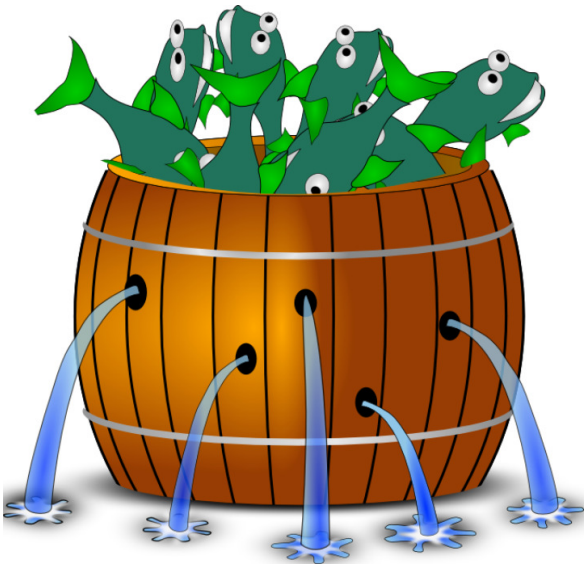
*Barrelfish Project*
*ETH Zurich*



# Sockeye in Barrelfish

*Barrelfish Technical Note 025*

Barrelfish project

09.02.2018

Systems Group
Department of Computer Science
ETH Zurich
CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland
`http://www.barrelfish.org/`

# Revision History

| Revision | Date | Author(s) | Description |
|---|---|---|---|
| 0.1 | 15.06.2017 | DS | Initial Version |
| 0.2 | 03.08.2017 | DS | Describe Modularity Features |
| 0.3 | 09.02.2018 | DS | Sockeye 2.0 |

# Contents

# Chapter 1

# Introduction and Usage

*Sockeye*[1] is a declarative domain specific language (DSL) to describe systems on a chip (SoCs)

Achermann et al. propose a formal model to describe hardware as a directed graph [1]. For the rest of this technote we'll call such a graph a hardware decoding net (HDN). The model captures the complex interactions within an between address translation hardware and the interrupt system. There is also work being done to extend the model to include clock distribution and power management.

Each node in the graph can accept a set of addresses and translate another (not necessarily disjunct) set of addresses (when describing interrupt routes they accept or translate interrupt vectors). While the nodes are modeled explicitly, the edges are implicitly given by the translation sets of the nodes.

Starting at a specific node, addresses can be resolved by following the appropriate edges in the HDN. When a node translates an address, resolution is continued at the referenced node. When a node accepts an address, resolution terminates

Sockeye uses HDNs as its underlying model. If offers language features to efficiently describe real hardware.

We currently envision two main use cases for Sockeye:

- Generate Isabell/HOL code from Sockeye specifications to be able to formally reason about the described hardware.
- Generate Prolog files that can be loaded into Barrelfish's System Knowledge Base (SKB) for the system to be able to reason about the hardware it's running on.

The Sockeye compiler is written in Haskell using the Parsec parsing library. The source code for the compiler can be found in `SOURCE/tools/sockeye`.

DS: The code for the old version is in the subdirectory `v1`. To not break building the tree `BUILD/tools/bin/sockeye` is compiled from the old code. The new code is compiled to `BUILD/tools/bin/sockeye2`.

[1] Sockeye salmon (Oncorhynchus nerka), also called red salmon, kokanee salmon, or blueback salmon, is an anadromous species of salmon found in the Northern Pacific Ocean and rivers discharging into it. This species is a Pacific salmon that is primarily red in hue during spawning. They can grow up to 84 cm in length and weigh 2.3 to 7 kg. Source: Wikipedia

## 1.1 Command Line Options

```
$ sockeye [options] file
```

The available options are:

**-P** Generate a Prolog file that can be loaded into the SKB (default).

**-I** Generate Isabelle/HOL code to formally reason about hardware.

**-i** Add a directory to the search path where Sockeye looks for imports.

**-o** `filename` The path to the output file (required)

**-d** `filename` The path to the dependency output file (optional)

**-h** show usage information

The backend (capital letter options) specified last takes precedence.

DS: The backends are not yet implemented in the new compiler version. For debugging purposes there are command line options to dump various internal data structures. Use -h for more info.

Multiple directories can be added by giving the `-i` options multiple times. Sockeye will first look for files in the current directory and then check the given directories in the order they were given.

When invoked with the `-d` option, the compiler will generate a dependency file for GNU make to be able to track changes in imported files.

The Sockeye file to compile is given via the `file` parameter.

# Chapter 2

# Language Constructs & Syntax

This chapter describes the language constructs in Sockeye and their syntax.

Most of the examples are taken from the Texas Instruments OMAP44xx SoC as used on the PandaboardES[1].

## 2.1 Natural Numbers

Sockeye supports addition, subtraction and multiplication of natural numbers. Additionally natural numbers can be interpreted as bit arrays and can then be *sliced* (selecting a contiguous range of bits of a number's binary representation) and *concatenated*.

```
/* Slicing: */
4[0] = 0
4[1] = 0
4[2] = 14
5[1 to 2] = 0b10 = 2

/* Concatenation: */
8 ++ 0xF[1 to 2] = 0b100011 = 35
```

The concatenation operator is left associative and the right hand side has to be a slice expression for the concatenation to be defined (the number of bits the left hand side has to be shifted has to be known). The operator precedence for the standard operations is as expected, slicing has higher precedence than the standard operations and concatenation has lower precedence.

Sockeye also has syntax to describe contiguous ranges of natural numbers:

[1] The technical reference manual can be found here.

```
/* Singleton range */
42

/* Base and limit */
0 to 8 // 0 up to and including 8
5 to 11

/* Base base and a number of variable bits */
(0 bits 2) // 0 up to and excluding 2^4

/* The variable bits have to be 0 in the base */
(0x100 bits 8) // OK
(0x110 bits 8) // Error
```

A (possibly sparse) set of natural numbers can be expressed as a comma-separated list of contiguous ranges.

```
/* These are equivalent */
(0 bits 2, 5 to 7, 11)
(0, 1, 2, 3, 5, 6, 7, 11)
```

## 2.2 Addresses

DS: Do we call them addresses, although might be interrupt vectors, clock signals etc?

In HDNs addresses are natural numbers. Sockeye models them as tuples of natural numbers. This allows easier modeling of cases where different parts of an address are used for different purposes. An example for this would e.g. be a lookup table that uses some bits as an index and prepends the rest of the incoming address with the value indexed by these bits. Note that this is not more powerful than the underlying model as there is a bijection between tuples of natural numbers and natural numbers (e.g. diagonalisation).

Sockeye syntax not only allows to specify single addresses but address sets. The dimensions of an address tuple are separated by semicolons:

```
(0; 8; 15)
(0x0; 0x8; 0xF)
(0 bits 12; 0 bits 8; 0 bits 12)
```

An address set contains all addresses in the Cartesian product of its dimensions.

## 2.3 Nodes

Sockeye nodes closely correspond to nodes in an HDN: They have a set of addresses that they accept, and a set of translations of incoming addresses to other nodes. In addition Sockeye nodes have an input and output *domain* and *type*.

**Domains** There are the following node domains in Sockeye:

- *memory*: Nodes in this domain are part of the memory system.
- *intr*: Nodes in this domain are part of the interrupt system.
- *power*: Nodes in this domain are part of the power management system.
- *clock*: Nodes in this domain are part of the clock distribution system.

Standard nodes have the same input and output domain. Nodes with different input and output domains are called *conversion nodes*. They offer a controlled way of crossing the boundaries between domains e.g. to model message signaled interrupts.

DS: Do we allow all combinations of input/output domains or do we want to e.g. disallow crossing from memory to clock?

**Types** The types in Sockeye allow to constrain the addresses. A type is simply an address set that the address has to be an element of.

Sockeye separates the declaration of a node and its definition. However, to keep specifications as readable as possible, the definition should immediately follow the declaration whenever possible.

**Node Declarations** A node is declared by giving its input domain and type and optionally its output domain and type.

Nodes can either be declared as single nodes or (sparse) node arrays. The possible array indexes are a set of natural numbers.

```
/* Declare 32bit addressed RAM node */
memory (0 bits 32) SDRAM

/* If the output domain is not given it defaults to the input domain */
memory (0 bits 32) memory SDRAM // Equivalent to the above

/*
 * Declare the physical address space of 4 cores
 * (Array indexes do not have to be contiguous)
 */
memory (0 bits 12; 0 bits 8; 0 bits 12) CPU_Physical[1, 3, 5, 7]

/* Declare a MSIx controller converting memory accesses to interrupts */
memory (0 bits 32) intr MSIx_CTRL

/*
 * Give output type to limit translation range
 * to interrupt vectors 0 ... 1023
 */
memory (0 bits 32) intr (0 to 1023) MSIx_CTRL
```

**Node Definitions** There are four types of statements to define nodes:

- **accepts** is used to define the accepted addresses of a node
- **maps** is used to define the translations done by a node
- **converts** is the same as **maps** but for conversion nodes. This definition statement is only allowed for conversion nodes. It is also the only one allowed for conversion nodes.

  DS: Do we need/want to allow others?

- **overlays** is used to define a default translation for a node. Any address that is neither accepted nor translated explicitly is translated to the overlay node at the same address.

A node's definition is given by the union of all the statements about the node. The definition can contain multiple statements of the same type. The order of the statements does not matter.

**Accepts** Accepting addresses are defined by giving a list of semicolon-separated address sets. Address sets are tuples of sets of natural numbers. All accepted addresses have to be elements of the nodes input type. To accept all addresses in a dimension of the node's input type, a wildcard can be used.

```
  /*
   * SDRAM accepts all 32bit addresses
   */
  memory (0 bits 32) SDRAM
  SDRAM accepts [(0 bits 32)]

 /* Multiple entries (equivalent to above) */
  SDRAM accepts [
    (0x00000000 bits 30);
    (0x40000000 bits 30);
    (0x80000000 bits 30);
    (0xC0000000 bits 30)
  ]

/* Multiple statements (equivalent to above) */
SDRAM accepts [(0x00000000 bits 31)]
SDRAM accepts [(0x80000000 bits 31)]

/* Wildcard (equivalent to above) */
SDRAM accepts [(*)]
```

**Maps/Converts** Translations are defined by giving a semicolon-separated list of origin addresses and translation targets. A translation targets consists of a node reference and a target address. The origin addresses have to be elements of the node's input type. The target node has to have the same input domain as the node's output domain. The target address has to be within the target nodes input type. If the node has an output type, the target address additionally needs to be within the output type. To translate contiguous ranges of addresses, an

address set can be given instead of the origin address. To translate all addresses in a dimension of the node's input type, a wildcard can be used.

```
memory (0 bits 8) SCU
memory (0 bits 8) Global_Timer
memory (0 bits 8) Private_Timers
memory (0 bits 8) GIC_PROC

memory (0 bits 13) PRIVATE_PERIPH

/* Map single address */
  PRIVATE_PERIPH maps [
    (0x100) to GIC_PROC at (0x0)
  ]

/* Map contiguous range to single address */
  PRIVATE_PERIPH maps [
    (0x200 bits 8) to Global_Timer at (0x0)
  ]

/* Use wildcard */
  PRIVATE_PERIPH maps [
    (*) to GIC_PROC at (0x0)
  ]
```

If some dimensions of the address set are contiguous, the target address can also be a set with these dimensions being contiguous ranges of the same cardinality. The number of variable dimensions in the origin target has to be the same.

```
  memory (0 bits 2; 0 bits 2) BUS_2D;
  memory (0 bits 2; 0 bits 2) RAM_2D;

  /* The following are equivalent */
  BUS_2D maps [
    (0; 0) to RAM_2D at (0; 0);
    (1; 1) to RAM_2D at (1; 1);
    (2; 2) to RAM_2D at (2; 2);
    (3; 3) to RAM_2D at (3; 3);
  ]
  BUS_2D maps [
    (0 to 3; 0 to 3) to RAM_2D at (0 to 3; 0 to 3)
  ]
  BUS_2D maps [
    (*; *) to RAM_2D at (*; *)
  ]
```

When modeling conversion nodes like MSIx controllers translating from the memory domain to the interrupt domain, the data word can be modeled as an address dimension to make the translation dependent on the data.

```
  memory (0 bits 32; 0 bits 32) intr (0 to 1023) MSIx_CTRL
  intr (0 to 1023) CPU_INTR

  /* Make translation dependent on data word (2nd dimension) */
  MSIx_CTRL converts [
    (0x0, 0 to 1023) to CPU_INTR at (*)
  ]
```

**Overlays** An overlay represents a default translation target for all addresses that are neither accepted nor explicitly translated. Nodes can only have overlays if their output type is either unspecified or the same as the input type. The overlay node's input type has to be the same as the one of the node.

```
/* Map everything 1:1 to the L2 bus except the private peripherals region */
memory (0 bits 32) L2

memory (0 bits 32) CORTEXA9_PHYS
  CORTEXA9_PHYS maps [
    (0x48240000 bits 13) to PRIVATE_PERIPH at (*)
  ]

  CORTEXA9_PHYS overlays L2
```

Describe properties

## 2.4 Named Types

Sockeye allows to define named types (similar to typedefs in C). When declaring nodes, these types can then be referenced by name.

```
type BUS_TYPE (0 bits 32)
memory (BUS_TYPE) L2
```

## 2.5 Quantifiers

Sockeye supports quantifying definition statements over sets of natural numbers. Origin addresses may not contain bound variables in arithmetic expressions.

```
/* Modelling a lookup table (LUT) */
memory (0 bits 32) CPU_Phys
memory (0 bits 8; 0 bits 24) LUT
memory (0 bits 32) BUS;

forall a in (0 bits 32)
  CPU_Phys maps [
    (a) to LUT at (a[24 to 31]; a[0 to 23])
  ]


forall a in (0 bits 24)
  LUT maps [
    (0x0000; a) to BUS at (0x1234 ++ a[0 to 24]);
    (0x0001; a) to BUS at (0x4321 ++ a[0 to 24]);
    /* ... */
  ]
```

Declaration statements are not allowed in the body of a quantifier.

## 2.6 Modules

A module encapsulates a (partial) HDN. In fact all HDNs are modules in Sockeye. Modules can be instantiated to integrate the contained HDN into a larger HDN. To connect a module instance with its environment, a module defines its interface in the form of ports. There are two types of ports:

- **Input Ports** are nodes *defined* within the module that can be referenced from outside the module.

- **Output Ports** are nodes *declared* within the module but not *defined*. Nodes in the module can reference these nodes. When a module is instantiated, all the output ports have to be bound to nodes with the same domains and types. A module can also e parametrized. All module parameters are natural numbers and a module defines the allowed range of each parameter. The parameters can be used to parametrise addresses or array sizes. Module instances, like nodes, can be declared as single instances or instance arrays. Module instantiation and port binding statements can also be quantified over a sets.

```
module CortexA9_Core((0 bits 32) periphbase)
  output memory (0 bits 32) L2

  input intr (0 to 1023) CPU_INTR

  memory PRIVATE_PERIPH

  CPU_PHYS overlays L2


module CortexA9_MPCore((1 to 4) num_cores)
  /* Declare a module instance array */
  instance Core[0 to num_cores-1] of CortexA9_Core

  memory (0 bits 32) Internal_BUS

  /* Instantiate the module and bind ports with quantifier */
  forall c in (0 to num_cores-1)
    Core[c] instantiates CortexA9_Core(0x48240000)
    Core[c] binds [
      /* Format: <output port> to <declared node> */
      L2 to Internal_BUS
    ]


  /* Or equivalently with wildcards */
  Core[*] instantiates CortexA9_Core(0x48240000)
  Core[*] binds [
    /* Format: <output port> to <declared node> */
    L2 to Internal_BUS
  ]
```

DS: For the compiler to be able to enforce the port interface, it should be the same for all instances in an array This means that parameters used in array declarations must be the same for all array elements. We could enforce this with a check, but maybe we should bake it into the syntax and have different constructs for the tow parameter classes.

In addition to declarations, quantifiers and definitions a module body can also contain named constants. Constants are natural numbers and can be used wherever a natural number is expected. Constants must not shadow module parameters.

```
module CortexA9_MPCore((1 to 4) num_cores)

  /* Define constant */
  const PERIPHBASE 0x48240000

  instance Core[0 to num_cores-1] of CortexA9_Core

  /* Use constant */
  Core[*] instantiates CortexA9_Core(PERIPHBASE)
```

Declarations, definitions, quantifiers and constant definitions may appear in any order in the body of a module.

## 2.7 Sockeye Files

A Sockeye file consists of named type and module definitions. The order does not matter. Due to how the import system works (see Section 2.8), modules and types share a namespace.

## 2.8 Import System

The import system in Sockeye allows to split the definition of a HDN over several files and reuse files in a library like fashion. A file can either be imported as a whole, meaning all modules and named types from the imported file become available inside the imported file. Alternatively modules and types can be selectively imported. With selective imports, modules and types can also be renamed to avoid name clashes. Imported modules and types are not re-exported.

Imports have to be specified at the top of a Sockeye file. Imported files must have the extension `soc`. Imported files are resolved relative to

1. The working directory of the compiler.
2. The directories given with the `-i` option, in the order they were given

```
/* Import all symbols defined in file cortexA9.soc */
import cortexA9

/* Only import symbol CortexA9_Core and CortexA9_MPCore */
import cortexA9 (CortexA9_Core, CortexA9_MPCore)

/* Only import symbol CortexA9_Core and rename it */
import cortexA9 (CortexA9_Core as Core)
```

## 2.9 Extended Example

The following is an extended example showcasing all the syntactic constructs.

```
/*
 * Lexical conventions:
 *
 * identifiers -> ([a-z] | [A-Z]){[a-z] | [A-Z] | [0-9] | '_'}
 * numbers -> decimal | hexadecimal
 * arithmetic_op -> '+' | '-' | '*' | '[]' | '++'
 * boolean_op -> '!' | '&&' | '||'
 */

module CortexA9_Core((0 to 0xFFFFE000) core_periphbase) { // Module parameters are natural
    ↪  numbers and have a range
    /*
     * Node domains: {memory, intr, power, clock}
     * Node types have arbitrary dimension, meaning is hardware dependent
     * E.g. here: 1st dimension: address, 2nd dimension data word
     */
    output memory (0 bits 8; 0 bits 32) SCU
    output memory (0 bits 8; 0 bits 32) Global_Timer
    output memory (0 bits 8; 0 bits 32) GIC_PROC        // 0 bits 8 == 0 to 2^8-1
    output memory (0 bits 12; 0 bits 32/*;*/) GIC_DISTR     // Trailing ';' should be
        ↪ allowed but optional (Parser does not yet allow it)
    output memory (0 bits 32; 0 bits 32) L2

    /*
     * Node declaration and definition is separate
     * Convention: keep together whenever possible
     */
    input intr (0 to 1023) CPU_INTR
    CPU_INTR accepts [
        (0, 2, 8 to 1023) // Specify sparse ranges
    ]

    memory (0 bits 8; 0 bits 32) Private_Timers
    Private_Timers accepts [
        (0 bits 8; *)
    ]

    memory (0 bits 13; 0 bits 32) PRIVATE_PERIPH
    PRIVATE_PERIPH maps [
        /* All dimensions of origin and target need to be specified */
        (0x0000 to 0x00FC; 0 bits 32) to SCU at (0x0 to 0xFC; 0 bits 32);

        /*
         * Wildcards map whole range of dimension
         * The following are equivalent
         */
        (0x0100 bits 8; *) to GIC_PROC at (*; *);
        (0x0100 bits 8; 0 bits 32) to GIC_PROC at (0 bits 8; 0 bits 32);

        /* Mapped ranges must have same size or target must be constant */
        (0x0200 bits 8; *) to Global_Timer at (*; *); // OK, one-to-one mapping
```

```
        (0x0200 bits 8; *) to Global_Timer at (*; 0); // OK, 2nd dimension collapsed to 0
        (0x0200 bits 9; *) to Global_Timer at (*; *); // Error

        (0x0600 bits 8; *) to Private_Timers at (*; *);
        (0x1000 bits 12; *) to GIC_DISTR at (*; *)/*;*/ // Trailing ';' should be allowed
            ↪ but optional (Parser does not yet allow it)
    ]

    memory (0 bits 32, 0 bits 32) CPU_PHYS
    CPU_PHYS maps [
        (core_periphbase bits 13; *) to PRIVATE_PERIPH at (*; *)
    ]
    CPU_PHYS overlays L2 // overlay node's type must be the same as node's type
}

module CortexA9_MPCore((1 to 4) num_cores, (0 to 0xFFFFE000) periphbase) {
    input intr (32 to 1019) GIC
    input intr (0 to 1023) CPU_INTR[0 to num_cores-1]
    output memory (0 bits 32, 0 bits 32) L2

    /*
     * Module instances must be declared
     */
    instance Core[0 to num_cores-1] of CortexA9_Core
    Core[0 to num_cores-1] instantiates CortexA9_Core(perhiphbase)

    memory (0 bits 8; 0 bits 32) SCU
    SCU accepts [(0x0 to 0xFC; *)]

    memory (0 bits 8; 0 bits 32) Global_Timer
    Global_Timer accepts [(0 bits 8; *)]

    memory (0 bits 8; 0 bits 32) GIC_PROC
    GIC_PROC accepts [(0 bits 8; *)]

    memory (0 bits 12; 0 bits 32) GIC_DISTR
    GIC_DISTR accepts [(0 bits 12; *)]

    forall c in (0 to num_cores-1) {
        /*
         * Instantiate module and bind output ports
         * Format: <output_port> to <node>
         * All output ports must be bound, types of port and node must match exactly
         */
        Core[c] binds [
            SCU to Cluster_SCU;
            Global_Timer to Global_Timer;
            GIC_PROC to GIC_PROC;
            GIC_DISTR to GIC_DISTR;
            L2 to L2
        ]

        /*
         * Reference input ports with dot notation
         */
```

```
        GIC maps [
            (*) to Core[c].CPU_INTR at (0 to 1019-32)
        ]

        /*
         * Input port pass-through, for the moment introduces proxy node
         */
        CPU_INTR[c] overlays Core[c].CPU_INTR
    }
}

/*
 * Named types
 * Only possible at file scope
 * If we have a use case, we might introduce module scope types
 */
type L2_Bus (0 bits 32; 0 bits 32)

module OMAP44xx {
    /*
     * Named constants
     * Only possible at module scope, use parameters to pass to other modules
     * Only natural number constants are possible.
     * If we have a use case we might introduce tuple constants
     */
    const PERIPHBASE 0x48240000
    const NUM_CORES 2

    /*
     * Multidimensional arrays through tuple indices
     */
    instance MPU[1 to 2; 1 to 2] of CortexA9_MPCore
    MPU[*; *] instantiates CortexA9_MPCore(NUM_CORES, PERIPHBASE)  // Use constants

    MPU[*; *] binds [
        L2 to L2
    ]

    intr (0 to 1023) INTR_CTRL
    forall s in (1 to 2) {
        INTR_CTRL maps [
            /* Multicast interrupt vector 1 to all 1st cores of MPU[1; 1] and MPU[2; 1] */
            (1) to MPU[s; 1].CPU_INTR[1] at (0);
            /* Same for vector 2 to 2nd cores */
            (2) to MPU[s; 1].CPU_INTR[2] at (0);
            /* Same for vector 3 to 1st cores of MPI[1; 2] and MPU[2; 2] */
            (3) to MPU[s; 2].CPU_INTR[1] at (0);
            /* And for vector 4 to 2nd cores */
            (4) to MPU[s; 2].CPU_INTR[2] at (0)/*;*/
        ]
    }

    /*
     * Node definitions can be split over several statements
     */
```

```
INTR_CTRL maps [
    (5) to MPU[1; 1].CPU_INTR[1] at (1)
]

memory (0 bits 30, 0 bits 32) SDRAM
SDRAM accepts [
    /*
     * Specify 1st order logic formula for properties that has to be true for the
         ↪ block to match
     */
    (0x00000000 bits 28; *) read && !write;  // read-only
    (0x10000000 bits 28; *) read && write;   // read-write
    (0x20000000 bits 28; *) read;            // read, don't care about write
    (0x30000000 bits 28; *)                 // don't care about properties
]

memory (L2_Bus) L2 // Reference named type
L2 maps [
    /*
     * Specify 1st order logic formulas for incoming and outgoing properties
     */
    (0x80000000 bits 28; *) !write to SDRAM at (0 bits 28; *) read && !write;  // map
        ↪ all non writeable to 1st quarter of RAM as read-only
    (0x80000000 bits 28; *) write to SDRAM at (0 bits 28; *) read && write;   // map
        ↪ all writeable to 2nd quarter of RAM as read-write
    (0x90000000 bits 28; *) to SDRAM at (0x10000000 bits 28; *) read && write;
    (0xA0000000 bits 28; *) to SDRAM at (0x20000000 bits 28; *) read, SDRAM at (0
        ↪ x30000000, 0); // Multicast
    (0xB0000000 bits 28; *) to SDRAM at (0x30000000 bits 28; *)/*;*/
]

/*
 * Converting from one namespace to another
 *
 * Destination type is optional, if specified, all target ranges have to match
 */
memory (0 bits 32, 0 bits 32) to intr (32 to 1019) CHIPSET
CHIPSET converts [
    // (0; 0 to 1019-32) !read to MPU[1; 1].GIC at (*) edge_trig;
    (0; 1) to MPU[1;1].CPU_INTR[1] at (0); // Error, type does not match
    (0; 2) to MPU[1;1].CPU_INTR[1] at (1)/*;*/ // Error, type does not match
]

intr (0 to 1023) to memory MSI_CTRL
MSI_CTRL converts [
    /* ... */
]

/*
 * Changing the dimensionality through a mapping
 */
memory (0 bits 7; 0 to 1) RAM_2D
RAM_2D accepts [(*; *)]

memory (0 bits 3; 0 bits 3; 0 bits 2) BUS_3D
```

```
    forall a in (0 bits 3) {
        forall b in (0 bits 3) {
            forall c in (0 bits 2) {
                BUS_3D maps [
                    /* Split address parts with slice operator: */
                    (0; 0; c) to RAM_2D at (c[1]; c[0]);

                    /* Concatenate address parts with concat/slice operator :*/
                    (a; b; 0) to RAM_2D at (a ++ b[0 to 2]; 0);
                    (a; b; 1) to RAM_2D at (a ++ b[0 to 2]; 1);

                    /*
                     * ++[] is left associative
                     * The following are equal:
                     */
                    (a; b; c) to RAM_2D at ( a ++ b[0 to 2]  ++ c[1]; c[0]);
                    (a; b; c) to RAM_2D at ((a ++ b[0 to 2]) ++ c[1]; c[0])/*;*/
                ]
            }
        }
    }
}
```

Listing 2.1: Sockeye Example

# Chapter 3

# Lexical Conventions

The Sockeye parser uses the following conventions:

**Encoding** The file should be encoded using plain text.

**Whitespace:** As in C and Java, Sockeye considers sequences of space, newline, tab, and carriage return characters to be whitespace. Whitespace is generally not significant.

**Comments:** Sockeye supports C-style comments. Single line comments start with `//` and continue until the end of the line. Multiline comments are enclosed between `/*` and `*/`; anything in between is ignored and treated as white space. Nested comments are not supported.

**Identifiers:** Valid Sockeye identifiers are sequences of numbers (0-9), letters (a-z, A-Z), the underscore character "_" and the dash character "–". They must start with a letter.

$$\begin{aligned} \mathit{identifier} &\rightarrow \mathit{letter}(\mathit{letter} \mid \mathit{digit} \mid \_)^* \\ \mathit{letter} &\rightarrow (\mathsf{A} \ldots \mathsf{Z} \mid \mathsf{a} \ldots \mathsf{z}) \\ \mathit{digit} &\rightarrow (\mathsf{0} \ldots \mathsf{9}) \end{aligned}$$

**Case sensitivity:** Sockeye is case sensitive hence identifiers `node1` and `Node2` are not the same.

**Integer Literals:** A Sockeye integer literal is a sequence of digits, optionally preceded by a radix specifier. As in C, decimal (base 10) literals have no specifier and hexadecimal literals start with `0x`.

$$\begin{aligned} \mathit{decimal} &\rightarrow (\mathsf{0} \ldots \mathsf{9})^1 \\ \mathit{hexadecimal} &\rightarrow (\mathsf{0x})(\mathsf{0} \ldots \mathsf{9} \mid \mathsf{A} \ldots \mathsf{F} \mid \mathsf{a} \ldots \mathsf{f})^1 \end{aligned}$$

**Reserved words:** The following are reserved words in Sockeye:

```
import, as, module, input, output, type, const,
memory, intr, power, clock,
instance, of, forall, in,
accepts, maps, converts, overlays, instantiates, binds,
to, at, bits
```

# Chapter 4

# Checks

The following is a list of checks the Sockeye compiler performs. The goal is to have modular type checking, such that modules are save to instantiate at runtime with any parameters in the corresponding range. However, with parametrised sets of natural numbers this needs more involved static analysis. It might prove easier to check some of the properties in Isabelle.

## 4.1 Uniqueness Checks

**Circular Imports** Throws an error if there are cycles in the import graph.

**Duplicate Import** Throws an error two symbols with the same name are imported or two symbols are renamed to the same name while importing them.s

**No Such Export** Throws an error if a selective import tries to import a symbol that does not exist in the imported file.

**Import Shadowing** Throws an error if a type or module defined in a file has the same name as an imported symbol.

**Duplicate Module** Throws an error if two modules with the same name are defined in the same file.

**Duplicate Type** Throws an error if two types with same name are defined in the same file.

**Module/Type Clash** Throws an error if a type and a module defined in the same file have the same name.

**Duplicate Parameter** Throws an error if a module has two parameters with the same name.

**Duplicate Constant** Throws an error if two constants defined in the same module have the same name.

**Duplicate Variable** Throws an error if there are two bound variables with the same name in the same scope.

**Parameter Shadowing** Throws an error if a constant or bound variable has the same name as a module parameter of the module in which it is defined.

Check only implemented for constants, not for bound variables

**Constant Shadowing** Throws an error if a bound variable has the same name as a named constant.

Not yet implemented

**Duplicate Instance** Throws an error if there are two module instance declarations with the same name in the same module.

**Duplicate Node** Throws an error if there are two node declarations with the same name in the same module.

**Instance/Node Clash** Throws an error if there is an instance and a node declaration with the same name in the same module.

Not yet implemented

## 4.2 Integrity Checks

None of these checks are currently implemented

**Undefined Type** Throws an error if a named type that is neither defined in the same file nor imported is referenced.

**Undefined Module** Throws an error if a module that is neither defined in the same file nor imported is instantiated.

**Undefined Instance Reference** Throws an error if an undefined instance is referenced Note that this only checks whether the name exists and whether an instance array is treated as a single instance or vice versa. It does not check whether the array index exists.

**Module Instantiation Mismatch** Throws an error if the instantiated module in an **instantiates** statement does not match the module of the declared instance.

**Argument Count Mismatch** Throws an error if a module is instantiated with the wrong number of arguments.

**Undefined Output Port** Throws an error if an output port being bound does not exist on the module. Note that this only checks whether there is an output port with the referenced name and whether a output port array is treated as a single port or vice versa. It does not check whether the array index exists.

**Undefined Node Reference** Throws an error if an *internal node* is referenced that does not exist. Note that this only checks whether there is an internal node with the referenced name and whether a node array is treated as a single node or vice versa. It does not check whether the array index exists.

**Undefined Input Port** Throws an error if an input port is referenced that does not exist. Note that this only checks whether there is an input port with the referenced name and whether an input port array is treated as a single port or vice versa. It does not check whether the array index exists.

**Undefined Variable** Throws an error if a variable is referenced and there is neither a module parameter, constant or bound variable with that name.

## 4.3 Semantic Checks

None of these checks are currently implemented

DS: Note concerning all the checks that involve analysing sets and arithmetic expressions. With parametrised sets and variables in arithmetic expressions this will not be feasible exactly. Depending on how precise the approximation should be, it might be easier to perform the checks in Isabelle.

**Argument not in Range** Throws an error if an argument to a module instantiation is not in the allowed range of the corresponding parameter.

**Array Index Does not Exist** Throws an error when an array is indexed with an index it does not contain.

**Uninstantiated Instance** Throws an error if a declared instance is never instantiated.

**Duplicate Instantiation** Throws an error if a declared instance is instantiated twice.

**Module Instantiation Loop** Throws an error if there is a loop in module instantiations and therefore potentially infinite nesting of HDNs.

Might also just be a warning

**Unbound Output Port** Throws an error if an output port of a module instance is never bound.

**Duplicate Port Binding** Throws an error if an output port of a module instance is bound twice.

**Empty Node** Throws an error if a node is declared but no definition statement ever references it.

Might also just be a warning

**Domain Mismatch** Throws an error if

- A target node's input domain in a translation does not match the output domain of the origin node.
- An output ports input or output domain are different from the node it is bound to.
- An overlay nodes input or output domain are different from the node.

**Node Type Mismatch** Throws an error if

- An origin address in a translation is not in the origin node's input type.
- A target address in a translation is not in the origin node's output type (if given).
- A target address in a translation is not in the target node's input type.
- An output port is bound to a node with a different type.
- An overlay node does not have the same type as the node itself.

**Illegal Translation** Throws an error if a translation is specified using address sets and on of the following is violated:

- A dimension's range is not contiguous
- The number of variable dimensions in the target address does not match the number of variable dimensions in the origin address
- The cardinality of corresponding variable dimensions does not match.

**Bound Variable in Arithmetic Expression** Throws an error if an origin address of a translation contains an arithmetic expressions with bound variables. Bound variables can only occur in origin addresses if no arithmetic is done with them.

Not yet implemented

**Unknown Bit Width for Concatenation** Throws an error if the right hand side of a concatenation operator is not a slice expression and the bit width can therefore not be inferred.

Not yet implemented

# Chapter 5

# Prolog Mapping for Sockeye

LH: Work in progress for the new Prolog backend.

The definitions of the struct and dynamic facts (and queries, once we implement them) can be found in the Barrelfish SKB file `usr/skb/programs/decoding_net2.pl`.

The Sockeye compiler generates ECL$^i$PS$^e$-Prolog[1] to be used within the SKB.

The *address domain* is represented as an atom (one of memory, intr, power, clock).

An *address* is a tuple of first, an address domain, followed by an integer corresponding to each address dimension.

Since we allow multi dimensional and non-zero based arrays, an *array index* is represented as a tuple of integers.

*Node identifiers* are represented as a list of strings and integers. The list of creates a hiearchical name, each module instance name will be prepended to the list until it ends at a local node name. In case of arrays, the array name is added prepended by the array index.

According to the encoding of addresses, *Address blocks* are tuples of an address domain, followed by for each dimension, a `block` functor is instantiated. A `block` functor takes two arguments, base and limit, restricting the corresponding address dimension.

We then use these definitions to express a HDN with the following predicates

- `node_translate(src_node :: node_id, src_address_block :: address_block, dest_node ↪ :: node_id, dest_address_block :: address_block)`
- `node_accepts(node :: node_id, address_block :: address_block)`
- `node_overlays(src_node :: node_id, dest_node :: node_id)`

To assert these predicates, the compiler generates a predicate for each module.

The code is generated using ECL$^i$PS$^e$'s structure notation. This enables more readable and concise notation when accessing specific arguments of the functors.

Listing 5.2 shows the generated Prolog code for the Sockeye example in Listing 5.1. Listing 5.3 shows the facts that are instantiated after calling `add_SOCKET(["root"])`.

[1] http://eclipseclp.org/

```
module DRAM {
    input memory (0 bits 40) GDDR0
    output memory (0 bits 40) RAMOUT
    GDDR0 accepts [(0x000000000 to 0x0fedfffff)]

    memory (0 bits 40) DRAMMAP
    DRAMMAP maps [
        (0x000000000 to 0x0fedfffff) to RAMOUT at (0x000000000 to 0x0fedfffff)
    ]
}

module SOCKET {
    instance RAM[1 to 2; 10 to 11] of DRAM

    memory (0 bits 40) LOCAL
    LOCAL accepts [(0x000000000 to 0x0fedfffff)]

    memory (0 bits 40) LOCAL_SRC

    forall x in (1 to 2, 10 to 11) {
        forall y in (10 to 11) {
            RAM[x; y] instantiates DRAM
            RAM[x; y] binds [
                RAMOUT to LOCAL
            ]

            LOCAL_SRC maps [
                (0x1000 to 0x2000) to RAM[x; y].GDDR0 at (0x1000 to 0x2000)
            ]
        }
    }

}
```

Listing 5.1: Sockeye Example

```
add_DRAM(Id) :-
    is_list(Id),
    (ID_GDDR0,INKIND_GDDR0,OUTKIND_GDDR0) = (["GDDR0" | Id],memory,memory),
    (ID_RAMOUT,INKIND_RAMOUT,OUTKIND_RAMOUT) = (["RAMOUT" | Id],memory,memory),
    (ID_DRAMMAP,INKIND_DRAMMAP,OUTKIND_DRAMMAP) = (["DRAMMAP" | Id],memory,memory),
    assert(node_accept(ID_GDDR0,([block{base:0,limit:4276092927}]))),
    assert(node_translate(ID_DRAMMAP,([block{base:0,limit:4276092927}]),ID_RAMOUT,([block{
        ↪ base:0,limit:4276092927}]))).

add_SOCKET(Id) :-
    is_list(Id),
    (ID_LOCAL,INKIND_LOCAL,OUTKIND_LOCAL) = (["LOCAL" | Id],memory,memory),
    (ID_LOCAL_SRC,INKIND_LOCAL_SRC,OUTKIND_LOCAL_SRC) = (["LOCAL_SRC" | Id],memory,memory)
        ↪ ,
    ID_RAM = ["RAM" | Id],
    assert(node_accept(ID_LOCAL,([block{base:0,limit:4276092927}]))),
    (block_values([block{base:1,limit:2},block{base:10,limit:11}],IDL_x),(foreach(IDT_x,
        ↪ IDL_x),param(Id),param(ID_LOCAL),param(ID_LOCAL_SRC),param(ID_RAM) do

    (block_values([block{base:10,limit:11}],IDL_y),(foreach(IDT_y,IDL_y),param(Id),param(
        ↪ ID_LOCAL),param(ID_LOCAL_SRC),param(ID_RAM),param(IDT_x) do
        add_DRAM([[IDT_x,IDT_y] | ID_RAM]),
        assert(node_overlay(["RAMOUT" | [[IDT_x,IDT_y] | ID_RAM]],ID_LOCAL)),
        assert(node_translate(ID_LOCAL_SRC,([block{base:4096,limit:8192}]),["GDDR0" | [[
            ↪ IDT_x,IDT_y] | ID_RAM]],([block{base:4096,limit:8192}])))
    ))
)).
```

Listing 5.2: Generated Prolog code

```
node_overlay(["RAMOUT", [1, 10], "RAM", "root"], ["LOCAL", "root"]).
node_overlay(["RAMOUT", [1, 11], "RAM", "root"], ["LOCAL", "root"]).
node_overlay(["RAMOUT", [2, 10], "RAM", "root"], ["LOCAL", "root"]).
node_overlay(["RAMOUT", [2, 11], "RAM", "root"], ["LOCAL", "root"]).
node_overlay(["RAMOUT", [10, 10], "RAM", "root"], ["LOCAL", "root"]).
node_overlay(["RAMOUT", [10, 11], "RAM", "root"], ["LOCAL", "root"]).
node_overlay(["RAMOUT", [11, 10], "RAM", "root"], ["LOCAL", "root"]).
node_overlay(["RAMOUT", [11, 11], "RAM", "root"], ["LOCAL", "root"]).

node_accept(["LOCAL", "root"], [block(0, 4276092927, _142)]).
node_accept(["GDDR0", [1, 10], "RAM", "root"], [block(0, 4276092927, _152)]).
node_accept(["GDDR0", [1, 11], "RAM", "root"], [block(0, 4276092927, _152)]).
node_accept(["GDDR0", [2, 10], "RAM", "root"], [block(0, 4276092927, _152)]).
node_accept(["GDDR0", [2, 11], "RAM", "root"], [block(0, 4276092927, _152)]).
node_accept(["GDDR0", [10, 10], "RAM", "root"], [block(0, 4276092927, _152)]).
node_accept(["GDDR0", [10, 11], "RAM", "root"], [block(0, 4276092927, _152)]).
node_accept(["GDDR0", [11, 10], "RAM", "root"], [block(0, 4276092927, _152)]).
node_accept(["GDDR0", [11, 11], "RAM", "root"], [block(0, 4276092927, _151)]).

node_translate(["DRAMMAP", [1, 10], "RAM", "root"], [block(0, 4276092927, _182)], ["RAMOUT
    ↪ ", [1, 10], "RAM", "root"], [block(0, 4276092927, _192)]).
node_translate(["LOCAL_SRC", "root"], [block(4096, 8192, _172)], ["GDDR0", [1, 10], "RAM",
    ↪  "root"], [block(4096, 8192, _192)]).
node_translate(["DRAMMAP", [1, 11], "RAM", "root"], [block(0, 4276092927, _182)], ["RAMOUT
    ↪ ", [1, 11], "RAM", "root"], [block(0, 4276092927, _192)]).
node_translate(["LOCAL_SRC", "root"], [block(4096, 8192, _172)], ["GDDR0", [1, 11], "RAM",
    ↪  "root"], [block(4096, 8192, _192)]).
node_translate(["DRAMMAP", [2, 10], "RAM", "root"], [block(0, 4276092927, _182)], ["RAMOUT
    ↪ ", [2, 10], "RAM", "root"], [block(0, 4276092927, _192)]).
node_translate(["LOCAL_SRC", "root"], [block(4096, 8192, _172)], ["GDDR0", [2, 10], "RAM",
    ↪  "root"], [block(4096, 8192, _192)]).
node_translate(["DRAMMAP", [2, 11], "RAM", "root"], [block(0, 4276092927, _182)], ["RAMOUT
    ↪ ", [2, 11], "RAM", "root"], [block(0, 4276092927, _192)]).
node_translate(["LOCAL_SRC", "root"], [block(4096, 8192, _172)], ["GDDR0", [2, 11], "RAM",
    ↪  "root"], [block(4096, 8192, _192)]).
node_translate(["DRAMMAP", [10, 10], "RAM", "root"], [block(0, 4276092927, _182)], ["
    ↪ RAMOUT", [10, 10], "RAM", "root"], [block(0, 4276092927, _192)]).
node_translate(["LOCAL_SRC", "root"], [block(4096, 8192, _172)], ["GDDR0", [10, 10], "RAM"
    ↪ , "root"], [block(4096, 8192, _192)]).
node_translate(["DRAMMAP", [10, 11], "RAM", "root"], [block(0, 4276092927, _182)], ["
    ↪ RAMOUT", [10, 11], "RAM", "root"], [block(0, 4276092927, _192)]).
node_translate(["LOCAL_SRC", "root"], [block(4096, 8192, _172)], ["GDDR0", [10, 11], "RAM"
    ↪ , "root"], [block(4096, 8192, _192)]).
node_translate(["DRAMMAP", [11, 10], "RAM", "root"], [block(0, 4276092927, _182)], ["
    ↪ RAMOUT", [11, 10], "RAM", "root"], [block(0, 4276092927, _192)]).
node_translate(["LOCAL_SRC", "root"], [block(4096, 8192, _172)], ["GDDR0", [11, 10], "RAM"
    ↪ , "root"], [block(4096, 8192, _192)]).
node_translate(["DRAMMAP", [11, 11], "RAM", "root"], [block(0, 4276092927, _182)], ["
    ↪ RAMOUT", [11, 11], "RAM", "root"], [block(0, 4276092927, _192)]).
node_translate(["LOCAL_SRC", "root"], [block(4096, 8192, _171)], ["GDDR0", [11, 11], "RAM"
    ↪ , "root"], [block(4096, 8192, _191)]).
```

Listing 5.3: Asserted Prolog facts

# Chapter 6

# Compiling Sockeye Files with Hake

SoC descriptions are placed in the directory `SOURCE/socs` with the file extension `soc`. Each top-level Sockeye file has to be added to the list of SoCs in the Hakefile in the same directory. When passed a filename (without extension), the function `sockeye :: String -> HRule` creates a rule to compile the file `SOURCE/socs/<filename>.soc` to `BUILD/sockeyefacts/<filename>.pl`. The rule will also generate `BUILD/sockeyefacts/<filename>.pl.depend` (with the `-d` option of the Sockeye compiler) and include it in the Makefile. This causes `make` to rebuild the file also when imported files are changed. To add a compiled Sockeye specification to the SKB RAM disk, the filename can be added to the `sockeyeFiles` list in the SKB's Hakefile.

# References

[1] R. Achermann, L. Humbel, D. Cock, and T. Roscoe. Formalizing memory accesses and interrupts. In *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems*, pages 66–116, 2017.