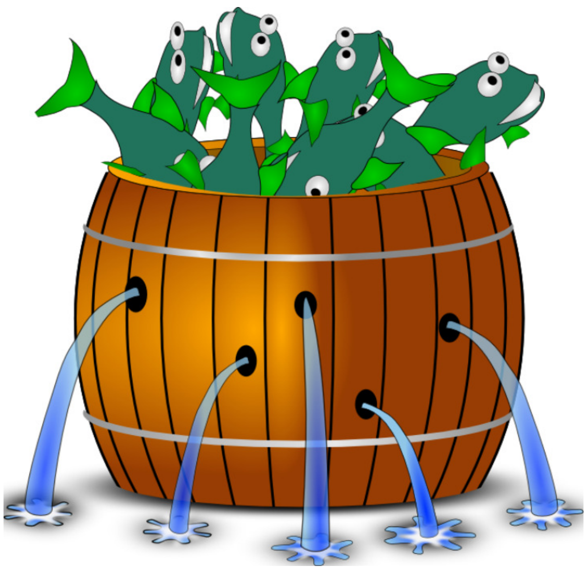
Barrelfish Project
ETH Zurich



Skate in Barrelfish

Barrelfish Technical Note 020

Barrelfish project

20.04.2017

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	16.11.2015	MH	Initial Version
0.2	20.04.2017	RA	Renaming ot Skate and expanding.

Contents

Chapter 1

Introduction and usage

Skate[1] is a domain specific language to describe the schema of Barrelfish's System Knowledge Base (SKB) [?]. The SKB stores all statically or dynamically discovered facts about the system. Static facts are known and exist already at compile time of the SKB ramdisk or are added through an initialization script or program.

Examples for static facts include the device database, that associates known drivers with devices or the devices of a wellknown SoC. Dynamic facts, on the otherhand, are added to the SKB during and based on hardware discovery. Examples for dynamic facts include the number of processors or PCI Express devices.

Inside the SKB, a prolog based constraint solver takes the added facts and computes a solution for hardware configuration such as PCI bridge programming, NUMA information for memory allocation or device driver lookup. Programs can query the SKB using Prolog statements and obtain device configuration and PCI bridge programming, interrupt routing and constructing routing trees for IPC. Applications can use information to determine hardware characteristics such as cores, nodes, caches and memory as well as their affinity.

The Skate language is used to define format of those facts. The DSL is then compiled into a set of fact definitions and functions that are wrappers arround the SKB client functions, in particular `skb_add_fact()`, to ensure the correct format of the added facts.

The intention when designing Skate is that the contents of system descriptor tables such as ACPI, hardware information obtained by CPUID or PCI discovery can be extracted from the respective manuals and easily specified in a Skate file.

Skate complements the SKB by defining a *schema* of the data stored in the SKB. A schema defines facts and their structure, which is similar to Prolog facts and their arity. A code-generation tool generates a C-API to populate the SKB according to a specific schema instance.

The Skate compiler is written in Haskell using the Parsec parsing library. It generates C header files from the Skate files. In addition it supports the generation of Schema documentation.

The source code for Skate can be found in `SOURCE/tools/skate`.

[1]Skates are cartilaginous fish belonging to the family Rajidae in the superorder Batoidea of rays. More than 200 species have been described, in 32 genera. The two subfamilies are Rajinae (hardnose skates) and Arhynchobatinae (softnose skates). Source: Wikipedia

1.1 Use cases

We envision the following non exhausting list of possible use cases for Skate:

- A programmer is writing PCI discovery code or a device driver. The program inserts various facts about the discovered devices and the state of them into the SKB. To make the inserted facts usable to other programs running on the system, the format of the facts have to be known. For this purpose we need a common to specify the format of those facts and their meaning.
- Each program needs to ultimately deal with the issue of actually inserting the facts into the SKB or query them. For this purpose, the fact strings need to be formatted accordingly, and this may be done differently for various languages and is error prone to typos. Skate is intended to remove the burden from the programmer by providing a language native (e.g. C or Rust) to ensure a safe way of inserting facts into the SKB.
- Just knowing the format and the existence of certain facts is useless. A programmer needs to understand the meaning of them and their fields. It's not enough just to list the facts with the fields. Skate provides a way to generate a documentation about the specified facts. This enables programmers to reason about which facts should be used in particular selecting the right level of abstraction. This is important given that facts entered into the SKB from hardware discovery are intentionally as un-abstracted as possible.
- Documenting the available inference rules that the SKB implements to abstract facts into useful concepts for the OS.

1.2 Command line options

```
$ skate <options> INFILE.skt
```

Where options is one of

-o *filename* The output file name

-L generate latex documentation

-H generate headerfile

-W generate Wiki syntax documentation

Chapter 2

Lexical Conventions

The Skate parser follows a similar convention as opted by modern day programming languages like C and Java. Hence, Skate uses a java-style-like parser based on the Haskell Parsec Library. The following conventions are used:

Encoding The file should be encoded using plain text.

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

Comments: Skate supports C-style comments. Single line comments start with // and continue until the end of the line. Multiline comments are enclosed between /* and */; anything inbetween is ignored and treated as white space.

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

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

Note that a single underscore "_" by itself is a special, "don't care" or anonymous identifier which is treated differently inside the language.

Case Sensitivity Skate is not case sensitive hence identifiers `foo` and `Foo` will be the same.

Integer Literals: A Skate 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`. Binary literals start with `0b`.

In addition, as a special case the string `1s` can be used to indicate an integer which is composed entirely of binary 1's.

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

String Literals String literals are enclosed in double quotes and should not span multiple lines.

Reserved words: The following are reserved words in Skate:

```
schema, fact, flags, constants, enumeration, text, section
```

Special characters: The following characters are used as operators, separators, terminators or other special purposes in Skate:

```
{ } [ ] ( ) + - * / ; , . =
```

Chapter 3

Schema Declaration

In this chapter we define the layout of a Skate schema file, which declarations it must contain and what other declarations it can have. Each Skate schema file defines exactly one schema, which may refer to other schemas.

3.1 Syntax Highlights

In the following sections we use the syntax highlighting as follows:

bold:	Keywords
italic:	Identifiers / strings chosen by the user
verbatim	constructs, symbols etc

3.2 Conventions

There are a some conventions that should be followed when writing a schema declaration. Following the conventions ensures consistency among different schemas and allows generating a readable and well structured documentation.

Identifiers Either camelcase or underscore can be used to separate words. Identifiers must be unique i.e. their fully qualified identifier must be unique. A fully qualified identifier can be constructed as $schema.(namespace.)*name$.

Descriptions The description fields of the declarations should be used as a more human readable representation of the identifier. No use of abbreviations or

Hierarchy/Grouping Declarations of the same concept should be grouped in a schema file (e.g. a single ACPI table). The declarations may be grouped further using namespaces (e.g. IO or local interrupt controllers)

Sections/Text Additional information can be provided using text blocks and sections. Each declaration can be wrapped in a section.

which conventions do we actually want

3.3 The Skate File

A Skate file must consist of zero or more *import* declarations (see Section 3.6) followed by a single *schema* declaration (see Section 3.6) which contains the actual definitions. The Skate file typically has the extension **.sks*, referring to a Skate (or SKB) schema.

```
/* Header comments */
(import schema)*

/* the actual schema declaration */
schema theschema "" {...}
```

Note that all imports must be stated at the beginning of the file. Comments can be inserted at any place.

3.4 Imports

An import statement makes the definitions in a different schema file available in the current schema definition, as described below. The syntax of an import declaration is as follows:

Syntax

```
import schema;
```

Fields

schema is the name of the schema to import definitions from.

The order of the imports does not matter to skate. At compile time, the Skate compiler will try to resolve the imports by searching the include paths and the path of the current schema file for an appropriate schema file. Imported files are parsed at the same time as the main schema file. The Skate compiler will attempt to parse all the imports of the imported files transitively. Cyclic dependencies between device files will not cause errors, but at present are unlikely to result in C header files which will successfully compile.

3.5 Types

The Skate type system consists of a set of built in types and a set of implicit type definitions based on the declarations of the schema. Skate performs some checks on the use of types.

3.5.1 BuiltIn Types

Skate supports the common C-like types such as integers, floats, chars as well as boolean values and Strings (character arrays). In addition, Skate treats the Barrelfish capability reference (`struct capref`) as a built in type.

```
    UInt8, UInt16, UInt32, UInt64, UIntPtr
    Int8, Int16, Int32, Int64, IntPtr
    Float, Double
    Char, String
    Bool
    Capref
```

3.5.2 Declaring Types

All declarations stated in Section 3.8 are implicitly types and can be used within the fact declarations. This can restrict the values that are valid in a field. The syntax of the declarations enforces certain restrictions on which types can be used in the given context.

In particular, fact declarations allow fields to be of type fact which allows a notion of inheritance and common abstractions. For example, PCI devices and USB devices may implement a specialization of the device abstraction. Note, cicular dependencies must be avoided.

Defining type aliases using a a C-Like typedef is currently not supported.

3.6 Schema

A schema groups all the facts of a particular topic together. For example, a schema could be the PCI Express devices, memory regions or an ACPI table. Each schema must have a unique name, which must match the name of the file, and it must have at least one declaration to be considered a valid file. All checks that are being executed by Skate are stated in Chapter 4. There can only be one schema declaration in a single Schema file.

Syntax

```
schema name "description" {
  declaration;
  ...
};
```

Fields

name is an identifier for the Schema type, and will be used to generate identifiers in the target language (typically C). The name of the schema *must* correspond to the filename of the file, including case sensitivity: for example, the file `cpuid.sks` will define a schema type of name `cpuid`.

description is a string literal in double quotes, which describes the schema type being specified, for example `"CPUID Information Schema"`.

declaration must contain at least one of the following declarations:

- namespace – Section 3.7
- flags – Section 3.8.1
- constants – Section 3.8.2
- enumeration – Section 3.8.3
- facts – Section 3.8.4
- section – Section 3.9.3
- text – Section 3.9.2

3.7 Namespaces

The idea of a namespaces is to provide more hierarchical structure similar to Java packages or URIs (schema.namespace.namespace) For example, a PCI devices may have virtual and physical functions or a processor has multiple cores. Namespaces can be nested within a schema to build a deeper hierarchy. Namespaces will have an effect on the code generation.

does everything has to live in a namespace?, or is there an implicit default namespace?

Syntax

```
namespace name "description" {
    declaration;
    ...
};
```

Fields

name the identifier of this namespace.

description human readable description of this namespace

declarations One or more declarations that are valid a schema definition.

3.8 Declarations

In this section we define the syntax for the possible fact, constant, flags and enumeration declarations in Skate. Each of the following declarations will define a type and can be used.

3.8.1 Flags

Flags are bit fields of a fixed size (8, 16, 32, 64 bits) where each bit position has a specific meaning e.g. the CPU is enabled or an interrupt is edge-triggered.

In contrast to constants and enumerations, the bit positions of the flags have a particular meaning and two flags can be combined effectively enabling both options whereas the combination of enumeration values or constants may not be defined. Bit positions that are not defined in the flag group are treated as zero.

As an example of where to use the flags use case we take the GICC CPU Interface flags as defined in the MADT Table of the ACPI specification.

Flag	Bit	Description
Enabled	0	If zero, this processor is unusable.
Performance Interrupt Mode	1	0 - Level-triggered,
VGIC Maintenance Interrupt Mode	2	0 - Level-triggered, 1 - Edge-Triggered
Reserved	3..31	Reserved

Syntax

```
flags name width "description" {
    position1 name1 "description1" ;
    ...
};
```

Fields

name the identifier of this flag group. Must be unique for all declarations.

width The width in bits of this flag group. Defines the maximum number of flags supported. This is one of 8, 16, 32, 64.

description description in double quotes is a short explanation of what the flag group represents.

name1 identifier of the flag. Must be unique within the flag group.

position1 integer defining which bit position the flag sets

description1 description of this particular flag.

Type

Flags with identifier *name* define the following type:

```
flag name;
```

Example

The example from the ACPI table can be expressed in Skate as follows:

```
flags CPUInterfaceFlags 32 "GICC CPU Interface Flags" {
    0 Enabled         "The CPU is enabled and can be used" ;
    1 Performance     "Performance Interrupt Mode Edge-Triggered " ;
    1 VGICMaintenance "VGIC Maintenance Interrupt Mode Edge-Triggered" ;
};
```

3.8.2 Constants

Constants provide a way to specify a set of predefined values of a particular type. They are defined in a constant group and every constant of this group needs to be of the same type.

Compared to flags, the combination of two constants has no meaning (e.g. adding two version numbers). In addition, constants only define a set of known values, but do not rule out the possibility of observing other values. As an example for this may be the vendor ID of a PCI Express device, where the constant group contains the known vendor IDs.

As an example where constants ca be used we take the GIC version field of the GICD entry of the ACPI MADT Table.

Value	Meaning
`0x00`	No GIC version is specified, fall back to hardware discovery for GIC version
`0x01`	Controller is a GICv1
`0x02`	Controller is a GICv2
`0x03`	Controller is a GICv3
`0x04`	Controller is a GICv4
`0x05-0xFF`	Reserved for future use.

Syntax

```
constants name builtintype "description" {
    name1 = value1 "description1" ;
    ...
};
```

Fields

name the identifier of this constants group. Must be unique for all declarations.

builtintype the type of the constant group. Must be one of the builtin types as defined in 3.5

description description in double quotes is a short explanation of what the constant group represents.

name1 identifier of the constant. Must be unique within the constant group.

value1 the value of the constant. Must match the declared type.

description1 description of this particular constant

Type

Constants with identifier *name* define the following type:

```
const name;
```

Example

The GIC version of our example can be expressed in the syntax as follows:

```
constants GICVersion uint8 "The GIC Version" {
    unspecified = 0x00 "No GIC version is specified" ;
    GICv1       = 0x01 "Controller is a GICv1" ;
    GICv2       = 0x02 "Controller is a GICv2" ;
    GICv3       = 0x03 "Controller is a GICv3" ;
    GICv4       = 0x04 "Controller is a GICv4" ;
};
```

3.8.3 Enumerations

Enumerations model a finite set of states effectively constants that only allow the specified values. However, in contrast to constants they are not assigned an specific value. Two enumeration values cannot be combined. As an example, the enumeration construct can be used to express the state of a device in the system which can be in one of the following states: *uninitialized, operational, suspended, halted.* It's obvious, that the combination of the states operational and suspended is meaning less.

Syntax

```
enumeration name "description" {
    name1 "description1";
    ...
};
```

Fields

name the identifier of this enumeration group. Must be unique for all declarations.

description description in double quotes is a short explanation of what the enumeration group represents.

name1 identifier of the element. Must be unique within the enumeration group.

description1 description of this particular element

Type

Enumerations with identifier *name* define the following type:

```
enum name;
```

Example

```
enumeration DeviceState "Possible device states" {
    uninitialized "The device is uninitialized";
    operational   "The device is operaetional";
    suspended     "The device is suspended";
    halted        "The device is halted";
};
```

3.8.4 Facts

The fact is the central element of Skate. It defines the actual facts about the system that are put into the SKB. Each fact has a name and one or more fields of a given type. Facts should be defined such that they do not require any transformation. For example, take the entries of an ACPI table and define a fact for each of the entry types.

Syntax

```
fact name  "description" {
    type1 name1 "description1" ;
    ...
};
```

Fields

name the identifier of this fact. Must be unique for all declarations.

description description in double quotes is a short English explanation of what the fact defines. (e.g. Local APIC)

type1 the type of the fact field. Must be one of the BuiltIn types or one of the constants, flags or other facts. When using facts as field types, there must be no recursive nesting.

name1 identifier of a fact field. Must be unique within the Fact group.

description1 description of this particular field

Type

Facts with identifier *name* define the following type.

```
fact name;
```

3.9 Documentation

The schema declaration may contain *section* and *text* blocks that allow providing an introduction or additional information for the schema declared in the Skate file. The two constructs are for documentation purpose only and do not affect code generation. The section and text blocks can appear at any place in the Schema declaration. There is no type being defined for documentation blocks.

3.9.1 Schema

The generated documentation will contain all the schemas declared in the source tree. The different schema files correspond to chapters in the resulting documentation or form a page of a Wiki for instance.

3.9.2 Text

By adding `text` blocks, additional content can be added to the generated documentation. This includes examples and additional information of the declarations of the schema. The text blocks are omitted when generating code. Note, each of the text lines must be wrapped in double quotes. Generally, a block of text will translate to a paragraph.

Syntax

```
text {
    "text"
    ...
};
```

Fields

text A line of text in double quotes.

3.9.3 Sections

The `section` construct allows to insert section headings into the documentation. A section logically groups the declarations and text blocks together to allow expressing a logical hierarchy.

Syntax

```
section "name"  {
    declaration;
    ...
};
```

Fields

name the name will be used as the section heading

declaration declarations belonging to this section.

Note, nested sections will result into (sub)subheadings or heading 2, 3, ... Namespaces will appear as sections in the documentation.

Chapter 4

Operations and checks on the AST

The following checks are executed after the parser has consumed the entire Skate file and created the AST.

4.1 Filename Check

As already stated, the name of the Skate (without extension) must match the identifier of the declared schema in the Skate file. This is required for resolving imports of other Schemas.

4.2 Uniqueness of declarations / fields

Skate ensures that all declarations within a namespace are unique no matter which type they are i.e. there cannot be a fact and a constant definition with the same identifier. Moreover, the same check is applied to the fact attributes as well as flags, enumerations and constant values.

Checks are based on the qualified identifier.

4.3 Type Checks

4.4 Sorting of Declarations

This requires generated a dependency graph for the facts etc.

Chapter 5

C mapping for Schema Definitions

For each schema specification, Skate generates ....

Abbrevations In all the sections of this chapter, we use the follwing abbrevations, where the actual value may be upper or lower case depending on the conventions:

SN The schema name as used in the schema declaration.

DN The declaration name as used in the flags / constants / enumeration / facts declaration

FN The field name as used in field declaration of flags / constants / enumeration / facts

In general all defined functions, types and macros are prefixed with the schema name SN.

Conventions We use the follwing conventions for the generated code:

- macro definitions and enumerations are uppercase.
- type definitions, function names are lowercase.
- use of the underscore `'_'` to separate words

just a header file (cf mackerel), or also C functions (cf. flounder)?

5.1 Using Schemas

Developers can use the schemas by including the generated header file of a schema. All header files are placed in the schema subdirectory of the main include folder of the build tree. For example, the schema `SN` would generate the file `SN_schema.h` and can be included by a C program with:

```
#include <schema/SN_schema.h
```

5.2 Preamble

The generated headerfile is protected by a include guard that depends on the schema name. For example, the schema `SN` will be guarded by the macro definition `__SCHEMADEF__SN_H_`. The header file will include the folling header files:

1. a common header `skate.h` providing the missing macro and function definitions for correct C generation.
2. an include for each of the imported schema devices.

5.3 Constants

For any declared constant group, Skate will generate the following:

Type and macro definitions

1. A type definition for the declared type of the constant group. The type typename will be `SN_DN_t`.
2. A set of CPP macro definitions, one for each of the declared constants. Each macro will have the name as in `SN_DN_FN` and expands to the field value cast to the type of the field.

Function definitions

1. A function to describe the value

   ```
   SN_DN_describe(SN_DN_t);
   ```

2. An snprintf-like function to pretty-print values of type SN_DN_t, with prototype:

   ```
   int SN_DN_print(char *s, size_t sz);
   ```

Do we need more ?

5.4 Flags

Type and macro definitions

1. A type definition for the declared type of the flag group. The type typename will be `SN_DN_t`.

Function definitions

1. A function to describe the value

   ```
   SN_DN_describe(SN_DN_t);
   ```

Do we need more ?

5.5 Enumerations

Enumerations translate one-to-one to the C enumeration type in a straight forward manner:

```
typdef enum { SN_DN_FN1, ... } SN_DN_t;
```

Function definitions

1. A function to describe the value

   ```
   SN_DN_describe(SN_DN_t);
   ```

2. A function to pretty-print the value

   ```
   SN_DN_print(char *b, size_t sz, SN_DN_t val);
   ```

5.6 Facts

Type and macro definitions

1. A type definition for the declared type of the flag group. The type typename will be `SN_DN_t`.

Function definitions

1. A function to describe the value

   ```
   SN_DN_describe(SN_DN_t);
   ```

2. A function to add a fact to the SKB
3. A function to retrieve all the facts of this type from the SKB
4. A function to delete the fact from the SKB

Provide some way of wildcard values. e.g. list all facts with this filter or delete all facts that match the filter.

5.7 Namespaces

Function definitions

1. A function to retrieve all the facts belonging to a name space

5.8 Sections and text blocks

For the `section` and `text` blocks in the schema file, there won't be any visible C constructs generated, but rather turned into comment blocks in the generated C files.

Chapter 6

Prolog mapping for Schema Definitions

Each fact added to the SKB using Skate is represented by a single Prolog functor. The functor name in Prolog consist of the schema and fact name. The fact defined in Listing **??** is represented by the functor `cpuid_vendor` and has an arity of three.

Chapter 7

Generated Documentation

Chapter 8

Access Control

on the level of schema or namespaces.

Chapter 9

Integration into the Hake build system

Skate is a tool that is integrated with Hake. Add the attribute `SkateSchema` to a Hakefile to invoke Skate as shown in Listing 9.1.

```
[ build application {
    SkateSchema = [ "cpu" ]
    ...
} ]
```

Listing 9.1: Including Skate schemata in Hake

Adding an entry for `SkateSchema` to a Hakefile will generate both header and implementation and adds it to the list of compiled resources. A Skate schema is referred to by its name and Skate will look for a file ending with `.Skate` containing the schema definition.

The header file is placed in `include/schema` in the build tree, the C implementation is stored in the Hakefile application or library directory.

References