Communicating Process Architectures 2015
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2015

Guppy: Process-Oriented Programming on Embedded Devices

Frederick R.M. BARNES

School of Computing, University of Kent, Canterbury, Kent UK. CT2 7NF.

F.R.M.Barnes@kent.ac.uk

Abstract. Guppy is a new and experimental process-oriented programming language, taking much inspiration (and some code-base) from the existing occam-π language. This paper reports on a variety of aspects related to this, specifically language, compiler and run-time system development, enabling Guppy programs to run on desktop and embedded systems. A native code-generation approach is taken, using C as the intermediate language, and with stack-space requirements determined at compile-time.

Keywords. Guppy, Concurrency, Process-oriented, LEGO® MINDSTORMS® EV3

Introduction

This paper reports on a thread of experimentation in a new process-oriented language named Guppy, based on and influenced by the occam-π/Transputer/CSP software, systems and models [1,2,3]. Previous work has successfully seen occam-π process-oriented programs running on a variety of desktop and embedded systems, employing a range of techniques from interpretation to native code generation.

Of particular interest to the work here is the LEGO® Group's MINDSTORMS® third generation of programmable brick, the EV3. Occam-π programs have been run successfully on the first two generations [4,5], the LEGO® MINDSTORMS® RCX and NXT, using the Transterpreter [6] — a virtual Transputer emulator[1].

Following the trend of process-oriented programming for embedded devices, we are experimenting with Guppy using the LEGO® EV3 as a target platform, in addition to a standard Linux desktop. In order to handle target code-generation in a platform independent way, the (experimental) compiler generates C code, that is further compiled or cross-compiled for the particular target platform. This is then linked with a suitable run-time system that manages process scheduling and communication.

This paper reports on this ongoing work, covering language design, compiler and run-time implementation. The aims of the work are two-fold: firstly, to explore concurrent language design, to produce a successor to the occam-π language for a range of platforms; and secondly, to support process-oriented programming on the LEGO® EV3 platform. The latter is a more easily reached goal. The paper is arranged as follows: Section 1 gives some background on the technologies involved, specifically the existing occam-π toolchains and the LEGO® EV3. The Guppy language is described in Section 2, compiled down to C code using the "nocc" compiler framework, described in Section 3. The minimal run-time implemented for the EV3 platform is described in section 4. Section 5 examines a specific example: imple-

[1]The Transterpreter has greatly improved the accessibility of occam-π programming, having been ported to a variety of devices (embedded / robotics and desktop). Porting it to the EV3 would not be a significant challenge.

menting the "dining philosophers", followed by conclusions and a discussion of future work in Section 6.

1. Background and Motivation

In creating a successor to occam-π, we wish to preserve all of the positive features. Specifically at run-time, low memory overheads for concurrent processes and efficient scheduling and synchronisation (communication) between these. Having these properties enables systems with large numbers of concurrent processes (tens, thousands, millions) and complex dynamic interactions to be created and studied — an avenue that we and others are keen to explore.

The *lightweight* nature of concurrency in occam-π stems from two sources. Firstly, the Transputer/occam execution model [2,7], that statically allocates and *packs* together process memories (workspaces). Secondly, the efficient multicore CCSP run-time system [8]. The *synchronous* nature of occam-π, that stems from Hoare's CSP [3], makes process management and communication relatively simple to implement — as was once implemented on the Transputer in hardware/microcode[2].

1.1. The KRoC Toolchain

Figure 1 shows the typical KRoC toolchain for turning an occam-π source file '`prog.occ`' into an executable '`a.out`' for a standard 32-bit Linux platform. Also shown is an occam-π*library*, that is '`#USE`'d by the program. The occam-π compiler, `occ21`, is a heavily modified version of the occam 2.1 compiler developed at Inmos in the early 1990s. This has been extended over the years, incorporating a variety of new (and at times, experimental and not entirely stable) language features; among them the various dynamic extensions inspired by Milner's π-calculus [9]. A good overview of the occam-π language enhancements is given in [1].

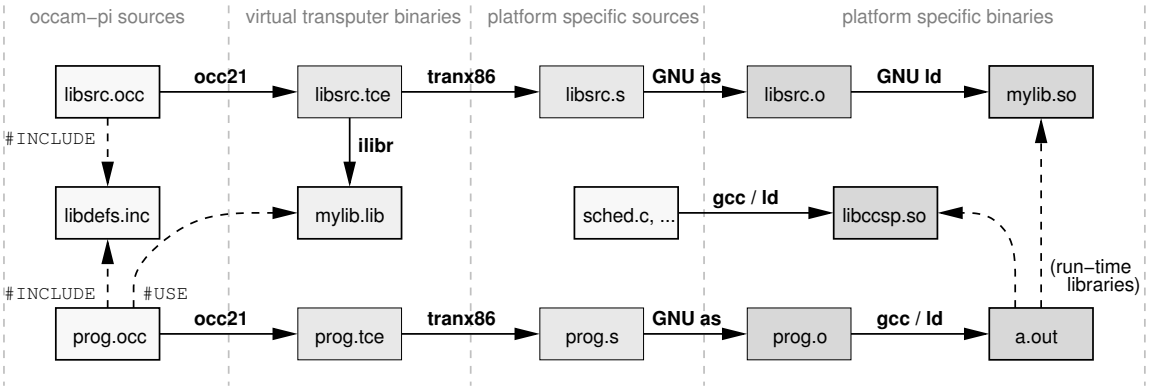


Figure 1. The KRoC occam-π toolchain.

The output of the occam-π compiler is a form of Transputer binary, *Extended Transputer Code* [10]. Whilst this has been heavily enhanced to support various new language features (provided largely by the run-time system), the machine `occ21` generates code for is still fundamentally a Transputer: a stack-based machine, with 3-level deep integer and floating-point stacks, and a *workspace pointer* that acts as a moveable process descriptor (similar to a stack pointer). Communication channels are single words in memory, that are either null[3] or hold a

[2]There are constructs in CSP that are difficult to implement effectively in a language or run-time, such as resolving multi-way synchronisations. Occam (and occam-π) restrict slightly what is available in CSP terms to avoid these.

[3]On the Transputer, this *null* value was really 0x80 00 00 00, or the most negative signed 32-bit integer.

process descriptor of a process blocked on communication. The minimum memory footprint of a non-trivial process is just enough to hold its scheduling state: between 3 and 6 words. With local variables, internal concurrency structures and parameters, a typical process memory footprint is at least 32 words (128 bytes). Even with large numbers of concurrent processes *and* communication channels, memory is not usually an issue — this has encouraged its use on small-memory devices using the Transterpreter (Section 1.2). The binary (machine) code generated is also highly space-efficient, with most instructions requiring 1 or 2 bytes. Alongside instructions, the generated code (generally referred to as *Transputer bytecode*) includes information such as process memory requirements and debugging records (source names and line numbers).

One issue that exists with the current system is the fact that the compiler code-generates for a 3-deep evaluation stack: any complex expressions requiring more than 3 words (registers) to evaluate are spilled to memory. Modern machines typically have many more registers available, that if used effectively would reduce the memory footprint *and* execution efficiency even further — unfortunately changing the 3-deep stack in the compiler's code-generator would require *extensive* re-engineering.

For standard x86 machines, `tranx86` takes the Transputer bytecode and translates it into Intel x86 (32-bit) assembler. The evaluation stack is mapped onto general-purpose registers, along with the other run-time registers (workspace pointer, etc.). Instructions for channel communication, process scheduling and similar are mapped to calls into the run-time system (CCSP). The translator performs some small-scale *peephole* style optimisations, but little beyond this. These are assembled down to native (x86) binaries and linked with libraries and the run-time system to produce an executable.

The CCSP run-time system is mostly coded in C, with some portions of in-line assembly to optimise switching in and out of the run-time system. At the very least, the run-time system has to provide for process creation, shut-down and scheduling, channel communication (including alternatives/choice) and any other features required.

1.2. The Transterpreter

The Transterpreter was created to provide a *portable* way of running occam-π programs [6]. It is essentially an emulator, written mostly in C, that interprets the Transputer bytecode. A separate program, `slinker` or `plinker` collects up the Transputer bytecode files (including libraries), resolving cross-references, to produce a single binary image for emulation. The resulting file is, in many cases, small (a few kilobytes). This makes it highly suited to concurrent programming on small-memory devices such as the LEGO® RCX (an 8/16-bit microcontroller with 32 KiB SRAM) [5], the LEGO® NXT (a 32-bit ARM microcontroller with 64 KiB SRAM and 256 KiB FLASH) and the Arduino (an 8/16-bit microcontroller with up to 8 KiB SRAM and 256 KiB FLASH). In these embedded environments, the Transterpreter generally replaces any default firmware on the device.

Even though emulation incurs a performance penalty, this has not been an issue for typical LEGO® robotics applications — where response times of tens of milliseconds are acceptable. One place this has been an issue is when programming the Arduino's serial communication ports: the device runs at 16 MHz — keeping up-to-speed with serial data transmission or reception, from the emulated occam-π program, can be problematic.

In addition to embedded environments, the Transterpreter provides a convenient and portable way to run occam-π programs on a whole range of systems — e.g. Microsoft Windows, Solaris (Oracle), Raspberry Pi, in general any Linux system. This has contributed to the popularity of occam-π, and for the LEGO® platforms in particular, provides a straightforward hands-on (tangible) introduction to concurrent programming. Whilst it would be relatively straightforward to port the Transterpreter to the LEGO® EV3, that would still expect

Transputer bytecode as input, where we are keen to explore more efficient code-generation routes.

1.3. The LEGO® EV3

The LEGO® MINDSTORMS® EV3 is a significantly more powerful device than its predecessors, based around an ARM9 core running at 300 MHz, with 64 MiB RAM and 16 MiB FLASH. More significantly, the EV3 runs a modified version of the Linux kernel (that includes device-driver support to interface with the various sensor, motor ports and the small display/buttons). On top of this runs a single application that provides the EV3 user interface, including communication with the host (over USB using a proprietary, but well-documented, protocol). The embedded device itself contains a Micro-SD slot, USB host interface (intended for a WiFi dongle) and Bluetooth, that are handled by Linux in the normal way.

The base kernel build for the particular ARM, and standard GNU/Linux OS layer tools are provided via "CodeSourcery" from Mentor Graphics — that also provides the cross-compile environment for a standard Linux desktop. This can be used to create custom images on an SD card, from which the EV3 will boot when present and correctly configured. Other open-source efforts have also ported a more recent kernel and further customised it for the EV3[4] and is the kernel/OS we are using.

2. Experimenting with Process-Oriented Languages

This section presents an overview of the Guppy language, a potential successor to the existing occam-π language. The concurrency semantics are the same as occam-π: based on the CSP concept of synchronising/communicating concurrent processes, augmented with features of *dynamics* and *mobility*, implemented using the existing CCSP run-time system. It should be noted that the following sections describe the Guppy language in its *current* state, that may still be subject to changes in the future.

2.1. General Syntax

The syntax of Guppy is similar in many respects to occam-π, with some notable exceptions. Firstly, lowercase keywords — syntax highlighting editors are common enough that the need to distinguish language keywords using uppercase is perhaps redundant. There is some argument for keeping uppercase keywords (it certainly has some elegance) but the common preference is for lowercase syntax. Secondly, indentation-based layout. In occam-π (and previously occam) this was fixed at 2 spaces; languages such as Python, also indentation-based, are more flexible, allowing the programmer to choose whatever spacing he/she prefers. This is the approach we have taken for Guppy, even though most code shown here uses 2 or 4 spaces. Comments in Guppy start with the hash character '`#`' and continue to the end of the line.

2.2. Primitive, Structured and Named Processes

Given the CSP semantics for processes, the primitive and structured processes of Guppy are semantically identical to those of occam-π. The two most basic primitive processes are '`skip`' and '`stop`' — corresponding to their CSP equivalents: successful termination and deadlock (or run-time error). The structured processes are '`seq`', '`par`', '`alt`', '`if`' and '`while`'. As per occam, the first 4 of these may be *replicated*. The '`alt`' and '`if`' structures contain *guarded* processes.

[4]This kernel (ev3dev) can be found on Github and at http://www.ev3dev.org/.

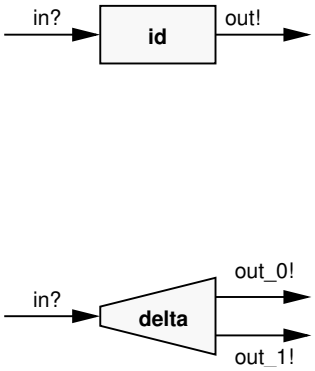
For convenience we allow a shortened version of '`if`' that takes an expression immediately after the keyword and contains a single indented process (executed if the expression evaluates to true). Another convenience, though some have argued against it, is an optional '`seq`'. This is primarily to appease programmers more used to languages like C and Java, where sequential execution is assumed.

Named processes (procedures) are introduced with the '`define`' keyword — as are other language artefacts such as structured types and protocols. This is followed by the procedure name and a list of zero or more parameters. Procedures are also permitted to return zero or more *values*. This is a slight difference from occam-π, whereby procedures (assumed to be side-effecting) can only return values through reference parameters. As an example of the language syntax, the following shows Guppy definitions for some well-known processes:

```
define id (chan(int) in?, out!)
  while true
    int n
    seq
      in ? n
      out ! n
end

define delta (chan(int) in?, out_0!, out_1!)
  while true
    int v

    in ? v
    par
      out_0 ! v
      out_1 ! v
end
```



Both procedure definitions finish with the '`end`' keyword. This is a convenience and included to make the ends of such definitions more obvious (and visually neater). The definition for '`id`' is straightforward and looks remarkably similar to the occam-π version of this process. The definition for '`delta`' illustrates the optional '`seq`' as it assumed that the input and the parallel outputs happen sequentially, and both with the variable '`v`' in scope.

2.3. Variables and Types

As can be seen in the above code, variables are declared in the usual way: a *type* followed by a comma-separated list of *names*. Although variables must still be declared *before* the process that uses them, the compiler will assume sequential execution of processes (statements) immediately following a declaration. This permits declarations at any point inside sequential code, that remain in-scope until the end of the block they appeared in.

The primitive types in Guppy are as they are in occam-π, but also including an 8-bit signed integer type, explicitly unsigned integer types (`uint`) and types for strings and characters. These are summarised in Table 1.

While only partially supported, types in Guppy may also be parameterised. The earlier code shows this in its channel parameters, that are of type '`chan(int)`'. Channels are a slightly special case, since a single channel has two logical ends (the inputting end indicated with '`?`', and the outputting end with '`!`'). As with occam, there are no explicit *pointer* types, and no uncontrolled aliasing.

Table 1. Primitive types in Guppy.

Type	Description	Type	Description
`bool`	boolean `true` or `false`	`uint8`	unsigned 8-bit integer
`byte`	8-bit byte	`uint16`	unsigned 16-bit integer
`int`	signed integer (32-bit default)	`uint32`	unsigned 16-bit integer
`int8`	signed 8-bit integer	`uint64`	unsigned 16-bit integer
`int16`	signed 16-bit integer	`real`	floating-point number (32-bit default)
`int32`	signed 32-bit integer	`real32`	32-bit (single) floating-point number
`int64`	signed 64-bit integer	`real64`	64-bit (double) floating-point number
`uint`	unsigned integer (32-bit default)	`string`	string type (Unicode aware)
`char`	character (Unicode aware)		

2.4. Expressions, Communication and Assignment

Expressions in Guppy, as in occam-π, are required to be side-effect free. Certain exceptions are allowed however, for programmer convenience. The semantics given to these makes explicit what the operational behaviour is as far as expression evaluation is concerned (i.e. side-effecting expressions are only permitted in certain and well-defined situations). In a similar way to occam-π, there is no operator precedence[5], so explicit bracketing is typically required. Unlike occam-π however, associativity is specified for arithmetic operators, so expressions such as '`a + b - c`' are valid (and evaluated left-to-right).

As the earlier code example shows, the syntax for channel communication is essentially the same as in occam and occam-π. That is, a channel name, followed by '`?`' for input or '`!`' for output, and a list of variables or expressions respectively. Assignment is also similar (semantically) to occam-π, and multiple assignment is permitted.

The occam-π language has a very clear distinction about the difference between a *procedure* and a *function* — the latter not permitted to contain side-effects (must be *deterministic*), but in most cases, is written in a very unfamiliar syntax (using a '`VALOF ... RESULT`' expression). The distinction in Guppy is more automatic: procedures are also functions if they *do not* contain side-effects. To be useful, a procedure (or function) has to be able to *return* values. For example:

```
define genseed () -> int
  timer t
  int v

  t ? v                     # read current time
  return (v >> 2) + 1       # ensure 1..maxint and return
end
```

This is a *procedure* that returns a single integer value. Reading the current time (as this code does with '`t ? v`') is considered a side-effect and thus '`genseed()`' is not a pure function — if it were, it could be optimised away to a constant value (in theory). Alternatively, a function that returns *two* integers can be defined with:

```
# based on 'minimal' standard described in "Random number generators:
#   Good ones are hard to find", K.P. Park & K.W. Miller (1988),
#   Comm. ACM 31(10), 1192–1201; implemented in occam by David Morse.
#
define random (val int max, seed) -> int, int
  val int magic = 16807
```

[5]An exception, again for convenience, is that the two unary operators (minus and bitwise-not) do have a higher precedence than other operators.

```
  val int period = 2147483647
  val int quot = period / magic
  val int rem = period \ magic

  int hi = seed / quot
  int lo = seed \ quot
  int test = (magic ** lo) -- (rem ** hi)
  int nseed

  if
    (test > 0)
      nseed = test
    true
      nseed = test ++ period

  return (nseed \ max), nseed
end
```

This is a pure function, in the sense that the mapping from inputs to outputs is fixed. Regarding arithmetic operators, the intention is for them to be checked for arithmetic overflow (and a run-time error raised in these situations). Three non-overflowing arithmetic operators are provided, '++', '--' and '**'. The various operators are listed in Table 2.

Table 2. Operators in Guppy.

Arithmetic:	
- A	unary minus
A + B	addition
A - B	subtraction
A * B	multiplication
A / B	division
A \ B	remainder
A >>> B	arithmetic shift right
A ++ B	addition (no overflow)
A -- B	subtraction (no overflow)
A ** B	multiplication (no overflow)

Relational:	
A == B	equal to
A != B	not-equal
A < B	less-than
A > B	greater-than
A <= B	less-than or equal
A >= B	greater-than or equal

Bitwise:	
~ X	not
X & Y	and
X | Y	or
X >< Y	exclusive or
X << Y	logical shift left
X >> Y	logical shift right

Boolean:	
! V	not
V && W	and
V || W	or
V >< W	exclusive or

Other:	
V -> A : B	if-then-else
P ; Q	sequence
P ||| Q	parallel

The only operator precedence is for unary minus and bitwise not. Arithmetic, bitwise and boolean operators are left-associative, with '+' and '-' being considered the same (as well as '++' and '--'). Other combinations of operators must be suitably bracketed.

For convenience, Guppy permits a number of side-effecting and commonly seen (in Java and C) single-process expressions. For example:

```
int x = 0, y = 0

seq i = 1 for 100
  y += i
```

```
x++
```

Shortcuts such as '`+=`' and '`++`' (as a postfix operator) are merely syntactic replacements for the complete assignments, i.e. '`y = y + i`' and '`x = x + 1`'. They are not in themselves *expressions* (instead assignment processes) so cannot be used as part of another expression. For instance, the assignment '`x = y++`' would be illegal.

3. New Compilers for New Languages

To explore and experiment with concurrent languages, and ultimately to develop a viable successor to occam-π, a new compiler framework is required. Specifically one that allows us to experiment easily: e.g. to add a new feature to the language, with its particular type, (parallel) usage checks and code-generation. For our experiments with Guppy, and other less well-developed languages, we are using the 'nocc' compiler framework[6].

Nocc was originally written to be a drop-in replacement for the existing '`occ21`' occam-π compiler — generating Transputer bytecode from occam-π sources. This particular aspect of the compiler is still substantially incomplete. Instead of being hard-coded to a particular language or target architecture, nocc provides a general compiler framework in which various source languages and/or target architectures can be supported. Part of the reason for this is to allow experimentation with mixed-language sources, or multiple targets in code-generation (e.g. CPU and GPU, DSP, etc.).

3.1. Do We Really Need Another Compiler?

This question has been raised several times, and related, if we do need a new compiler, in what language should it be written? Whilst the existing (and heavily modified) '`occ21`' compiler has sufficed, it is not an easy code-base to work with and is fundamentally the barrier to further work on occam-π (including bug-fixing). Over the years there have been various efforts to create new occam compilers, with varying degrees of success, and with many undocumented. One of the more successful is *Tock* [11], written in the Haskell [12] programming language. A longstanding alternative to the KRoC toolchain, though less developed, is provided by SPOC [13], where a mostly C compiler translates occam sources into C, with a hard-coded deterministic scheduler. Attempts have also been made to support occam in existing compiler frameworks, though these have met with limited success — the strict parallel-usage and aliasing checks required can be an issue.

Being (mostly) abstract tree manipulation machines, the motivation for writing compilers in high-level (and particularly functional) languages is compelling — tree-transformations are much easier to express in high-level pattern-matching languages. But there can be drawbacks: a large memory footprint from frequent object creation and less frequent garbage collection; execution overheads of interpretation (if not compiled down to native code or *just-in-time* compiled); a programming environment limited to what the language provides (particularly regarding libraries); and an extra hurdle for user adoption. Nocc has been written, for the most part, in plain C. It makes extensive use of function-pointer containing structures, that allow for *object-oriented* style behaviours (e.g. function overriding) and *aspect-oriented* (e.g. code insertion) [14] ones. The advantages of using C include cross-platform support, extensive optimisations and meaningful debugging. The disadvantages are principally a large code-base, and general code-bulk, plus a steep learning-curve to working on the compiler — though, we suspect, less impenetrable than the existing occam-π compiler.

[6] Available on Github at https://github.com/concurrency/nocc/.

3.2. Compiler Structure

Nocc, as a compiler framework, has a fairly complex start-up procedure. The various language and target specific parts of the compiler first 'register' themselves. Language definitions include an amount of metadata, including the specific language version, what file-name suffixes they associate with, and details of the language's current maintainer. Other initialisation activities within the compiler include setting up the cryptographic framework (for digital signing of compiler output) and setting up the file-system abstraction (that allows, conveniently or dangerously, program sources to refer to other files using a URL).

Once initialised, the specified source file(s) are opened and processed by a language-specific *lexer* (lexical analyser). This, as in other compilers, results in a stream of *tokens*. The parser framework provided in nocc is complex (and perhaps over-engineered) but ultimately supports a type of recursive descent parser. The grammar for a particular language is generally not built into the compiler, but instead provided in a specific *language definition* file. When the compiler initialises for a particular language, these definitions are loaded and incorporated (along with compiled-in information about parse-tree node types and their structure). This is examined in more detail in section 3.3.

Once a parse-tree has been successfully generated, a variety of tree-transformation passes take place. These are broadly divided into two categories: front-end passes, that are focused on language-specific transforms and have little or no knowledge of the particular target; and back-end passes, that have full knowledge of the target. A subset of these are shown in Figure 2. Languages and targets themselves can add extra passes to the compiler, as well as attach code to existing ones.

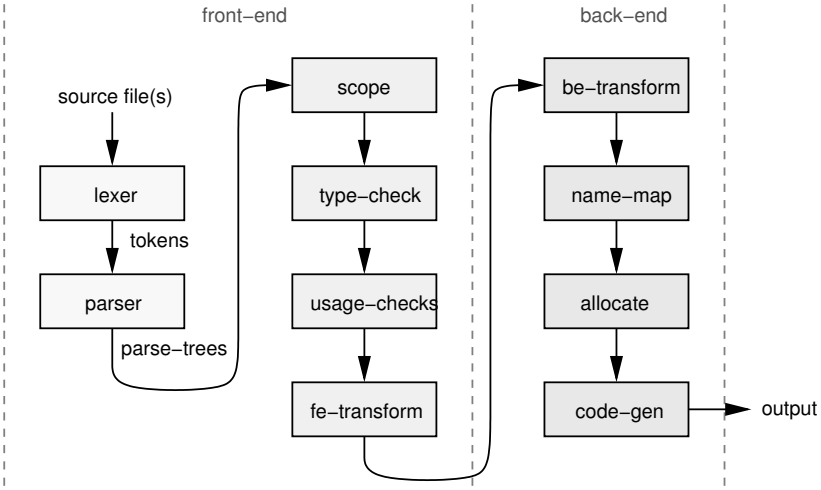


Figure 2. Nocc outline compiler passes.

Throughout the compiler, parse-tree *nodes* have the same C type, '`tnode_t`'. This is a simple structure in practice:

```
typedef struct {
    ntdef_t   *tag;              // node tag definition
    srclocn_t *org;              // source location

    DYNARRAY (void *, items);    // subtrees, names, hooks
    DYNARRAY (void *, chooks);   // compiler hooks
} tnode_t;
```

The '`DYNARRAY`' macros are used to implement resizeable arrays in C (the macro expands into 3 variable declarations: *cur* and *max* counters, and a pointer to the array). The '`tag`' is a unique

pointer to a language or target provided structure that defines a particular *node tag*. These 'tag's point to more general *node type* structures that describe the arrangement of '`items`' (as the number of subtrees, names and 'hook' pointers contained within). A variety of functions and macros within the compiler provide access to these (e.g. '`tnode_nthsubof(...)`'). The 'hook' pointers allow language and target-specific code within the compiler to attach arbitrary structures to parse-tree nodes.

The '`org`' component of a tree node provides a reference back to a source location — primarily for the purpose of reporting errors and warnings. The last, '`chooks`', is an array of *compiler-hooks* that are used to attach compiler-specific data structures to tree nodes (e.g. related to metadata generation or parallel-usage checks).

3.3. Language Definition and Parsing

Language definitions, that are primarily the concern of the compiler front-end, are presented to the compiler as a collection of *front-end units*. Each unit is typically responsible for a particular aspect of an input language, e.g. process constructors in Guppy. The initialisation code for these creates the various parse-tree node *types* and *tags* within the compiler, attaching locally defined (or generic) functions to them. These function calls are attached to either *compiler-operations*, that implement specific handling for each pass within the compiler (e.g. "scope-in"), or *language-operations*, that provide language-specific functionality within the compiler (e.g. "get-type"). The way this is implemented also allows front-end units to interfere with the way processing on other nodes is performed — usually to intercept a specific compiler pass on a particular node type or tag, with the option of further calling the displaced function.

As an example, the following shows a fragment of code from the Guppy process-constructor handling file ('`frontend/guppy_cnode.c`'), that is responsible for handling the '`seq`' and '`par`' (and '`alt`') constructs:

```
int i;
tndef_t *tnd;
compops_t *cops;

i = -1;
tnd = tnode_newnodetype ("guppy:cnode", &i, 2, 0, 0, TNF_LONGPROC);
cops = tnode_newcompops ();
tnode_setcompop (cops, "prescope", 2, COMPOPTYPE (g_prescope_cnode));
tnode_setcompop (copy, "declify", 2, COMPOPTYPE (g_declify_cnode));
...
tnd->ops = cops;

i = -1;
gup.tag_SEQ = tnode_newnodetag ("SEQ", &i, tnd, NTF_IND_PROC_LIST);
i = -1;
gup.tag_PAR = tnode_newnodetag ("PAR", &i, tnd, NTF_IND_PROC_LIST);
```

The call on line 6 to '`tnode_newnodetype`' creates a new parse-tree node type within the compiler, specifying the number of sub-trees (2), names (0) and hooks (0). The flag '`TNF_LONGPROC`' is used to instruct the higher-level parser to expect an indented list of processes next. The calls on lines 8 and 9 attach local functions to node-type definition, that will be called during the appropriate compiler pass (e.g. pre-scoping). The last few lines of code create the actual parse-tree node tags, that are stored away in a language-specific global structure ('`gup`').

In addition to the various front-end units, languages must provide a lexer and parser. The lexers for all languages currently supported are individually coded within the compiler. Nocc provides an amount of support for keyword and symbol matching however.

The parsers for nearly all input languages in nocc utilise a generic DFA parser within the compiler. In the majority of cases, a specific language-definition file describes named DFA state transition rules, with reductions described as code for an abstract stack machine (that can manipulate a token-stack, node-stack and create new parse-tree nodes, amongst other things). These files are read, and parsed, as part of the language-specific initialisation. Although this will happen each time the compiler is run, it is sufficiently fast as not to cause any perceivable delay.

The main reason for implementing parsing this way is to allow for easy experimentation with language syntax and structure. It also gives language implementations a great deal of control over the way parsing is handled (at the expense of having to describe a language as DFA state transitions). Language definitions can also describe parsing structures in BNF where that is more convenient, relevant in particular to lists of things.

As an example, the following shows a fragment of Guppy's language definition file (‘`addons/guppy.ldef`’) that is responsible for parsing *channel-output*:

```
.SECTION "guppy-io"

.GRULE  "gup:outputreduce" "SN0N+N+V0C[OUTPUT]3R-"
.TABLE  "guppy:output ::= [ 0 guppy:name 1 ] [ 1 @@! 2 ]"
                "[ 2 guppy:expr 3 ] [ 3 {<gup:outputreduce>} -* ]"
```

The particular DFA rule for parsing an output (‘`guppy:output`’) is described as a *name* followed by the ‘`!`’ symbol and an *expression*. The reduction rule ‘`gup:outputreduce`’ combines the parsed name and expression into a new ‘`OUTPUT`’ node and makes this the ‘result’ of the output parse. For more complex parsing or reductions, language definitions can refer to named functions within the compiler (that are registered on language initialisation).

3.4. Code Generation for C and CCSP

After the last step in the front-end of the compiler, the parse-tree is assumed to be correct (i.e. represents a valid program with respect to types, parallel-usage, aliasing, etc.). By this point, a specific code-generation target should be known. Languages specify by default what architecture to target, though this can be changed. Importantly, the linkage between an input language and a particular target architecture for code-generation is handled on the language side, meaning languages (specifically front-end units) need to be aware of target specifics regarding back-end processing, but targets need not be aware of their source languages.

The back-end (target) used for Guppy, named “CCCSP”, generates C code to interface with the existing CCSP run-time system [8] using a version of the CIF (“C interface”) API. The run-time (via the API) is expected to handle all aspects of process scheduling and communication (including support for event choice through the ‘`alt`’ language construct). The API calls themselves are similar in name and structure to the original Transputer C API, except that all C *processes* generated by the compiler (nocc) carry around an explicit *workspace* parameter that points to the concurrent process context, separate from the C (and architectural) stack-pointer.

The CCCSP back-end supports *two* architecture sub-targets. One specifically for the standard 32-bit Linux environment, using the existing CCSP scheduler, and one specifically for the LEGO® EV3. The overall code-generation for a C target is not particularly complex — more or less a case of translating Guppy procedures into C functions that call into the run-time API where needed. Certain parts of the language, the ‘`par`’ construct in particular, have a non-trivial equivalent in C — i.e. allocating and initialising the various sub-processes, scheduling them, then waiting for them to terminate.

3.5. Stack Sizing for C Processes

One issue associated with fine-grained concurrent programming in C, in general, is the issue of unbounded (or unknown) stack sizes. Stacks are expected to be contiguous blocks of (virtual) memory, allocated at the point the process (or thread) is created, and used for local variables and the call stack. In ordinary threaded C programming this is not generally an issue: virtual memory is large (especially on 64-bit platforms) and the number of threads is small (< 100). With user-level (lightweight) threads, scheduled on single cores or across multiple cores (with multiple run-time threads scheduling these), stack-sizing is generally left to the programmer (and usually overly generous — e.g. plenty of room for future dependent library updates).

Generating code for the CIF API, the CCCSP back-end still needs to specify the stack-size required for processes and allocate this. Fortunately, recent versions of '`gcc`' (since version 4.6, March 2011) include an option, '`-fstack-usage`', that causes the compiler to emit stack-usage information to a `.su` file. This is done on a per-function basis, where the resulting output looks something like:

```
armccsp_if.h:209:21:MAlloc            40      static
test_g62.c:68:27:GuppyStringInit      24      static
test_g62.c:388:6:gproc_test_g62       24      static
test_g62.c:394:6:gproc_guppy_main     456     static
test_g62.c:415:5:main                 24      static
```

The first column identifies the particular function and where it is defined; the second column gives the required stack-frame size, in bytes; and the third column a usage qualifier (for our purposes we are only considering those "static" or "dynamic,bounded", in both cases indicating the function stack size's upper limit). The stack size given *does not* include parameters, nor the stack required by any other functions called from this one. A main function that calls '`printf()`', e.g. hello world, will have a small fixed stack requirement — 32 bytes, even though '`printf`' may require much more.

Once the resulting C file has been compiled (with incorrect/incomplete allocation within the generated code) the generated `.su` file is read back in, and the allocation and code-generation passes re-run. This is illustrated in Figure 3. In addition to the particular source's `.su` file, other stack-usage information is read in: those of language-specific libraries, and a separate file ('`api-call-chain-ev3`') that specifies the call-chain for kernel (and library) API calls. The '`cc-compile`' pass will also (re-)compile the language-specific libraries (API and kernel interface) as needed.

The '`cc-sfi`' pass calculates the stack-size required (in total) for each function generated by the compiler (nocc). This is, at most, the stack size required for the function, plus the maximum of the stack size for any functions called, including interaction with the run-time API. Once calculated, the previously reserved memories for parallel process workspaces and stacks are re-allocated, the C code re-generated and re-compiled.

Where parallel processes are involved, the compiler will aim to allocate the stacks for these statically (that it can do in most cases) by declaring explicit arrays to hold these, becoming part of the containing process's stack. As an example of this, and of the C code generated by nocc, the parallel delta shown on page 5 (that, each cycle, creates and shuts-down two parallel subprocesses) is coded as follows:

```
void gproc_tmp_1_086fa280 (Workspace wptr3)
{
    Channel* out0 = ProcGetParam (wptr3, 0, Channel*);
    int* v = ProcGetParam (wptr3, 1, int*);
    ChanOut (wptr3, out0, v, 4);
```

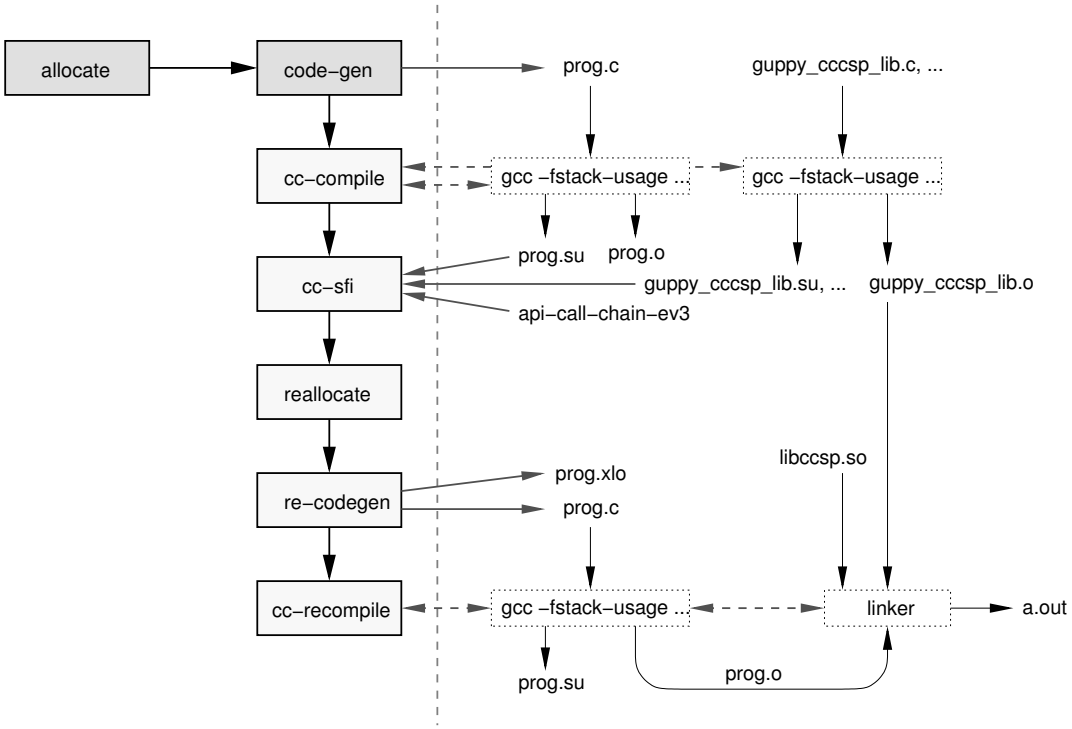


Figure 3. CCCSP back-end passes for stack sizing and the standard library.

```
}

void gproc_tmp_2_087094d0 (Workspace wptr4)
{
    Channel* out1 = ProcGetParam (wptr4, 0, Channel*);
    int* v = ProcGetParam (wptr4, 1, int*);
    ChanOut (wptr4, out1, v, 4);
}

void gcf_delta (Workspace wptr5, Channel* in,
                                 Channel* out0, Channel* out1)
{
    while (1) { {
        int v;
        ChanIn (wptr5, in, &v, 4);
        {
            Workspace wptr6;
            Workspace wptr7;
            word ws8[WORKSPACE_SIZE(2,25)];
            word ws9[WORKSPACE_SIZE(2,25)];

            wptr6 = LightProcInit (wptr5, ws8, 2, 25);
            ProcParam (wptr5, wptr6, 0, out0);
            ProcParam (wptr5, wptr6, 1, &v);
            wptr7 = LightProcInit (wptr5, ws9, 2, 25);
            ProcParam (wptr5, wptr7, 0, out1);
            ProcParam (wptr5, wptr7, 1, &v);
            ProcPar (wptr5, 2, wptr6, gproc_tmp_1_086fa280,
                               wptr7, gproc_tmp_2_087094d0);
        }
    } }
}
```

The first two (slightly oddly named) functions represent the two parallel sub-processes. As with all launched parallel processes, the function parameters must be collected via the workspace (using '`ProcGetParam`'). Whilst this makes parameter handling a bit more cumbersome (as they must be set and gotten explicitly) it avoids the need to handle anything other than a single parameter when scheduling a parallel process for the first time. Different architectures and compilers handle parameter passing in different ways, usually using both registers and the stack, that can get awkward — though manageable — when interfacing from assembly.

4. Lightweight Scheduling on the LEGO® EV3

The standard CCSP run-time system currently only supports Intel IA32 (x86) based architectures, although other architectures were supported (to varying degrees) prior to the integration of multicore scheduling support. This is largely due to the assembly code within it, part of which is used to interface occam-π with the run-time, and part of which is contained in the various algorithms that implement the multi-threaded (multicore) work-stealing scheduler, channel communication, etc. Porting this to the ARM9 architecture (used in the LEGO® EV3) would be possible, but since this is a single-core device, perhaps excessive. There have been previous CSP-style run-time systems developed for the StrongARM [15], but these are comparatively old (1998) — and for a quite different ARM microprocessor.

The specific scheduler developed for the EV3's ARM9 processor has been engineered for *simplicity* (more than execution efficiency in some aspects). The current run-time, implemented almost entirely in C with some inline assembler (for switching between the Guppy and the run-time), weighs in at a little under 2,000 lines of code[7], but is incomplete — no support for process priority and little optimisation for performance. I.e. the current implementation represents a *worst case* in terms of performance, though it is adequate.

Each process incorporates a *workspace descriptor* that contains the state of the (lightweight) process. For the Transputer, and the derivative engineered into the KRoC occam-π system, this 'descriptor' (scheduler state) is only used as and when needed. For the EV3 run-time, this state is fairly extensive, and contributes to a lot of the memory overhead of processes:

```
typedef struct TAG_ccsp_pws {
    struct TAG_ccsp_sched *sched;     /* link to scheduler */
    void *stack;                      /* process stack pointer */
    void *raddr;                      /* process return address */

    void *stack_base;                 /* process stack (base) */
    uint32_t stack_size;              /* stack size (in bytes) */
    struct TAG_ccsp_pws *link;        /* link to next process */
    void *pointer;                    /* to data (I/O) or ALT state */
    uint32_t priofinity;
    struct TAG_ccsp_pws *tlink;       /* next link on timer queue */
    int timeout;                      /* timeout (absolute time) */

    void *pbar;                       /* termination barrier */
    void *iproc;                      /* address of initial process */
    int nparams;
    uint32_t params[MAXPARAMS];
} __attribute__ ((packed)) ccsp_pws_t;
```

[7]This includes support for simple round-robin process scheduling, channel communication (including '`alt`'s) and timeouts.

This structure is at present dynamically allocated each time a new process is created (by calling '`LightProcInit`') and with '`MAXPARAMS`' set at 16 requires 29 words of memory — plus the overhead of the run-time kernel's standard memory allocator (2 words typically). For short-lived parallel processes, this is a *significant* overhead, in a situation where the memory for this structure *could* be statically allocated (at the bottom of the process's stack, for instance). For the time being, there is a debugging benefit to having these separate from the process stacks.

4.1. Interfacing with the Run-Time System

The interface from the generated code to the run-time system is via a modified version of the CIF API [16], that itself was based on the Transputer C API [17]. A single C header file '`cif.h`' is exported by the run-time system that provides this API — either through pre-processor macros, '`extern`' declarations for functions in the run-time itself, or as inlined C functions.

Since Guppy processes operate in their own (fixed size and allocated) C stack, entry to the run-time system must switch stacks (this will happen whenever a process is descheduled due to timeout or communication, for instance). This stack switching activity, and call into the run-time itself, is handled within the run-time provided interface. When the run-time schedules a Guppy process, the reverse happens — the stack is switched back to the process's own stack.

To avoid issues relating to the passing of multiple parameters, the majority of run-time calls go via a single entry routine, '`ccsp_entry`'. This expects *three* parameters: a pointer to the workspace of the calling process (that also contains the state needed to resume it), an integer kernel-call identifier, and a pointer to an array of parameters. Thus, when the generated code needs to call into the run-time, the various arguments are collected into an array, and unpacked again on the other side (when in the run-time kernel stack context). A constant table describing the various kernel calls, with pointers to functions that implement them, is defined in the run-time's '`kernel.c`' — along with '`ccsp_entry`', that looks up the call in the table and invokes it. This entry mechanism has some overhead and is a good candidate for later optimisation.

As an example, the *channel output* call '`ChanOut`' is implemented as an inlined function, provided by the run-time (via '`cif.h`'):

```
static inline void ChanOut (Workspace p, Channel *c,
                            const void *ptr, const int bytes)
{
    void *dargs[4] = {(void *)p, (void *)c, (void *)ptr, (void *)bytes};
    int call = CALL_CHANOUT;

    ENTER_KERNEL (p, call, dargs);
}
```

The macro '`ENTER_KERNEL`' contains the inline assembler that switches stacks and performs the actual call into the run-time:

```
#define ENTER_KERNEL(p,c,a) do {
    __asm__ __volatile__ ("                                          \n"
        "   mov     r0, %0                                           \n"
        "   add     r3, r0, #4          @ r3 = &(p->stack)           \n"
        "   push    {lr}                @ save link-register         \n"
        "   str     sp, [r3, #0]        @ p->stack = sp              \n"
        "   add     r3, r0, #8          @ r3 = &(p->raddr)           \n"
        "   adr     r2, 0f                                           \n"
```

```
        "   str     r2, [r3, #0]        @ p->raddr = label 0    \n"
        "                                                       \n"
        "   ldr     r2, [r0, #0]        @ r2 = (p->sched)       \n"
        "   ldr     r3, [r2, #0]        @ r3 = p->sched->stack  \n"
        "   mov     sp, r3              @ switch stacks         \n"
        "   mov     r1, %1                                      \n"
        "   mov     r2, %2                                      \n"
        "   bl      ccsp_entry(PLT)                             \n"
        "0:                                                     \n"
        "                               @ note: when we get back\n"
        "                               @ r0 = process-desc     \n"
        "   add     r3, r0, #4          @ r3 = &(p->stack)      \n"
        "   ldr     r1, [r3, #0]        @ r1 = p->stack         \n"
        "   mov     sp, r1              @ switch back           \n"
        "   pop     {lr}                @ restore link-register \n"
        "                                                       \n"
        : : "r" (p), "r" (c), "r" (a)
        : "memory", "cc", "r0", "r1", "r2", "r3");
    } while (0)
```

A similar (though smaller) macro is used to switch back from the C run-time into the Guppy (or other lightweight) process. That particular piece of code simply jumps to the address setup in the assembler block above (label '`0:`') where the stack switch takes place.

4.2. Interfacing with LEGO® Devices

The "ev3dev" (ev3dev.org) open-source software provides a Debian-based Linux distribution for the LEGO® EV3. The specific platform drivers for the LEGO® motors and sensors are exposed to applications via '`/sys`' (and in the current version of the drivers, only via `/sys`). The device naming scheme used is a little verbose, but done in a way that allows the device-tree to be searched easily. For instance, a standard LEGO® EV3 (or NXT) motor connected to port "outB" can be controlled by writing to files in the directory '`/sys/class/tacho-motor/motor`*`NN`*`/`', where the number '*`NN`*' is a kernel-assigned counter — incremented for each motor that is (re-)connected. One of the files in this directory ('`port_name`') yields the physical port name associated with the motor; other files (when read or written) extract settings/values and/or control the motor in various ways. The specific details are less important, and well-documented on the 'ev3dev' site.

A collection of C functions is used to interact with LEGO® devices (motors and sensors) that present a more convenient abstraction then direct manipulation of files in '`/sys`', e.g. "make the motor on port A go forwards at 50% power". This code also handles the issue of mapping physical port names into logical motor numbers, that is straightforward but verbose. In order to call these C functions from Guppy, *external declarations* are made in a compiler-shipped header file ('`guppy_ev3lib.gpi`'). Such declarations provide a general way of interfacing with C code from within Guppy, similar to the external code calling mechanism in KRoC [18]. For example:

```
@external "guppy"
    "define ev3_mot_on_fwd (val ev3_outp p, val int pwr) -> bool = -1"
```

The type '`ev3_outp`' is an enumerated type with specific values attached to identifiers such as '`PORT_A`'. The integer specified at the end of the external declaration represents the amount of stack space required. The constant -1 indicates that the *actual* stack requirements must be provided elsewhere (typically the API call-chain file). For general external code calling, this must be specific, and accurate. The actual C function called must match how nocc transforms this prototype, in the case of the above:

```
void gcf_ev3_mot_on_fwd (Workspace wptr, int *result,
                                         int port, int power)
{
    ExternalCallN (igcf_ev3_mot_on, 3, result, port, power);
}
```

As is evident, this simply calls through to another C function '`igcf_ev3_mot_on`', passing it 3 parameters. The macro/function '`ExternalCallN`' (part of the CIF API [16]) is used to call a function using the *run-time system's* stack. Despite the overhead of the extra call and stack switch, this potentially saves on memory — not reserving space in the calling process's stack. The file manipulation involved, in this particular function has a relatively large stack requirement (kilobytes as opposed to tens of bytes). On a standard PC, with large amounts of paged virtual memory, this is not typically an issue. However, the LEGO® MINDSTORMS® EV3 has only 64 MiB of RAM, and typically around 20–30 MiB available in general use[8]. The swap-space, if available at all, would typically be on the installed SSD (flash-memory) card — thus, its use should be avoided where possible.

5. The Dining Philosophers

As a proof-of-concept, the classic *Dining Philosophers* problem (reformulated by Hoare from Dijkstra's original), has been built in LEGO® and programmed in Guppy. Due to the right-angled nature of LEGO® construction, and the fact there are only 4 output ports on the LEGO® EV3, this version of the program has just 4 philosophers and 4 forks. Figure 4 shows the process network, alongside the LEGO® model.

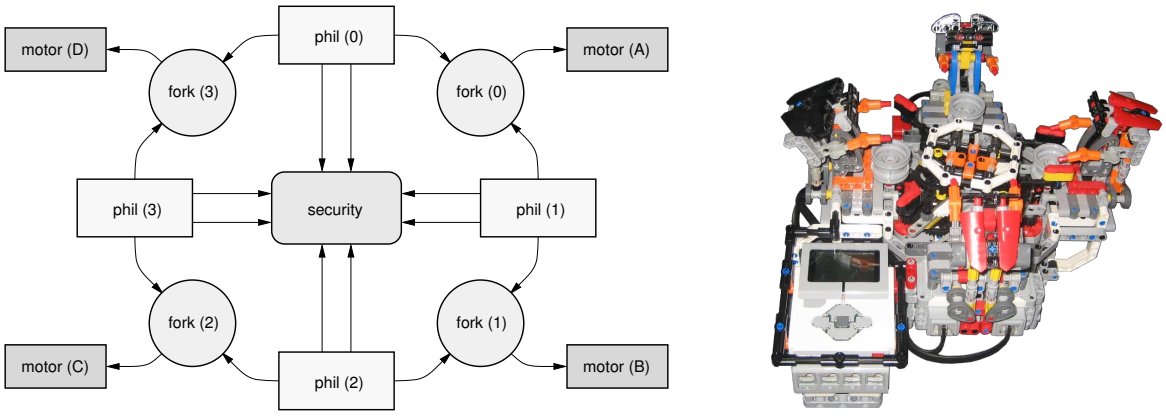


Figure 4. Dining-philosophers using the LEGO® EV3.

The operation of the '`phil`', '`fork`' and '`security`' processes is as expected: the forks are essentially mutual-exclusion semaphores (can be 'held' by one of the connected philosophers at a time) and the philosophers cycle through states of *thinking*, *hungry* and *eating*. Before eating, the philosopher must sit-down then acquire both forks. The security process prevents more than 3 philosophers sitting down simultaneously, avoiding the potential deadlock where each philosopher has one fork and waits for the other (forever). The '`fork`' processes are also connected to '`motor`' processes, that drive the actual LEGO® motors (and thus the forks). The interactions between the various processes are more extensive than normal, but only for appropriate synchronisation with the physical system (Guppy does not yet support the *extended-rendezvous* that would mitigate the need for this additional explicit synchronisation).

[8]More memory can be made available by removing/disabling services such as SSH, bluetooth, etc. and/or minimising the kernel configuration.

6. Conclusions and Further Work

This paper has touched on language design, compiler frameworks and run-time system implementation. A short-term aim, to demonstrate a proof-of-concept of process-oriented programming on the LEGO® MINDSTORMS® EV3, has achieved successfully. The long-term objective is to develop a replacement language for occam-π, with the same occam-π/CSP philosophies. This is on-going; Section 2 gives an (incomplete) overview of the language, but whether or not the final version *looks* the same (syntax), or what additional features might be added, is still open to suggestion: we are *experimenting*. The compiler framework used (Section 3) has been specifically engineered to enable language experimentation. This does not mean that it is straightforward to work with, but offers a reasonable trade-off between flexibility and performance.

The EV3 (and ARM9) specific run-time, Section 4, is functional though incomplete — it supports around half of the CIF API calls (compared to the CCSP occam-π run-time). However, this is sufficient in the short-term to work with the LEGO® EV3. One (presently) lacking aspect is handling sensor input with the EV3. A polling solution would be trivial to add (external C function calls) but is not efficient and incurs latency. The better (wait / interrupt) solution requires the use of blocking system calls (handled in a separate OS thread) that have not yet been implemented.

6.1. Performance

Table 3 shows comparative figures for the 'commstime' micro-benchmark. In terms of execution performance, the EV3 run-time described here clearly performs less well than the CCSP run-time. This is not unexpected, since very little effort has been spent optimising the EV3 run-time. The EV3's ARM CPU runs an order of magnitude slower than the typical PC (300 MHz vs. 3 GHz) but sub-microsecond context switch times should be attainable. The high cost of the parallel-delta in Guppy is due to the high cost of process creation and destruction, currently involving dynamic allocation and freeing of the process descriptor — this is considerably higher for the EV3 run-time (almost 19 microseconds).

Table 3. Micro-benchmark results for Guppy and the EV3 run-time (approximate).

	Guppy		occam-π
Benchmark	PC	EV3	PC
commstime par-delta context-switch (ns)	67	12,900	21
commstime seq-delta context-switch (ns)	41	3,500	13
commstime process start-stop (ns)	184	18,800	64
commstime process size (bytes)	1652	1508	496

In terms of memory occupancy, Guppy is comparable with occam-π, requiring approximately 3–4 times the space for this particular benchmark. This demonstrates how much more compact the existing occam-π toolchain is in terms of memory-allocation. A lot of the Guppy memory overhead is the process descriptor, requiring 116 bytes per process currently.

6.2. Current State and Future Directions

This paper has discussed lots of starts, but few ends, and much is still incomplete. The state of the implementation of Guppy within the 'nocc' compiler framework is best described as *work-in-progress*. Little has been implemented beyond what is required for the dining-philosophers, the commstime benchmark and other simple test programs. This includes a general lack of parallel-usage, aliasing and other semantic checks (though these are currently under development) — at the moment, programs are required to be semantically correct.

Other lacking language elements include arrays, records (partially supported), type parameters, structured protocol definitions and exception handling. The C code generator in nocc ('cccsp') does not yet perform arithmetic overflow checking correctly, and support for types other than integer or string is limited.

By comparison, the EV3 run-time is in a better state, being functionally half complete. Once functionally complete (in terms of the API required by compiled Guppy programs) the focus will shift towards optimisation.

6.3. Source Code and Documentation

The 'nocc' compiler framework and EV3 run-time can be obtained from Github:

```
https://github.com/concurrency/nocc
https://github.com/concurrency/kroc/tree/master/runtime/armccsp
```

Some documentation on 'nocc' can be found at:

```
https://www.cs.kent.ac.uk/research/groups/plas/wiki/NOCC/
```

Acknowledgements

"LEGO" and "MINDSTORMS" are registered trademarks of the LEGO Group. "Linux" is a registered trademark of Linus Torvalds. The term "EV3 run-time" appearing in the text refers to the run-time created and described here, targetting the LEGO® EV3 device, and is not in anyway affiliated nor endorsed by the LEGO® Group (Denmark).

References

[1] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[2] M. Homewood, D. May, D. Shepherd, and R. Shepherd. The IMS T800 Transputer. *IEEE Micro*, pages 10–26, October 1987.

[3] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.

[4] Christian L. Jacobsen and Matthew C. Jadud. Towards concrete concurrency: occam-pi on the LEGO mindstorms. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 431–435, New York, NY, USA, 2005. ACM Press.

[5] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. A Native Transterpreter for the LEGO Mindstorms RCX. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering*, pages 339–348, Amsterdam, The Netherlands, July 2007. IOS Press.

[6] Christian Jacobson and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, volume 62 of *WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575*, pages 99–106, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

[7] Inmos Limited. *The T9000 Transputer Instruction Set Manual*. SGS-Thompson Microelectronics, 1993. Document number: 72 TRN 240 01.

[8] Carl G. Ritson, Adam T. Sampson, and Frederick R.M. Barnes. Multicore scheduling for lightweight communicating processes. *Science of Computer Programming*, 77(6):727–740, June 2012. Article in press. doi:10.1016/j.scico.2011.04.006.

[9] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN: 0-52165-869-1.

[10] M.D. Poole. Extended Transputer Code - a Target-Independent Representation of Parallel Programs. In P.H. Welch and A.W.P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 187–198, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[11] Neil C.C. Brown and Adam T. Sampson. Alloy: Fast generic transformations for Haskell. In *Haskell '09: Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, pages 105–116, 2009.

[12] P. Hudak, S.L. Peyton-Jones, and P. Walder. Report on the Programming Language Haskell: A Non-Strict Purely Functional Language, version 1.2. *ACM SIGPLAN notices*, 27(5), May 1992.

[13] M. Debbage, M. Hill, S. Wykes, and Dennis Nicole. Southampton's portable occam compiler (SPOC). In R. Miles and A. Chalmers, editors, *Proceedings of WoTUG 17: Progress in Transputer and Occam Research*, volume 38 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, April 1994. IOS Press. ISBN: 90-5199-163-0.

[14] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es):154, 1996.

[15] Brian C. O'Neill, G.C. Coulson, Adam K.L. Wong, R. Hotchkiss, J.H. Ng, S. Clark, P.D. Thomas, and A. Crawley. A Distributed Parallel Processing System for the StrongARM Microprocessor. In P.H. Welch and A.W.P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 39–48, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[16] F.R.M. Barnes. Interfacing C and occam-pi. In J.F. Broenink, H.W. Roebbers, J.P.E. Sunter, P.H. Welch, and D.C. Wood, editors, *Proceedings of Communicating Process Architectures 2005*. IOS Press, September 2005.

[17] INMOS Limited. *ANSI C Language and Libraries Reference Manual*. INMOS Limited, 1992. Document number 72-TDS-347-01.

[18] David C. Wood. KRoC – Calling C Functions from occam. Technical report, Computing Laboratory, University of Kent at Canterbury, August 1998.