A user's guide for SWI-DSP 1.0

July 31, 2008

1 Intro

SWI-DSP is a logic-based framework that defines a declarative front-end to create DSP worflows. It is implemented in SWI-Prolog and interfaces to foreign languages (C, C++ and Java).

In this document we overview some Prolog basics and provide some hints for users to get started to SWI-DSP.

2 Prolog

Prolog is a programming language that implements a predicate logic (close to 1st order logic). A Prolog program is composed by different clauses: facts, rules and queries.

- facts are ground truth assertations. (e.g. author_name('David').)
- rules are structured as Header:-Body. (e.g. grandfather(X, Y):- father(X, Z), father(Z, Y).
- queries do not have a header :-Goal. (e.g. :-grandfather('David', X)

Queries are solved by backtracking collecting all solutions verifying the goal.

These clauses are composed of predicates and terms. Predicates are relationships among n elements and are specified by their name and their arity (e.g. grandfather/2). Each one of the elements are Prolog terms that can be primitive types (numbers, atoms, strings...), lists, compound terms or variables.

SWI-Prolog source code does not include further information by itself about predicates, but the source documentation gives the following information:

The instantiation pattern of a predicate determines the different ways to call correctly a Prolog predicate.

- + The term must be instantiated
- - The term must be passed for unification (must be a variable)
- ? The term can be either instantiated or passed for unification

- : The term is a meta-argument. implies +
- @ The term is not further instantiated
- ! The argument contains a mutable structure

SWI-Prolog does not include modes and types in its core. Types determine structure of the terms. Modes specify the determinism of the predicate, this means, the behaviour of the predicate when it is instantiated.

- det: the predicate succeeds just once
- semidet: the predicate may fail or succeed just once
- multi: the predicate succeeds more than once
- nondet: the most flexible. Does not constrain how many times the predicate has to succeed.

We can also declare Prolog predicates establishing functional relationships between terms. This capability allows us to wrap calls to external APIs and processes under Prolog predicates (e.g. decode(+AudioFile, -AudioSignal)).

Prolog programs can be developed with a declarative or procedural meaning. Procedural meaning is allowed in Prolog but it does not exploit all its capabilities. Proper Prolog programs should be built with a declarative meaning, this means, that you can evaluate the truth of an expression.

3 Modules

We divide SWI-DSP in different modules. Each module is designed to wrap a specific API or to perform a bunch of tasks/processes.

Each module can be loaded by launching pl from the home module folder and loading the module from the prompt:

?- use_module('swiaudio').

All the components can be loaded by launching the script in the swi-dsp folder.

4 swidata

The first module of SWI-DSP is called swidata. This module defines a new type for large data. It wraps large data objects under atomic identifiers (e.g. '__data_45') and defines operations that deal with these universal references. The module also provides builtin predicates to check the status of the ids storage. Let us see some examples:

:-blobs(X)

X = 239 We have 239 data objects in dynamic memory in this session.

:-busy_blob('__data_34')

true This identifier is a reference to a data object which is in memory at the moment

:-blob_size('__data_45', L)

L = 34890 This data object has 34890 samples (keep in mind that each data object is 1-D)

:-blob_frame('__data_4321', 0, 345, F)

F = '__data_4323' This new id is a reference to a frame (sample 0 to 345) of the original data object

This BLOBIDs are used in SWI-DSP to represent large data objects within higher level terms such as signals or frames.

5 swiaudio

A module to deal with audio data sources. The main predicate is decode/2:

:-decode('myfile.mp3', Signal)

Signal = signal(44100, ['__data_0', '__data_1') This compound term wraps the sample rate and the PCM objects of the signal encoded in an audio file.

The system supports: mp3, ogg, wav and m4a (and some equivalent formats).

:-get_frame(signal(44100, ['__data_0' ,'__data_1']), 0, 44100, Frame), get_timestamp(Frame, Time).

Frame = frame(44100, 0, ['__data_2', '__data_3']) Time = timestamp(0.0, 1.0)

These terms represent an audio frame and the timestamp given the framing parameters.

This module is loaded by other DSP modules to provide audio data for further analysis.

6 swivamp

This module wraps the VAMP Plugins API for extraction of audio features. The module is subdivided in two modules: vamp and vamp_transform. The former is a communication layer with Vamp APIs so we can control the plugin lifecycle and described in the VAMP source. Some predicates of this module are:

:-vamp_plugin_system(?PluginKey) will check the availability of a plugin given its key (see vamp doc) or return the plugins in your local system

:-vamp_pluginParameters(+PluginKey, -ListOfParameters) returns a list containing the parameters identifier for the given plugin key

:-vamp_plugin_for(?PluginKey, ?Output, ?Index) checks the relationships between the different outputs of the plugins and their index between the list of parameter descriptors

:-vmpl_process_block_framing(+Plugin, +Signal, +StartSample, +Size, +ListOfOutputs, -ListOfFeatures) wraps the process call of a working plugin (this plugin reference is not a key but the plugin itself represented by an atomic identifier like '__plugin::vamp_3'). Given the input and the parameters for framing and

the index of the frame, this predicate chunks the input to obtain the corresponding frame it, process it calling vamp and returning the queried features. This predicate has to be inserted in a higher extraction rule that process the whole signal controling parameters, framing and remaining features (vamp plugins can return features that have been stored during the whole process, see vamp doc).

This module, as the reader may notice, has to namespace keywords: vamp_ and vmpl_. The first one is for predicates that accept a plugin key as reference to a plugin, therefore they are general predicates that can be called interactively. The second one is used to point out runtime predicates that require an atomic id for the plugin instance and has to be included in a higher level rule that deals with them properly.

The latter module provides a higher level of predicates that allow us to execute vamp transforms hidding the real calls to the API. This module is built on top of routines written with the vamp module predicates (procedural meaning).

:-decode('myfile.ogg', Input), transform(Input, 'libqm-vamp-plugins', 'qm-tempotracker', [beats], FeatureList).

FeatureList = [[[feature(tempo, timestamp(25.4, 0), _)]]—...] This is a list of tempo features for the given input and default options (parameters, programs or specific step and block sizes)

Each feature functor (or compound term) is a Prolog term capturing the Vamp::Feature type. Each of these terms includes the feature identifier, the timestamp of the feature (start and duration) and a Prolog list containing the bins of the feature if any (it can be changed to return BLOBIDs instead).

7 SWI-DSP and HENRY

SWI-DSP can be used from the Prolog prompt or loaded by any progam as a set of builtins. However, this approaches are not as useful as the user may expect because computations can not be saved from session to session. Moreover, we also need to "table" predicates so the relationships between input, process and output is made persistent. Thus, we don't need to re-compute features and we are able to check the truth of an expression (declarative meaning). For example if we table the predicate decode('mysong', L), the result will be linked to the predicate and the input, so we can pass later on the output to check if that Signal term actually is encoded by our audio file.

Henry offers these mechanisms from a Notation3 front-end (see Henry doc). Notation 3 allows us to express facts (rdf statements) and rules (named graphs) and SPARQL becomes the query language that we can use instead of Prolog using web ontologies to identify the predicates.

Each Prolog predicate is declared as a Henry builtin (henry/dsp-builtins). Each builtin is identified by an URI, for example, the decoding predicate is now called using http://purl.org/ontology/dsp/decode. Note that any builtin has to be instantiated as an RDF statements (resource predicate resource), so the builtin must hide a x/2 predicate with an input context and a term wrapping

the output. The dsp builtins available in Henry are quite representative. If the predicate triggers an expensive computation it is better to declare it as tabled predicate, so the results will be made persistent and the data will be dumped to disk and retrieved afterwards.

builtin:decode(literal(AudioFileURI), [literal(Sr), Data]):- aspl_decode(AudioFileURI, signal(Sr, Data)).

Henry also accepts n3 input to perform entailment and reason about attached data. Some examples can be found at henry/dsp-n3. For example for a decoding process:

{?a dsp:decode (?sr ?signals)} =¿ { ?signals a mo:Signal; mo:time [a tl:Interval; tl:timeline _:tl.] . }

SWI-DSP provides Henry with processing services that is able to link several sources of data belonging to different knowledge domains.