.2 Scheduler

The interpreter holds a queue of processes that are ready to run. Each running process has its own *workspace*, a contiguous region of memory where the process holds its local variables, temporaries, along with four special locations for control information. The structure of a workspace is as follows - offset 0 ($M[W]$) is the static link. Offsets $M[W + i]$ for $i > 0$ are used to store local variables and temporaries, and offsets $M[W - i]$ for $i > 0$ are used to store control information. The special workspace locations are described in Table 1.

We store the workspace of the next scheduled process at $M[W-4]$ as a natural way of implementing the process queue in main memory. When a process is descheduled, it stores its current **I** at $M[W-3]$ so that it knows where to continue when it is rescheduled. We use two special registers, **qHead**

W - 1	ALT State
W - 2	Channel byte | Alt selection
W - 3	Saved I
W - 4	Workspace of successor processs in the queue

Table 1: Special Workspace Locations

and **qTail**, to store the workspaces of the first, and respectively last, processes in the queue. In the case of an empty queue, these registers hold the value *Scheduler::kNone*, corresponding to the highest integer that is representable on 64 bits. The special registers **tickCount** and **elapesdTicks** measure the total tick count from the last reset, and respectively the number of ticks that the current process has been running for.

```
void scheduleProc(Word Q) {
    /** Append to the process queue. */
    M[qTail - 3] = Q;
    qTail = Q;
    M[Q - 3] = Sched::kNone;
}

void descheduleCurrent() {
    /** Save I in the current workspace. */
    M[W - 4] = I;
    if (qHead != Sched::kNone) {
        /** Is there another process to schedule? */
        W = qHead;
        I = M[W - 4];
        if (qHead != qTail) {
            /** There are still other processes to schedule. */
            qHead = M[W - 3];
        } else {
            /** This was the last one. */
            qHead = qTail = Sched::kNone;
        }
    } else {
        /** Stop. */
        W = Sched::kNone;
    }
    elapsedTicks = 0;
}
```

```
void rescheduleCurrent() {
    if (qHead != Sched::kNone) {
        Word Q = W;
        descheduleCurrent();
        scheduleProc(Q);
    }
    elapsedTicks = 0;
}
```

In order to enforce time-sharing between processes, the interpreter checks, after each tick, whether the instruction that it just executed was a jump. If so, and the current process has exceeded its allowed number of ticks (1024), we deschedule it. Since we only do this on jumps, there is no need to preserve the three evaluation stack registers. This means that context switches are extremely fast as we only need to change **W**.

```
bool isSliceable(const Byte instrByte) {
        u8 iCode = instrByte >> 4;
        return iCode == 0x9; /*  j */
}

void tick() {
    if (W != Sched::kNone) {
        /** Execute an instruction for some process. */
        const Byte instrByte = IMem[I];
        doInstr(instrByte);
        if (isSliceable(instrByte) && elapsedTicks >= kMaxSlice) {
            /** Timeslice the processes on jump instructions. */
            rescheduleCurrent();
        }
        elapsedTicks++;
    }
    /** Otherwise, there is no process currently running on the interpreter. */
    tickCount++;
}
```

3.3 Channels, PAR & ALT

Communication is done through the *inword* and *outword* instructions. Each channel is associated with a special word in memory called the *channel word*. We identify channels with this special word. Since our language does not implement arrays, we allow only for communications that consist of a single word. Say two processes with descriptors P and Q try to communicate through a channel with channel word C. Since only one process runs on the interpreter at a time, one of the processes will become ready first [10]. Suppose that process is P - then P writes its descriptor into the channel word ($M[C] = P$). It then writes **B** to the special location $M[W-2]$ and deschedules itself.

At some point afterwards, Q will be scheduled and execute an input or output instruction. It reads $M[C]$ and finds that it contains a valid address, which it interprets as the process descriptor P. If Q is receiving, then it writes the byte that P had stored to the address specified by B ($M[B] = M[P-2]$). If it is sending, it examines $M[P-1]$ to determine if the process that it would write to is currently executing an ALT. If so, then it 'delays' the output by writing its descriptor into C as if it were the first process to become ready. This is because the code executing the alternation needs to be notified when a channel has become available for communication without actually performing the input. If that alternative is chosen, the code on the receiving side will complete the communication. Otherwise, it writes **B** into the address that P had stored previously ($M[M[P-2]] = B$).

```
case 0x0C: {
    /** inword
      * A := Channel address.
      * B := Input address. */
    if (M[A] != Channel::kCleared) {
        /** The channel word contains a valid process descriptor. */
        Word Q = M[A];
        M[B] = M[Q - 2];
        M[A] = Channel::kCleared;
        scheduleProc(Q);
    } else {
        /** Write ourselves in the channel word to mark our intent to
          * receive, then deschedule ourselves. */
        M[A] = W;
        M[W - 2] = B;
        descheduleCurrent();
    }
    A = C;
    break;
case 0x0D: {
    /* outword
     * A := Channel address.
     * B := Word to output. */
    if (M[A] != Channel::kCleared) {
        /** The channel word contains a valid process descriptor. */
        Word Q = M[A];
        if (M[Q - 1] != Alt::kNone) {
            /** The receiving process is currently executing an ALT,
              * and is waiting for input to select an alternative.
              * We do not communicate yet, but instead write our own
              * descriptor in the channel word to indicate that we
              * are ready for communication, then schedule the receiving
              * process. Its code will handle the communication. */
            M[A] = W;
            scheduleProc(Q);
        } else {
            /** In unconditional communication, addr contains the
              * word to be sent. */
            M[Q - 2] = B;
            M[A] = Channel::kCleared;
            scheduleProc(Q);
        }
    } else {
        /** Write ourselves in the channel word, to mark our intent to
          * send, then deschedule ourselves. */
        M[A] = W;
        M[W - 2] = B;
        descheduleCurrent();
    }
    A = C;
    break;
}
```

startp and *endp* are used to implement the *PAR* construct. Upon execution of a *PAR*, the number of parallel processes is stored at $M[W-1]$. It is the duty of each *endp* instruction, located at the end of each process, to decrement this value by 1 in order to show to the calling process that its child has stopped.

```
case 0x0A: {
    /** startp
          * A := Address of new workspace.
          * B := Offset from after startp to start of process. */
    scheduleProc(A);
    M[A - 4] = I + B;
    I += B;
    A  = C;
    break;
}
case 0x0B: {
        /** endp
          * A := Continuing process workspace. */
    if (M[A - 1] > 1) {
        M[A - 1]--;
        descheduleCurrent();
    } else {
        W = A;
        I = M[W - 4];
    }
    A = B;
    B = C;
    break;
}
```

We use seven instructions to implement *ALT* constructs. These correspond to updating the state of the alternation, enabling and disabling different kinds of guards. We store the *state* of the alternation at special location $M[W-1]$. There are four possible values that it may take - *kEnabling*, *kReady*, *kWaiting* and *kNone*. We need *kNone* for an output to be able to tell if the receiving process is performing an *ALT* or not.

```
case 0x0E: {
    /** enbs
      * A := Boolean that controls guard. */
    if (A) {
        /** Guard is ready. Set the state of the ALT to ready. */
        M[W - 1] = Alt::kReady;
    }
    A = B;
    B = C;
    break;
}
case 0x0F: {
    /** enbc
      * A := Boolean that controls guard.
      * B := Channel address.
      */
    if (A) {
        if (M[B] != Channel::kCleared) {
            Word Q = M[B];
            if (Q != W) {
                /** Another process is already sending on B. */
                M[W - 1] = Alt::kReady;
            }
        } else {
            /** Write ourselves in the channel word to initiate
              * communication. */
            M[B] = W;
        }
    }
    A = C;
    break;
}
case 0x10: {
    /** diss
      * A := Offset from altend to start of alternative code.
      * B := Boolean that controls guard.
      */
    if (B && M[W - 2] == Alt::kUnselected) {
        /** Select this alternative. */
        M[W - 2] = A;
    }
    A = C;
    break;
}
case 0x11: {
    /** disc
      * A := Offset from altend to start of alternative code.
      * B := Boolean that controls guard.
      * C := Channel address. */
    if (B && M[W - 2] == Alt::kUnselected &&
        M[C] != Channel::kCleared) {
        /** Select this alternative. */
        M[W - 2] = A;
    }
    break;
}
case 0x12: {
    /** alt */
    M[W - 1] = Alt::kEnabling;
    break;
}
case 0x13: {
    /** altwt */
    if (M[W - 1] != Alt::kReady) {
        M[W - 1] = Alt::kWaiting;
        M[W - 2] = Alt::kUnselected;
        descheduleCurrent();
    }
    break;
}
case 0x14: {
    /** altend
      * Continue as selected alternative. */
    I += M[W - 2];
    break;
}
```

3.4 Assembler

The assembler converts low-level code consisting of statements and labels into machine-readable instructions. The interface to the assembler is represented by the *Statement* abstraction. Statements represent series of consecutive machine instructions that form a single coherent logical unit. For example, *ldc 256* is a single statement that represents the series of instructions *pfix 1*, *pfix 0*, *ldc 0*. A statement consists of an instruction and a value, which can be either numeric, a label, or the distance between two labels. The constructor of a statement decides at compile-time, through the use of constexpr-maps, if the string literal that it was passed denotes an instruction or an operation, and generates bytecode appropriately. This simplifies the work of the compiler writer, who may for example write *Statement("alt")* to correspond to the instruction sequence *pfix 0*, *opr 0*.

```
Statement::Statement(const std::string& __iCode) {
        iStr = __iCode;
        if (kInstrs.contains(__iCode)) {
                iCode = kInstrs.at(__iCode);
        } else {
                iCode = kOpr;
                iVal  = Value(kOpers.at(__iCode));
        }
}
```

3.5 Interface to the Compiler

We provide the *Code* interface to the compiler, in much the same way as *Statement* provides an interface to the virtual machine below. Blocks of code consist of a sequence of statements, together with some labels. They may be created from individual statements and labels, concatenated, and assembled.

Labels are global, unique identifiers that are assigned from a static counter. Statements that reference a particular label do so by referencing its identifier. Within the context of a block of code, each label may be assigned an index. A label with index *i* is taken, in the final bytecode, to point to the offset in the assembled block that contains the first instruction of statement *i*. The abstraction is held together by the fact that we only ever reference byte offsets that land on the first instruction of a statement, and never strictly inside.

Inside a code block, we hold the statements as a vector, and the labels as a hash map from identifiers to indexes. Since each label holds an index into a sequence, care must be taken when concatenating two code blocks to ensure that the labels from the second block still point to where they should. The simple solution to this is to increment each index from the second block by the number of statements contained in the first block.

```
struct Code {
    std::vector<Statement> allSs;
    std::unordered_map<u32, u32> allLabels;

    void Code::operator +=(const Code& o) {
        for (auto& [i, n] : o.allLabels) {
            allLabels[i] = allSs.size() + n;
        }
        allSs.insert(allSs.end(), o.allSs.begin(), o.allSs.end());
    }

    ...
}
```

Blocks are constructed either from single statements, labels or other blocks of code. Constructing a block from a label is considered to have move semantics - that is, using the same label object again to construct a different block leads to undefined behaviour. Constructing a block from a statement forwards the arguments to the constructor of the statement, for ease of use. In particular, C++11's list initialization permits blocks to be created from a list of other blocks, leading to convenient syntax as will be shown throughout Section 4.

```
Code::Code(const Label& l) {
    allSs.clear();
    allLabels[l.ID] = 0;
}

Code::Code(std::initializer_list<Code> l) {
    allSs.clear();
    allLabels.clear();
    for (const Code& c : l) {
        (*this) += c;
    }
}
```

3.6 Resolving Offsets

Sometimes, instructions need to load, store or otherwise manipulate values that represent the distance between two labels after assembly time. This happens implicitly for every jump (with the first label taken as pointing to past the statement that made the jump), since the variable size of statements means that the offset of the jump is not known until assemble time. Other times, we need to explicitly load the distance between two labels into a register (e.g. *diss* expects the offset between the *altend* instruction and the alternative code in **A**). We formally define the *distance* between two labels as the signed total number of machine instructions that make up the code between them. Given that machine instructions are always exactly one byte long, this corresponds to the number of bytes, a distance ultimately counts bytes.

Since distances count instructions, and must themselves be constructed through prefixing, it is possible for the instruction size of the statements that use the distance to be interdependent. In order for code to be correctly assembled, the assembler must find some consistent assignment of sizes to the instructions. There may be many solutions (in particular, it is possible to encode each distance with 8 instructions, the maximum it takes to encode any 64-bit number), but it is naturally in our interest to minimise the total program length. We therefore implement an algorithm due to Inmos [10] that consists of two stages.

First, we iterate over all the statements and associate with each one an estimate of the minimal number of bytes needed to encode it. If the statement does not refer to any labels or distance, then we calculate it on the spot. Otherwise, set it to 0.

```
std::vector<u32> cycSs;
for (u32 i = 0; i < allSs.size(); i++) {
        auto& s = allSs[i];
        auto& v = s.iVal;
        if (v.isLabel() || v.isDistance()) {
                /** Size is dependent on other sizes. */
                cycSs.push_back(i);
                s.iSize = 0;
        } else {
                /** Size is fixed. */
                s.iSize = countPrefixSeq(v.numValue);
        }
}
```

Second, we loop. At each step we calculate the value of each offset under the size assumptions that we set out at the previous step. We then count the minimal number of bytes needed to encode these values, and compare them against the size that we assumed. If they are different, then we update our estimate and loop again.

```
/** Calculate the sum of statement sizes in the range [i, j]. */
auto getSum = [&](u32 i, u32 j) -> i32 {
        i32 s = 0;
        for (u32 k = i; k <= j; k++) {
                s += allSs[k].iSize;
        }
};

...

/** Calculate the numeric value that the right hand side of
  * this instruction would have, under the present assumptions
  * about the other instructions. */
auto calculateDistance = [&](const u32 i) -> i64 {
        const auto& s = allSs[i];
        const auto& v = s.iVal;
        i64 d = 0;
        if (v.isLabel()) {
                /** Jump from the current statement @l, to statement @r. */
                const u32 l = i;
                const u32 r = allLabels[v.labID];
                if (l < r) {
                        /** Jump forwards. Since the jump is the last instruction
                          * in statement @i, and we want to land on the first
                          * instruction in statement @j, the offset is the sum of
                          * the sizes of all statements in between. */
                        d = getSum(l + 1, r - 1);
                }
                else {
                        /** Jump backwards. We need to traverse both the current
                          * statement and statement @r again, so offset is negative
                          * the sum of all statements from r to l inclusive. This
                          * case also handles jumping to the same statement (like
                          * in the case of STOP), where we just want to traverse
                        * @l again. */
                        d = -getSum(r, l);
                }
        } else {
                /** Calculate the distance between two specified labebls. */
                const u32 l = v.d.lID;
                const u32 r = v.d.rID;
                if (l < r) {
                        d =  getSum(l + 1, r);
                } else {
                        d = -getSum(r + 1, l);
                }
        }
};

...

bool fixPoint;
do {
        fixPoint = true;
        for (u32 i : cycSs) {
        auto& iSize = allSs[i].iSize;
        const i64 d = calculateDistance(i);
        const u32 c = countPrefixSeq(d);
        if (iSize != c) {
            /** We have not yet reached a fixed point. */
            iSize  = c;
            fixPoint = false;
        }
    }
} while(!fixPoint);
/** Assign the calculated offset to the values of these instructions. */
for (u32 i : cycSs) {
        allSs[i].iVal = Value(calculateDistance(i));
}
```

This algorithm always terminates and produces decent code, but it will never perfectly prefix some pathological examples. In the worst case, the time complexity is quadratic in the number of statements. We can improve this to linearithmic time by using segment trees to compute the range sums. Alternatively (and probably faster in a practical case), we can use Fenwick tree and express each sum query as the difference of two prefix queries.

3.7 Generating Bytecode

For efficient mapping of string literals to instruction and operation codes, we employ two constexpr-maps, described in detail in Appendix B. We use the following functions to count and generate minimal prefix sequences:

```
u8 countPrefixSeq(i64 vImm) {
    if (vImm < 0) {
    vImm = ~vImm;
    }
    u8 c = 0;
    while (vImm > 0) {
        c++;
    vImm >>= 4;
    }
    return c;
}

void genPrefixSeq(const Byte iCode, const i64 vImm, Byte*& p) {
    if (vImm >= 16) {
        genPrefixSeq(kPfix, vImm >> 4, p);
    } else if (vImm < 0) {
        genPrefixSeq(kNfix, (~vImm) >> 4, p);
    }
    *p = (iCode << 4) | (vImm & 0xF);
    p++;
}
```

After each statement has been assigned a size, we can assemle the code in a single pass. Since the size of a statement may be higher than the length of its minimal prefix sequence, care must be taken to pad out with no-ops where necessary. We assemble directly into the instruction memory of the interpreter.

```
void Code::assembleAt(Byte*& p) {
        resolveDistances();
        for (Statement& s : allSs) {
                auto n = s.iVal.numValue;
                genPrefixSeq(s.iCode, n, p);
                for (u32 c = countPrefixSeq(n); c < s.iSize; c++) {
                        /** Some statements may be registered as compiling into more
                          * instructions than they actually do, as part of the algorithm
                          * to resolve distances. We pad out the remaining instructions
                          * with no-ops. */
```

```
                    *p = kNoop;
                    p++;
                    c++;
                }
        }
}
```

4 Compiling

4.1 Lexing & Parsing

We employ *re2c* [11] as a lexer generator and the industry standard *bison* [12] as a parser generator. We bring these together along with our own code through using semi-specialised *CMake* files.

We implement the *off-side rule* for our language, expressing code blocks by their indentation. This is achieved by introducing special INDENT and DEDENT tokens to correspond to opening and closing braces in languages like C++. In addition to this, we extract NEWLINE tokens to use as statement separators. Being aware of the current level of indentation requires that the lexer take on extra state. It is easier to add this state through *re2c* than the more usual *flex* since it allows us to define *yylex()* ourselves and inject auto-generated code into it, which is why we opted for it.

The off-side rule also introduces some unique problems for parsing. It is generally more difficult for parsers to recognise statements when their productions end in a delimiter than start in one. Coupled with the off-side rule, this introduced many ambiguities to the grammar. Our solution was to turn the grammar upside-down, and instead start each process and declaration with a newline. This creates a new problem where newlines and indentation tokens are extracted the wrong way around, which can be solved by adding more state to the lexer.

4.2 Abstract Syntax

4.2.1 Identifiers & Literals

Identifiers to denote names, like variables and channels. In our abstract syntax, we equate identifiers with simple strings. Literals are integer values of up to 64 bytes, TRUE or FALSE.

4.2.2 Expressions

Syntactically, expressions contain a type, list of subexpressions, and optionally an identifier or literal. Since there is no operator precedence in occam [3], all expressions are explicitly constructed with parentheses, the number of subexpressions is at most two.

```
struct Expr {
    enum class Type {
        kNoop,
        kIdent,
        kLiteral,
        /** Arithmetical types. */
        kNeg,
        kAdd,
        kSub,
        kMul,
        kDiv,
        kRem,
        /** Relational types. */
        kEq,
        kNeq,
        kLt,
        kGt,
        kLeq,
        kGeq,
        /** Logical types. */
        kNot,
        kAnd,
        kOr
    };

    Type t;
    i64  v;
    std::vector<Expr*> es;
};
```

4.2.3 Definitions

We store additional information about syntactic structures in *definitions* and use this information during code generation. For variables and channels, the definition will have to contain the offset in the current frame at which they are located.

```
enum class DeclType {
    kNone,
    kVar,
    kChan
};

struct Definition {
    DeclType t;
    i32 addrOf;

    Definition(): t(DeclType::kNone) {}

    explicit Definition(const DeclType& _t, const u32 _addrOf):
        t(_t),
        addrOf(_addrOf)
    {}
};

typedef std::pair<Identifier, DeclType> Declaration;
```

Declarations are pairs of identifiers and *DeclType* are parsed from the declarations in the source file and are stored next to each process during the building of the abstract syntax tree. They are used to generate definitions during code generaiton.

4.2.4 Processes

```
struct Process {
        enum class Type {
                kNone,
                kSkip,
                kStop,
                kAssignment,
                kInput,
                kOutput,
                kLoop,
                kSequence,
                kConditional,
                kParallel,
                kAlternation
        };

        Type t;
        std::vector<Declaration> decls;

        Process(Type __t): t(__t) {}

        void addDecls(std::vector<Declaration>&& ds) {
                for (auto&& d : ds) {
                        decls.push_back(std::move(d));
                }
        }

        void analyse(Frame& fr);
        Code genCode(Frame& fr);
};
```

We subclass *Process* for each of the types enumerated and include extra members depending on the type. We omit destructors and other miscellanea.

```
struct Assignment : Process {
        Identifier var;
        Expr* expr;

        explicit Assignment(Identifier&& v, Expr* e):
                  Process(Type::kAssignment),
                  var (std::move(v)),
                  expr(e)
        {}

    ...
};

struct Input : Process {
        Identifier chan;
        Identifier var;

        explicit Input(Identifier&& c, Identifier&& v):
            Process(Type::kInput),
            chan(std::move(c)),
                   var (std::move(v))
        {}
};

struct Output : Process {
        Identifier chan;
        Expr* expr;

        explicit Output(Identifier&& c, Expr* e):
            Process(Type::kOutput),
            chan(std::move(c)),
            expr(e)
        {}
    ...
};

struct Loop : Process {
        Expr* guard;
        Process* pr;

        explicit Loop(Expr* g, Process* p):
                Process(Type::kLoop),
                guard(g),
                pr   (p)
        {}

    ...
};

struct Sequence : Process {
        std::vector<Process*> seqs;

        explicit Sequence(std::vector<Process*>&& ss):
                Process(Type::kSequence),
                seqs(std::move(ss))
        {}

        ...
};

struct Parallel : Process {
        std::vector<Process*> pars;

        explicit Parallel(std::vector<Process*>&& ps):
                Process(Type::kParallel),
                pars   (std::move(ps))
        {}

        ...
};
```

To implement conditionals, we first define the auxiliary data structure *Choice*:

```
struct Choice {
        Expr* guard;
        Process* pr;

    explicit Choice(Expr* e, Process* p):
                guard(e),
                pr(p)
        {}
};

struct Conditional : Process {
        std::vector<Choice*> choices;

        explicit Conditional(std::vector<Choice*>&& c):
                Process(Type::kConditional),
                choices(std::move(c))
        {}

    ...
};
```

Same for alternatives - first define the auxiliary data structures *Guard* and *Alternative*:

```
struct Guard {
        Expr* boolean;
        Process* comm;

    explicit Guard(Expr* b = nullptr, Process* p = nullptr):
                boolean(b),
                comm(p)
        {}

        ...
};

struct Alternative {
        Guard*  guard;
        Process* proc;

    explicit Alternative(Guard* g, Process* p):
                guard(g),
                proc (p)
        {}
        ...
};
```

4.3 Analysis

Once we have built the abstract syntax tree, we run a depth-first-search on it in order to prepare it for code generation. We are interested in pre-generating all the activation records in the program. To do this, first we group all the processes into as few of them as possible. Formally, each component of a sequence, each choice of a conditional, and each alternative of an alternation are separate process with their own activation record and local variables. This is reflected in syntactic structure of programs, where each process may optionally start with a list of declarations. Practically, it would be very slow to perform a context switch every time we move processes within a SEQ. Likewise, only one choice, and one alternative respectively, eventually executes. For all intents and purposes we could, and for efficiency's sake should, treat all of these processes as though they were a single, aggregate process, with a shared activation record.

This strategy is a triple win. First, it is easier to code. It greatly speeds up the code by making it more local and reducing the number of context switches. But it also allows us to heavily optimise the memory footprint of the program. Since only a single process runs at a time, we can share offsets in the activation record between the local variables of processes. We will implement and use a stack mechanism for allocating these slots that will be provided by the following abstraction.

The *frame* abstraction aggregates information about the current activation record. Each frame contains additional information such as an identifier, its statically allocated address and, importantly, a pointer to the enclosing frame. A frame is responsible for a connected region of processes in the abstract syntax tree than run sequentially and share the same address space. We store the frames into a global array *allFrames*.

Next, we recurse through the processes, creating frames as we go, calculating their sizes and laying them out in memory. The only place where we need to create new frames is when evaluating a parallel construct - otherwise, we just aggregate everything onto the frame we already have. We pass the current frame size as we are recursing, while pretending to allocate variables and channels.

```
void Process::analyse(Frame* fr, u32 fSize) {
    /** Pretend to allocate the local variables to calculate the frame size. */
    fSize += decls.size();
    fr->frameSize = std::max(fr->frameSize, fSize);
        switch (t) {
        /** Recursively analyse subprocesses and expressions. */
                case Type::kAssignment: {
                        auto d = static_cast<Assignment*>(this);
                auto e = d->expr;
                /** Inside this call fr->frameSize will be updated again. */
                e->analyse(fr, fSize);
                break;
            }

            ...
            /** Similar cases. */
            ...
        }
        fSize -= decls.size();
}
```

Of particular interest is the PAR case, since that is the only place that we generate new frames.

```
case Type::kParallel: {
    auto d = static_cast<Parallel*>(this);
    /** Create a new frame for each parallel subprocess. Since we continue as the last process,
      * only generate pars.size() - 1 frames. */
    for (u32 i = 0; i < d->pars.size() - 1; i++) {
        auto p = d->pars[i];
        auto f = fr->createSubframe();
        p->analyse(f, 0);
    }
    break;
}
```

Analysing expressions means calculating their *depth*, the number of stack locations that are required in order to calculate them. Since we only have three evaluation stack registers, evaluating expressions of depth $d(e) > 3$ will require $t(e) = d(e) - 3$ temporary locations. Generally we can take $t(e) = max(0, d(e) - 3)$. Temporaries are handled by the same stack mechanism as local variables.

```
/** Calculate the depth as the maximum of the depths of subexpressions,
  * plus one if they are equal. */
void Expr::analyse(Frame* fr, u32 fSize) {
        for (auto e : es) {
                e->analyse();
        }
        if (t == Type::kLiteral || t == Type::kIdent)  {
                d = 1;
        } else if (t == Type::kNeg || t == Type::kNot) {
                d = es[0]->d;
        } else {
                auto l = es[0]->d;
                auto r = es[1]->d;
                d = std::max(l, r);
                if (l == r) {
                        d++;
                }
        }
    /** Update the frame size with the number of temporaries that this
      * expression needs. */
    if (d > 3) {
        fr->frameSize = std::max(fr->frameSize, fSize + d - 3);
    }
}
```

The frame abstraction also provides services to code generation like getting the definition of an identifier, generating static chains for non-local variable access, and allocating and deallocating local variables and temporaries.

By taking advantage of the sequential execution of the processes that make up the frame, we can use a stack to dish out and take back memory locations as we are recursing through the abstract syntax tree. In practice, it is only necessary to keep the index of the top of the stack. Evaluating deep expressions also requires the use of temporary memory locations, which we treat as an extension of the evaluation stack. As the number of these can be determined at compile time, for all intents and purposes we can treat them as local variables, and they are provided for by the same mechanism.

```
i32 nextLocal = 1;

i32 allocLocal() { return nextLocal++; }

void freeLocal() { nextLocal--; }
```

A *scope* is a mapping from identifiers to definitions and is the basic unit of containment for variables and channels. Each process has its own, possibly empty, scope containing its local variables. At any point during code generation, a number of scopes may be active in the current frame, organised as a stack. These scopes correspond to the local declarations of the processes on the way from the current location to the root of the abstract syntax tree. We create the definitions here, in the frame.

```
typedef std::map<Identifier, Definition> Scope;

std::vector<Scope> allScopes;
```

```
void define(const std::vector<Declaration>& decls) {
    allScopes.emplace_back();
    for (const auto& [name, dtype] : decls) {
        allScopes.back()[name] = Definition(dtype, allocLocal());
    }
}

void undef() {
    auto& defs = allScopes.back();
    for (u32 i = 0; i < defs.size(); i++) {
        freeLocal();
    }
    allScopes.pop_back();
}
```

We store the scopes as a stack inside the frame, and provide a way of retrieving the definition of individual identifiers. We also provide a means of getting the definition for a particular identifier. *get* traverses the active frames in reverse order, and within each frame traverses scopes in reverse order, so that it may find the last definition that has been associated with a particular identifier. It counts the number of frame changes and returns this value to the caller, which they can use to determine the length of the static chain associated with that identifier at the point that they are generating code. In using this traversal method, we ensure that variable and channel names can be shadowed.

```
std::pair<const Definition&, u32> get(const std::string& name) {
    for (auto i = allScopes.crbegin(); i != allScopes.crend(); i++) {
        if (auto j = i->find(name); j != i->end()) {
            return std::make_pair(j->second, 0);
        }
    }
    /** If this is not the root frame. */
    if (parentID != ID) {
        const auto& [defn, l] = allFrames[parentID]->get(name);
        return std::make_pair(defn, l + 1);
    }
}
```

Frames also provide the means for generating static chains, which are necessary to access a non-local address. The final instruction of the chain depends on the kind of access that we want to perform, and may be either *ldnl*, *stnl* or *ldnlp*.

```
enum class Instruction {
    kLoad,
    kStore,
    kPointer
};

std::string getInstr(const Instruction& instr, bool isLocal) { ... }

Code genStaticChain(const std::string& n, const Instruction& instr) {
    Code code;
    auto [defn, l] = get(n);
    i32 d = defn.addrOf;
    if (l > 0) {
        code += Code("ldl", 0);
        l--;
        while (l > 0) {
            /** Go up static link. */
            code += Code("ldnl", 0);
            l--;
        }
        code += Code(getInstr(instr, false), d);
    } else {
        /** Identifier was defined in the current process. */
        code += Code(getInstr(instr, true), d);
    }
}

Code genStore(const std::string& n) {
    return genStaticChain(n, Instruction::kStore);
}

Code genLoad (const std::string& n) {
    return genStaticChain(n, Instruction::kLoad);
}

Code genAddr (const std::string& n) {
    return genStaticChain(n, Instruction::kPointer);
}
```

4.4 Code Generation

As we are recursing down the abstract syntax tree generating code, we always keep a pointer to the current frame.

4.4.1 Expressions

Code generation for expression can be divided into five categories - arithmetical expressions, relational expressions, logical expressions, constants and identifiers.

Constants and identifiers are the simplest.

```
if (t == Type::kLiteral) {
    /** Load the literal stored in the expression. */
        return Code("ldc", v);
} else if (t == Type::kIdent) {
        return fr->genLoad(ident);
}
```

The workhorse for the other three cases is the *genOper(l, r, t, fr)* function, which generates code for loading the sub-expressions r and l onto the stack (in that order), under the assumption that they will be used to evaluate an expression of type t. Note that we expect l to be on the top of the evaluation stack. The main use of *genOper* is encapsulating the allocation and use of temporaries for deep expressions.

```
std::string toStr(const Expr::Type& t) { ... }

Code genOper(Expr* const l, Expr* const r, const Type t, Frame* fr) {
    Code code;
    if (l->d > r->d) {
        if (r->d > 2) {
            i32 x = fr->allocLocal();
            code += {
                l->genCode(fr),
                Code("stl", x),
                r->genCode(fr),
                Code("ldl", x),
            };
            fr->freeLocal();
        } else {
            /** If we generate l first, both expressions fit on the stack. */
            code += {
                l->genCode(fr),
                r->genCode(fr)
            };
            if (t != Type::kAdd && t != Type::kMul) {
                /** If we swapped l and r, we must swap the first places on the
                  * stack here, unless the operation that we are implementing
                  * is commutative, like addition and multiplication. */
                code += Code("rev");
            }
        }
    } else {
        /** Generate r first. This is preferrable since we do not need to swap. */
        if (l->d < 3) {
            code += {
                r->genCode(fr),
                l->genCode(fr),
            };
        } else {
            i32 x = fr->allocLocal();
            code += {
                l->genCode(fr),
                Code("stl", x),
                r->genCode(fr),
                Code("ldl", x),
            };
            fr->freeLocal();
        }
    }
    code += Code(toStr(t));
    return code;
}
```

Generating arithmetical expressions becomes quite simple:

```
Code genArithmetic(Expr* e, Frame* fr) {
    Code c;
    auto t = e->t;
    auto l = e->es[0];
    if (t == Type::kNeg) {
        c = {
            l->genCode(fr),
            Code("neg")
        };
    } else {
        auto r = e->es[1];
        c = genOper(l, r, t, fr);
    }
    return c;
}
```

For logical and relational expressions we include a helper operator:

```
Code operator ~(const Code& c) {
    return {
        c,
        Code("eqc", 0)
    };
}

Code genRelational(Expr* e, Frame* fr) {
    Code c;
    auto t = e->t;
    auto l = e->es[0];
    auto r = e->es[1];
    if (t == Type::kEq || t == Type::kNeq) {
        c  = genOper(l, r, Type::kSub, fr);
        c += Code("eqc", 0);
        if (t == Type::kNeq) {
            c = ~c;
        }
    } else if (t == Type::kLt || t == Type::kGeq) {
        c = genOper(r, l, Type::kGt, fr);
        if (t == Type::kGeq) {
            c = ~c;
        }
    } else if (t == Type::kGt || t == Type::kLeq) {
        c = genOper(l, r, Type::kGt, fr);
        if (t == Type::kLeq) {
            c = ~c;
        }
    }
    return c;
}

Code genLogical(Expr* e, Frame* fr) {
    Code c;
    auto t = e->t;
    auto l = e->es[0];
    if (t == Type::kNot) {
        c = ~(l->genCode(fr));
    } else {
        auto  r = e->es[1];
        Label lab;
        if (t == Type::kAnd) {
            c = {
                l->genCode(fr),
                Code("cj", lab),
                r->genCode(fr),
                Code(lab)
            };
        } else {
            c = {
                ~(l->genCode(fr)),
                Code("cj", lab),
                r->genCode(fr),
                ~(r->genCode(fr)),
                Code(lab)
            };
            c = ~c;
        }
    }
    return c;
}
```

4.4.2 Processes

Before generating any code, we define the variables and channels that are local to this procedure. After generation, we destroy this scope.

```
Code Process::genCode(Frame* fr) {
    Code code;
    fr->define(decls);
    ...
    fr->undef();
    return code;
}
```

Care must be taken for STOP to ensure that each instance is represented by a different loop.

```
case Type::kSkip: {
        /** Do nothing. */
        break;
}
case Type::kStop: {
        Label l;
        code += {
                Code(l),
                Code("j", l)
        };
        break;
}
```

Generating code for an assignment means generating code for the expression on the right-hand side, then code for storing into the address of the identifier.

```
case Type::kAssignment: {
        auto d = static_cast<Assignment*>(this);
        code += {
                d->expr->genCode(fr),
                fr->genStore(d->var),
        };
        break;
}
```

Input and output are similarly simple. We generate the code for either the address that we are receiving into, or the expression that we are outputting, followed by the address of the channel, followed by either the *inword* or *outword* instructions.

```
case Type::kInput: {
        auto d = static_cast<Input*>(this);
        code += {
                fr->genAddr(d->var),
                fr->genAddr(d->chan),
                Code("inword")
        };
        break;
}
case Type::kOutput: {
        auto d = static_cast<Output*>(this);
        code += {
        d->expr->genCode(fr),
                fr->genAddr(d->chan),
                Code("outword")
        };
        break;
}
```

For while loops, we use two labels - one for the start of the loop and one for the end. We test the condition at the start of the loop, and jump to the end label if it is false. Note the use of an unconditional jump (*j*) at the end - this is to allow the virtual machine to timeslice processes that are running busy loops.

```
case Type::kLoop: {
        auto d = static_cast<Loop*>(this);
        Label l;
        Label endl;
        code += {
                Code(l),
                d->guard->genCode(fr),
                Code("cj", endl),
                d->pr->genCode(fr),
                Code("j", l),
                Code(endl)
        };
        break;
}
```

For sequences, we simply generate code for all subprocesses, in order.

```
case Type::kSequence: {
        auto d = static_cast<Sequence*>(this);
    for (Process* s : d->seqs) {
                code += s->genCode(fr);
        }
        break;
}
```

Conditionals are a bit more tricky - we define an end label and then generate code for each choice separately. For any given choice, we define a label that points to the end of the choice, to be jumped to in case the guard is false. We generate code for the guard first, then check that it is true, then generate code for the process. Since we do not want to take multiple choices, we add an unconditional jump to the end label after the process.

```
case Type::kConditional: {
        auto d = static_cast<Conditional*>(this);
        Label endl;
        for (auto c : d->choices) {
                Label falsel;
                auto& g = c->guard;
                auto& p = c->pr;
                code += {
                        g->genCode(fr),
                        Code("cj", falsel),
                        p->genCode(fr),
                        Code("j", endl),
                        Code(falsel)
                };
        }
        code += Code(endl);
        break;
}
```

Parallels require us to hold two arrays of labels. When executing a parallel composition of K processes, the parent process first writes K into the special workspace location $M[W-1]$, to serve as a counter for the number of processes that are yet to finish - when a process ends, it executes the *endp* instruction, which decrements this value by 1. It then launches $K-1$ subprocesses through the *startp* instruction and continues as the K-th. When launching a new process, we need to load its workspace into **A**, and the offset from the after *startp* to the first instruction of the process. We generate code for each process as usual, except that we finish by loading the workspace of the parent process, which it would continue as, onto the stack, then exiting with the *endp* instruction. In total, we need $2*K$ labels: $K-1$ to start the processes, K to end the processes and an extra one to force a time-slice.

```
case Type::kParallel: {
        auto d = static_cast<Parallel*>(this);
        const i32 k = d->pars.size();
        Label startl;
        std::vector<Label> lcode(k);
        std::vector<Label> llaunch(k);
        /* Generate code for the control structure of PAR. */
        code += {
                Code(startl),
                Code("ldc",  k),
                Code("stl", -1)
        };
        /** Launch subprocesses i for i = [0, k). */
        for (int i = 0; i < k - 1; i++) {
                auto& p = d->pars[i];
                /** Change frames as we are recursing into a subprocess. */
                auto fchild = allFrames[fr->childIDs[i]];
                code += {
                        Code("ldc", lcode[i] - llaunch[i]),
                        Code("ldc", fchild->getWorkspace()),
                        Code("startp"),
                        Code(llaunch[i])
                };
        }
        /** Continue as process k. Jump to force the interpreter to time-slice here. */
        code += {
            Code(startl),
            Code("j", startl),
                d->pars[k - 1]->genCode(fr),
                Code("ldlp", -1),
                Code("endp")
        };
        /** Generate code for the parallel processes. */
        for (int i = 0; i < k - 1; i++) {
                auto& p = d->pars[i];
                auto fchild = allFrames[fr->childIDs[i]];
                code += {
                        Code(lcode[i]),
                        p->genCode(fchild),
                        /** Change back workspace. */
                        Code("ldl", fr->getWorkspace()),
                        Code("endp")
                };
        }
        break;
}
```

Alternations are the most difficult language construct to implement. During execution, an alternative may be in one of three states - *enabling*, *waiting* or *ready* - which we store in the special location $M[W-1]$ and manipulate through the instructions *alt*, *altwt* and *altend* respectively. We first set the state to *enabling*, then enable all of the guards, meaning that we evaluate the boolean that controls the *ALT* and optionally prepare the channel of the guard for input. Some of the guards - those with a true boolean and a ready channel, or no channel - might already be ready themselves, in which case we set the state of the *ALT* to *waiting*. The work of evaluating the boolean, setting up the channel and changing the state is encapsulated in the *enbs* (enable-skip) and *enbc* (enable-channel) instructions. An alternative may only become ready if another process is sending a value through its channel. If that happens, the corresponding alternative is disabled, meaning that the index of the alternative is written to the special location $M[W-1]$. Thus, the order in which alternatives are enabled does not matter, but the order in which they are disabled is crucial, as it determines their priority. As before, the *diss* (disable-skip) and *disc* (disable-channel) instructions encapsulate for each guard the evaluation of the boolean expression, updating the ALT state and selecting the alternative.

```
case Type::kAlternation: {
        auto d = static_cast<Alternation*>(this);
        const i32 k = d->alts.size();
        Code  code;
        Label endl;
        std::vector<Label> altlabs(k);
        for (u32 i = 0; i < k; i++) {
                auto& a = d->alts[i];
                auto& p = a->proc;
                auto& g = a->guard;
                code += Code(altlabs[i]);
                if (g->comm != nullptr) {
                        /** Generate code for input before code for procedure. */
                        code += g->comm->genCode(fr);
                }
                code += p->genCode(fr);
                code += Code("j", endl);
        }
        code += Code("alt");
        /** Enable all the guards. */
        for (auto& a : d->alts) {
                auto& g = a->guard;
                if (g->comm == nullptr) {
                        /** Input is SKIP. */
                        code += g->boolean->genCode(fr);
                        code += Code("enbs");
                } else {
                        code += g->comm->genCode(fr);
                        code += g->boolean->genCode(fr);
                        code += Code("enbc");
                }
        }
        code += Code("altwt");
        /** Disable all the guards. Order here is important. */
        for (u32 i = 0; i < k; i++) {
                auto& a = d->alts[i];
                auto& g = a->guard;
                if (g->comm == nullptr) {
                        /** Input is SKIP. */
                        code += {
                                g->boolean->genCode(fr),
                                Code("ldc", endl - altlabs[i]),
                                Code("diss")
                        };
                } else {
                        code += {
                                g->comm->genCode(fr),
                                g->boolean->genCode(fr),
                                Code("ldc", endl - altlabs[i]),
                                Code("disc")
                        };
                }
        }
        code += Code("altend");
        code += Code(endl);
        break;
}
```

4.5 Bringing Everything Together

It is quite simple to join all the parts that we have designed so far. First, we run the parser, depositing the root of the abstract syntax tree in *rootPtr*. We then generate the root frame, run the analysis and assign an offset to each frame. We run the code generation, then assemble directly into the instruction memory of the interpreter. Finally, we run the interpreter.

```
int main() {
    std::ifstream f(argv[1]);
    std::string fileBuf(std::istreambuf_iterator<char>(f), {});
    /** Prepare the lexer state. */
    LexContext Cxt;
    Cxt.cursor = Cxt.marker = Cxt.start = fileBuf.c_str();
    /** Parse. */
    yy::parser parser(Cxt);
    parser.parse();
    /** Create root frame. */
    Frame* fr = new Frame(0, 0);
    allFrames.push_back(fr);
    /** Run the analysis. */
    rootPtr->analyse(fr);
    /** Statically allocate memory for all the frames. */
    u32 s = 0;
    for (auto fr : allFrames) {
        fr->addrStart = s;
        s += fr->frameSize;
    }
    /** Generate code. */
    Code allCode = rootPtr->genCode(fr);
    /** Boot the interpreter. */
    reset();
    /** Set the workspace of the first process, accounting for special locations. */
    W = 4; /
    I = 0;
    Byte* p = IMem;
    /** Generate code to load the first process. */
    allCode.assembleAt(p);
    /** Interpret. */
    while (true) {
        tick();
    }
}
```

5 Procedures & Recursion

One of the original aims of the project was to introduce procedures and recursion to our language. In order to do this, we would first introduce *procedures* to our language, then make changes to the memory allocation to enable recursion. We outline the changes below:

5.1 Procedures

We would need to extend our grammar to be able to parse procedures. We could do this by introducing the PROC token, as well as suitable productions. Procedures would be defined just prior to the process that they are local to, along with other channels and variables. We would define productions for parameters and arguments, which we expect to be a list of *Expr*.

We would need to extend our *Definition* interface. For channels and variables, it is enough to collect pairs of identifiers and *DeclType* from the source code and place them in a data structure inside their parent process for code generation. Introducing procedures, though, would require us to subclass *Definition* into something to the effect of *ProcDefn* to containa a list of formals and a pointer to the process that constitutes the body of the procedure.

Procedure frames are slightly different to process frames, since we also need to store the three evaluation stack registers, along with the instruction pointer for returning. Therefore, we would need to make the *frame* abstraction aware of the kind of frame that it is referencing and change the code for calculating frame size, allocating locals and generating static chains accordingly.

Introducing procedures would require some changes to the memory allocation. It is possible for parallel processes to make calls to the same procedure concurrently, meaning that we cannot create a single frame for the procedure. Instead, we should calculate during code analysis the maximum number N of processes that may call the procedure concurrently, and allocate N copies of that frame. In any event, a procedure would be unable to live in the workspace of its parent.

We would need to add a new type of process, *kCall*. We would generate the code for a procedure body next to the code of the parent procedure and use the *addrOf* member of *Definition* to refer to its entry point. We would collect the formals into a scope and add it to the current frame before heading into the procedure body. Finally, we would set aside N clones of the frame in memory to account for parallelism. Generating code for a procedure call would require some sort of mechanism for dynamically selecting which of the clone frames to use for the current invocation.

In the assembler, procedure calls would be implemented with the *call* and *ajw* instructions, which have been left empty for this purpose. *call* would expect the new workspace of the process to be passed in **A** and save the three evaluation stack registers and **I** at that address. We would use the latter to mmanually set the workspace back upon returning from a procedure.

5.2 Recursion

Implementing recursion would be very simple lexically - simply add an optional REC token to be specified before PROC. In the abstract syntax, a single extra boolean inside *ProcDefn* would suffice. However, enabling recursion would require some fundamental changes to our memory allocation scheme, as static allocation cannot support it.

Since recursion by definition means that we do not know how many frames will be generated at compile-time, we would need some form of dynamic memory allocation. Straight up stack allocation is, unfortunately, not compatible with the conflicting demands of recursive processes (which require potentially unbounded stack space) and unbounded parallelism (which would require us to divide the stack into arbitrarily small segments). There is no good way to avoid recursive processes overrunning their allotted space and breaking into the stack frame of a concurrent process.

One solution, and decidedly a simple one, would be allocate memory for activation records on demand from a heap. In practical terms, this means that each parallel process and each procedure invocation would need to ask for enough memory to hold its frame size. Since we would still know the frame sizes at compile time, this allocation and decallocation could be done at the machine level through specialist instructions. The main problem with heap allocation is fragmentation. Since we are running the language on a virtual machine, and not an actual chip, garbage collection remains a very viable and attractive possibility. However, this would slow down execution and also break the physical correspondence between interpreter actions and transputer instructions. Alternatively, we could employ *dynamic allocation from static memory pools* in order to help with fragmentation. This would mean dividing up the heap into fixed size chunks and serving these to processes upon their asking for memory.

6 Evaluation

In order to evaluate our work, we have compiled some occam programs and ran them through the compiler, and manually inspected the statements and also the instructions that the assembler produces. We loaded this code onto the virtual machine and tested that it runs without hanging. We ran one test for each of the process constructs, along with one more complex test. The tests also cover expression generation and the use of temporary locations. The full list is below:

```
-- Test 1: Alt
CHAN c:
CHAN d:
CHAN e:
PAR
  c!1
  d!2
  VAR x:
  VAR y:
  ALT
    TRUE & c?x
      e!x
    TRUE & d?y
      e!y

-- Test 2: Assignment
VAR x:
VAR y:
VAR z:
SEQ
  x := (((((((((1 + 2) + 3) + 4) + 5) + 6) + 7) + 8) + 9) + 10)
  y := (1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + (9 + 10)))))))))
  z := (((1 + 2) + (3 + 4)) / ((5 + 6) + (7 + 8))) * ((9 + 10) / (11 * 12))

-- Test 3: Input
VAR x:
CHAN c:
c?x

-- Test 4: Output
VAR x:
CHAN c:
c!(1+(2*(x/(4-3))))

-- Test 5: Sequence
VAR x:
VAR y:
CHAN c:
SEQ
  x := 1
  c!y

-- Test 6: Parallel
CHAN c:
PAR
  c!((1*2)+3)
  VAR y:
  SEQ
    c?y
    y := 123456789

-- Test 7: Skip
SKIP

-- Test 8: Stop
STOP

-- Test 9: Larger test
VAR x:
VAR n:
VAR finished:
SEQ
```

```
x := 0
n := 0
finished := FALSE
CHAN c:
WHILE NOT finished
  SEQ
    c?x
    IF
      x = 3
        finished := TRUE
      TRUE
        n := n + 1
```

7 Conclusion

7.1 Summary

We have succeeded in achieving most of the goals that we set out for this project. We designed a simple subset of occam that still implements the basic process framework, a virtual machine based on the Inmos transputer and a compiler to target it. While we did not implement procedures and recursion, we have provided a discussion of how this could be done.

7.2 Reflection

I got a lot of enjoyment out of this project as my second foray into the field of compilers (the first had been in my second year, due to the Computer Science department). As a future low-level programmer, I initially picked this project for the chance to gain more familiarity with the implementation of programming languages and sharpen my C++ skills through writing a large project. To this end, it has more than met my expectations, and am glad that I chose it. The project was quite open-ended to start with, although there were some guidelines for the results that we wanted to achieve.

I particularly enjoyed designing and implementing the assembler, and making the *Code* interface as streamlined as possible. I spent quite some time thinking about it and found it very rewarding, and am glad that it worked out.

Perhaps predictably, I had the most trouble with (and got the least enjoyment from) setting up the parser and the lexer. This was painstaking and, along with defining the grammar and ensuring that it is as unambiguous as possible, took up roughly half of the time I spent on this project.

One of the aims of the project was to extend our language to include procedures and recursion, and

I am disappointed that I was unable to complete this in the time allotted. I did some work towards it, but had to present it as a discussion as it became clear that I would not have enough time to implement and test it to the standard of the other parts of the project. I found the topic of recursion very interesting and am glad to have learned about limitations of static memory allocation, the ways that dynamic allocation can provide for recursion, and the drawbacks that they introduce.

On the whole, I found this project to be both very challenging and rewarding. It broadened my perspective on the way that compilers work, and I feel that it has been an important step in crystallizing my understanding of the implementation of programming languages from text to machine code. I have become a better programmer by taking it, and now feel confident to attempt reading about the large-scale design of C++ compilers, like gcc.

7.3 Future Work

Aside from implementing procedures and recursion, there are many other ways in which to extend the language. Some possibilities are the introduction of data types, aliases and arrays. Alternatively, one could implement functions, or other process types, like CASE that are not strictly necessary. On the interpreter side, there is space to add support for more precise memory operations and the emulation of physical components, including timers and links to other transputers. Since transputers were meant to be connected into farms, a possibility is to implement several communicating copies of the virtual machine. Since this is a language project, the sky is the limit, and there is certainly at least another project's worth of work on both the compiler and machine sides.

In addition to this, the project would really benefit from a large, systematized testing framework. Unit and regression testing should prevail here, and there should be tests that stress each part of the compiler individually, then all possible arrangements of program components within each other. Throughout development, I experimented with (and would recommend) GoogleTest as the framework in which to implement these tests.

8 Appendices

8.1 Appendix A

This appendix contains the informal specification of our language, as defined by the interface that we implement to the parser generator.

```
%token END 0
%token INDENT DEDENT NEWLINE "\n"
%token VAR CHAN
%token IDENTIFIER INT_LITERAL TRUE FALSE
%token SKIP STOP SEQ PAR IF ALT WHILE
```

```
%token L_PAR "(" R_PAR ")" PLUS "+" MINUS "-" TIMES "*" DIV "/"  REM "\\"
%token LT "<" GT ">" EQ "=" LEQ "<=" GEQ ">=" NEQ "<>" AND "&&" OR  "||" NOT
%token ASSIGNMENT ":=" AMPERSAND "&" INP "?"  OUTP "!" COLON ":" COMMA ","

%type<i64> INT_LITERAL
%type<Expr*> Expr Oper
%type<Identifier> IDENTIFIER

%type<Process*> Program Simple_Process Process Assignment Input Output
%type<Process*> Loop Conditional Sequence Parallel Alternation

%type<Guard*>  Guard
%type<Choice*> Choice
%type<Alternative*> Alternative
%type<Declaration>  Declaration

%type<std::vector<Declaration>>  L1_Declaration
%type<std::vector<Alternative*>> L1_Alternative
%type<std::vector<Process*>> L1_Process
%type<std::vector<Choice*>> L1_Choice


/** -----------------------------------------------------------------------------*/

%%

Program: Process END                                        { rootPtr = $1; }

Process
    : "\n" Simple_Process                                       { $$ = $2; }
    | L1_Declaration "\n" Simple_Process      { $$ = $3; $$->addDecls(M($1)); }
    ;

Simple_Process
    : SKIP                                              { $$ = makeSkip(); }
    | STOP                                              { $$ = makeStop(); }
    | Assignment
    | Input
    | Output
    | Loop
    | Conditional
    | Sequence
    | Parallel
    | Alternation
    ;

/** -------------------- Assignment   ----------------------------------------*/

Assignment: IDENTIFIER ":=" Expr            { $$ = new Assignment(M($1), $3); };

/** -------------------- Sequence   ------------------------------------------*/

Sequence: SEQ INDENT L1_Process                   { $$ = new Sequence(M($3)); };

/** -------------------- PAR   -----------------------------------------------*/

Parallel: PAR INDENT L1_Process                   { $$ = new Parallel(M($3)); };

/** -------------------- LOOP   ----------------------------------------------*/

Loop: WHILE Expr INDENT Process                      { $$ = new Loop($2, $4); };

/** -------------------- IF   ------------------------------------------------*/

Choice: "\n" Expr INDENT Process                   { $$ = new Choice($2, $4); };

Conditional: IF INDENT L1_Choice               { $$ = new Conditional(M($3)); };

/** -------------------- ALT -------------------------------------------------*/

Guard
    : Input                                    { $$ = new Guard(nullptr, $1); }
    | Expr "&" Input                                { $$ = new Guard($1, $3); }
    | Expr "&" SKIP                            { $$ = new Guard($1, nullptr); }
    ;

Alternative: "\n" Guard INDENT Process        { $$ = new Alternative($2, $4); };

Alternation: ALT INDENT L1_Alternative         { $$ = new Alternation(M($3)); };
```

```
/** -------------------- Communication ---------------------------------------*/

Input:   IDENTIFIER "?" IDENTIFIER           { $$ = new Input (M($1), M($3)); };

Output:  IDENTIFIER "!" Expr                    { $$ = new Output(M($1), $3); };

/** -------------------- Declarations ----------------------------------------*/

Declaration
    : "\n" VAR  IDENTIFIER ":"    { $$ = Declaration(M($3), DeclType::kVar);  }
    | "\n" CHAN IDENTIFIER ":"    { $$ = Declaration(M($3), DeclType::kChan); }
    ;

/** -------------------- Expressions & Exprs ---------------------------------*/

Expr
    : Oper
    | "-"       Oper                                  { $$ = eNeg($2);     }
    | Oper "+"  Oper                                  { $$ = eAdd($1, $3); }
    | Oper "-"  Oper                                  { $$ = eSub($1, $3); }
    | Oper "*"  Oper                                  { $$ = eMul($1, $3); }
    | Oper "/"  Oper                                  { $$ = eDiv($1, $3); }
    | Oper "\\" Oper                                  { $$ = eRem($1, $3); }
    | Oper "="  Oper                                  { $$ = eEq($1, $3);  }
    | Oper "<"  Oper                                  { $$ = eLt($1, $3);  }
    | Oper ">"  Oper                                  { $$ = eGt($1, $3);  }
    | Oper "<>" Oper                                  { $$ = eNeq($1, $3); }
    | Oper "<=" Oper                                  { $$ = eLeq($1, $3); }
    | Oper ">=" Oper                                  { $$ = eGeq($1, $3); }
    | Oper "&&" Oper                                  { $$ = eAnd($1, $3); }
    | Oper "||" Oper                                  { $$ = eOr($1, $3);  }
    | NOT       Oper                                  { $$ = eNot($2);     }
    ;

Oper
    : INT_LITERAL                                     { $$ = new Expr($1); }
    | TRUE                                          { $$ = new Expr(true); }
    | FALSE                                        { $$ = new Expr(false); }
    | IDENTIFIER                                  { $$ = new Expr(M($1)); }
    | "(" Expr ")"                                              { $$ = $2; }
    ;

/** -------------------- Lists ----------------------------------------------*/

L1_Choice
    : Choice                                      { $$ = std::vector{$1}; }
    | L1_Choice DEDENT Choice             { $$ = M($1); $$.push_back($3); }
    ;

L1_Process
    : Process                                     { $$ = std::vector{$1}; }
    | L1_Process Process                  { $$ = M($1); $$.push_back($2); }
    ;

L1_Alternative
    : Alternative                                 { $$ = std::vector{$1}; }
    | L1_Alternative DEDENT Alternative           { $$ = std::vector{$1}; }
    ;

L1_Declaration
    : Declaration                                 { $$ = std::vector{$1}; }
    | L1_Declaration Declaration          { $$ = M($1); $$.push_back($2); }

%%

/** -----------------------------------------------------------------------*/
```

8.2 Appendix B

This appendix contains code for the constexpr-map that C++17 enables. It features greatly improved speed over regular hash maps.

```
template <typename Key, typename Value, std::size_t Size>
struct Map {
    std::array<std::pair<Key, Value>, Size> data;

    [[nodiscard]] constexpr Value at(const Key key) const {
        const auto it =
            std::find_if(begin(data), end(data),
                [&key](const auto &v) { return v.first == key; });
        if (it != end(data)) {
            return it->second;
        } else {
            throw std::range_error("Key not found");
        }
    }

    [[nodiscard]] constexpr bool contains(const Key key) const {
        const auto it =
            std::find_if(begin(data), end(data),
                [&key](const auto &v) { return v.first == key; });
        return it != end(data);
    }

    [[nodiscard]] constexpr std::size_t size() const {
        return data.size();
    }
};
```

Example of use:

```
inline constexpr auto kInstrs = Map<std::string_view, u8, 16>{{
std::array<std::pair<std::string_view, u8>, 16> {{
    {"pfix"sv,   0x0}, // Prefix.
    {"nfix"sv,   0x1}, // Negative prefix.
    {"opr"sv,    0x2}, // Operate.
    {"ldl"sv,    0x3}, // Load local.
    ...
  }}
}};
```

9 Bibliography

References

[1] Chris Jesshope. Parallel processing, the transputer and the future. *Microprocessors and Microsystems*, 13(1):33–37, 1989.

[2] David May and Richard Taylor. Occam - an overview. *Microprocessors and Microsystems*, 8(2):73–79, 1984.

[3] Inmos. occam 2.1 reference manual. `http://www.transputer.net/obooks/occref/oc21refman.pdf`.

[4] Michael L. Scott. 3 - names, scopes, and bindings. In Michael L. Scott, editor, *Programming Language Pragmatics (Third Edition)*, pages 111–173. Morgan Kaufmann, Boston, third edition edition, 2009.

[5] Jean-Philippe Vasseur and Adam Dunkels. Chapter 11 - smart object hardware and software. In Jean-Philippe Vasseur and Adam Dunkels, editors, *Interconnecting Smart Objects with IP*, pages 119–145. Morgan Kaufmann, Boston, 2010.

[6] Keith D. Cooper and Linda Torczon. Chapter 6 - the procedure abstraction. In Keith D. Cooper and Linda Torczon, editors, *Engineering a Compiler (Second Edition)*, pages 269–330. Morgan Kaufmann, Boston, second edition edition, 2012.

[7] M. Elizabeth and C. Hull. Occam—a programming language for multiprocessor systems. *Computer Languages*, 12(1):27–37, 1987.

[8] M.Elizabeth C. Hull and Adib Zarea-Aliabadi. Real-time system implementation — the transputer and occam alternative. *Microprocessing and Microprogramming*, 26(2):77–84, 1989.

[9] Inmos. Communicating processes and occam. `http://www.transputer.net/tn/20/tn20.html`.

[10] Inmos. Transputer instruction set. `http://www.cs.ox.ac.uk/people/geraint.jones/publications/book/inmos/inmoscwg.pdf`.

[11] re2c. `https://re2c.org/index.html`.

[12] bison. `https://www.gnu.org/software/bison/manual/bison.html`.