Dogelog Player Host

Version 1.3.2, March 29, 2025

XLOG Technologies AG

Dogelog Prolog

Dogelog Player 1.3.1

Host Interface

Author: XLOG Technologies AG
Jan Burse
Mittlere Mühlestrasse 2
8598 Bottighofen
Switzerland

Date: February 26, 2025

Warranty & Liability

Rights & License

Trademarks

Table of Contents

Change history

Jan Burse, December 17, 2023, 0.1:

- Initial version, spin off from reference and frequent document.

Jan Burse, May 20, 2024, 0.2:

- New host commons section introduced.

1 Introduction

t.b.d.

- **Dogelog Embedding:** t.b.d.
- **Dogelog Albufeira:** The files "machine" and “store” are already genuinely host language and need not be touched.
- **Dogelog Native:** The Dogelog Player allows to code libraries in the host programming language. Dynamic loading of native libraries is provided.
- **Dogelog Transpiler:** Concerning the files "compile" and "loader", these files need to be cross-compiled into the Albufeira embedding to build the Dogelog player.

2 Dogelog Embedding

t.b.d.

- **t.b.d.:** t.b.d..

2.1 t.b.d

3 Dogelog Albufeira

The files "machine" and “store” are already genuinely host language and need not be touched. The define the following parts of the Dogelog Player:

- **Folder "lang":** We describe the representation of Prolog terms in general and Prolog atomics in particular for each Dogelog Player target in more detail.
- **Folder "machine":** The instruction set of the Dogelog player is rather minimal. The same instruction set is used in cross compilation and at runtime.
- **Folder "store":** Terms are cooperatively garbage collected with the host. The knowledge base provides non-nested rollback.

3.1 Folder "lang"

We describe the representation of Prolog atomics for each Dogelog Player target in more detail. The issues that have to be resolved vary from target to target. We find different approaches to number shaping or atom equality. The overall goal is to have all targets behave exactly the same from the Prolog interpreter viewpoint.

- **Section "term":** The term API provides the creation and basic operations of Prolog terms. Only Prolog variables and Prolog compounds have their own classes.
- **Section "javascript":** t.b.d..
- **Section "python":** t.b.d..
- **Section "java":** t.b.d..

Section "term"

The term API provides the creation and basic operations of Prolog terms. Currently there are no API calls to access the content of Prolog terms, since they are mapped to classes of the host language and the API client can directly access the fields of these classes. This gives more speed in the implementation of built-ins through the native API.

The mapping to classes is lean, in that only Prolog variables and Prolog compounds have their own classes. Otherwise, the objects form the host language are used to represent Prolog atomics. Among the Prolog atomics are Prolog references, which are everything that is neither a Prolog atom or a Prolog number.

The following term API calls are provided:

deref(T): (host language)
: Return the dereferencing of the term T.

copy_term(T): (host language)
: Return a copy of the term T.

is_variable(T): (host language)
: Check whether the object T is a variable.

is_compound(T): (host language)
: Check whether the object T is a compound.

is_atom(T): (host language)
: Check whether the object T is an atom.

is_number(T): (host language)
: Check whether the object T is a number.

is_integer(T): (host language)
: Check whether the object T is an integer.

is_float(T): (host language)
: Check whether the object T is a float.

unify(S, T): (host language)
: Determine whether the two terms S and T unify.

equal_term(S, T): (host language)
: Determine whether the two terms S and T are syntactically equivalent.

compare_term(S, T): (host language)
: Return the syntactic relationship between the two terms S and T.

narrow_float(T): (host language)
: Return the Prolog number T narrowed to a float.

Section "javascript"

The specifics mainly deal with the number blending that is applied in the Dogelog Player for the JavaScript platform. Whereas Prolog atoms are simply mapped to JavaScript strings, the mapping of Prolog integers is towards JavaScript smallint and JavaScript bigint. Exclusion of NaN and infinites from Prolog floats, is done by mapping values to errors.

The following mapping from Prolog to JavaScript is used:

```
Prolog Atomic
 +--- Prolog Atom
       +--- JavaScript string
 +--- Prolog Number
       +--- Prolog integer
             +---- JavaScript smallint in -94906266..94906266
             +---- JavaScript bigint otherwise
       +--- Prolog float
             +---- JavaScript bigint in -94906266..94906266
             +---- JavaScript number otherwise
 +--- Prolog 0rNone Reference
       +--- JavaScript null
 +--- Prolog 0rFalse Reference
       +--- JavaScript false
 +--- Prolog 0rTrue Reference
       +--- JavaScript true
```

The following JavaScript specific calls are provided:

is_bigint(T): (host language)
Check whether the object T is a JavaScript bigint.
norm_smallint(T): (host language)
Return the JavaScript smallint T normalized to a Prolog integer.
norm_bigint(T): (host language)
Return the JavaScript bigint T normalized to a Prolog integer.
norm_float(T): (host language)
Return JavaScript float T normalized to a Prolog number.
widen_bigint(T): (host language)
Return the Prolog integer T widened to a JavaScript bigint.
char_count(C): (host language)
Return the 16-bit char count of a Unicode code point C.

Section "python"

The specifics mainly deal with a number blending problem on the Python platform. It turns out that unlike JavaScript, which has a type sensitive equality (===)/2, we didn't find such a thing in Python. Therefore, we need to be careful in syntactic comparison where the ISO core standard requires that the float number 1.0 is different from the integer number 1.

The following mapping from Prolog to Python is used:

```
Prolog Atomic
 +--- Prolog Atom
       +--- Python string
 +--- Prolog Number
       +--- Prolog integer
             +---- Python int
       +--- Prolog float
             +---- Python float
 +--- Prolog 0rNone Reference
       +--- Python None
 +--- Prolog 0rFalse Reference
       +--- Python False
 +--- Prolog 0rTrue Reference
       +--- Python True
```

The following Python specific calls are provided:

atomic_equal(S, T): (host language)

Determine whether the two atomics S and T are equal.

Section "java"

The specifics mainly deal with the number blending that is applied in the Dogelog Player for the Java platform. Whereas Prolog atoms are simply mapped to Javat String class, the mapping of Prolog integers is towards Java Integer class and JavaScript BigInteger class. Exclusion of NaN and infinites from Prolog floats, is done by mapping values to errors.

The following mapping from Prolog to Java is used:

```
Prolog Atomic
 +--- Prolog Atom
       +--- Java String
 +--- Prolog Number
       +--- Prolog integer
             +---- Java Integer
             +---- Java BigInteger
       +--- Prolog float
             +---- Java Double
 +--- Prolog 0rNone Reference
       +--- Java null
 +--- Prolog 0rFalse Reference
       +--- Java Boolean.FALSE
 +--- Prolog 0rTrue Reference
       +--- Java Boolean.TRUE
```

The following Java specific calls are provided:

is_bigint(T): (host language)
Check whether the object T is a Java BigInteger.
norm_smallint(T): (host language)
Return the Java long T normalized to a Prolog integer.
norm_bigint(T): (host language)
Return the Java BigInteger T normalized to a Prolog integer.
norm_float(T): (host language)
Return Java float T normalized to a Prolog number.
widen_bigint(T): (host language)
Return the Prolog integer T widened to a Java BigInteger.

3.2 Folder "machine"

The instruction set of the Dogelog player is rather minimal. The same instruction set is used in cross compilation and at runtime.

- **Section "embedding":** We do not combine different instruction streams. Instead, we have host language objects for Prolog clauses and Prolog goals.
- **Section "instruction":** Depending whether an instruction is found in the head or in the body, the behaviour of the instruction is differently.
- **Section "internals":** The Dogelog player keeps the number of Albufeira instructions low. It provides low-level control flow instructions via built-ins.
- **Section "shadow":** During cross compilation the Prolog texts receives the same style check as when they would be consulted by the Dogelog player..

Section "embedding"

We do not combine the unify instruction stream with the build instruction stream. Instead, we have host language objects for Prolog clauses and Prolog goals. The host language objects record further attributes to be able to assert or execute. Host language functions then allow passing these objects to the Dogelog player.

The host language routines make_defined() and make_special() yield anonymous predicates. These can then be used as an argument to add() so that a named predicate in the knowledge base results. Alternatively, they can be used as a functor of a compound or as an atom. They then become executable without registering them.

The cross-compiler or runtime generate the following objects and calls:

new Clause(S, H, B, R, K): (host language)
: The constructor creates an object representing a clause with variable count S, head instructions H, body instructions B, cut variable index R and index key K.

add(F, A, C): (host language)
: The function adds the clause or anonymous predicate C to the knowledge base for the predicate indicator F/A.

new Goal(S, B): (host language)
: The constructor creates an object representing a goal with variable count S, body instructions B.

run(G): (host language)
: The function executes the goal G, cuts away its choice points and undoes its bindings. If the goal fails, it throws a new error. If the goal throws an error, it re-throws this error. If the goal succeeds, the built-in succeeds.

make_defined(L): (host language)
: The function returns an anonymous predicate for the given clauses L.

make_special(P): (host language)
: The function returns an anonymous predicate for the special P.

Section "instruction"

Depending whether an instruction is found in the head or in the body, the behaviour of the instruction is differently. In the body, instructions do build a Prolog terms. In the head, instructions try to unify Prolog terms and they might fall back into the Prolog building mode. Instruction streams are represented as host language objects.

In build mode, to allow clause sets inlining, the instructions the functor of a callable is allowed to be an anonymous predicate. The corresponding goal is solved by directly branching into the clause set of the anonymous predicate. Further the functor can be a cache node which speeds up the lookup of a predicate indicator.

The cross-compiler or runtime generate the following objects:

new Place(W), W =-1: (host language)
: In building mode creates a fresh variable. In unify mode does nothing.

new Place(V), V < -1: (host language)
: In building mode creates a fresh variable for the display at index W where W = (-V)-2. In unify assigns the unify argument to the display at index W.

new Place(W), W >= 0: (host language)
: In building mode reuses the display at index W. In unify mode unifies the unify argument with the display at index W.

new Skeleton(F, L): (host language)
: In building mode creates a new undefined compound with functor F and arguments L, and opens a new arguments context. In unify mode defers the creation and performs initial specialized compound unification, with the unify argument.

new Cache(F): (host language)
: The constructor creates a cache node for the functor F.

otherwise C:
: In building mode gives the embedded constant C, which is allowed to be a ground term and need not be atomic only. In unify mode performs unification of the embedded constant C with the unify argument.

Section "internals"

The Dogelog player keeps the number of Albufeira instructions low. It provides low-level control flow instructions via built-ins. The Dogelog player allows to register deterministic and non-deterministic built-ins in the form of host language functions. A non-deterministic built-in creates a choice point as a side effect, registering another host language function.

The below built-ins are internal only and do not check their arguments. The redo argument should be a variable. The sequence goal list is just a Prolog list of Prolog terms each representing a goal. The alternative variant list consists of variant/1 pairs, the first component of the pair a redo option and the component a sequence goal list.

The cross-compiler or runtime generate the following objects:

'$CUT'(R): internal only
: The built-in removes the choice points up to R and succeeds.

'$MARK'(R): Internal only
: The built-in binds R the top choice point.

'$SEQ'(O, L): internal only
: If the Prolog term O has the form just(R) bind R the top choice point. Otherwise, do nothing. The built-in then sequentially adds the goals L to the continuation.

'$ALT'(L): internal only
: The built-in alternatively adds the variants L to the continuation and succeeds.

'$YIELD'(R): Internal only
: The built-in stops the interpreter loop with return value R.

Section "shadow"

The cross compiler only accepts conservative Prolog texts without arbitrary goals directives. During cross compilation the Prolog texts receives the same style check as when they would be consulted by the Dogelog player. Among the style checks are for example the singleton check, but we also do check other properties concerning the declaration of predicates.

The cross compiled Prolog text will not anymore repeat the style checks and will have the recognized directives omitted or replaced by more simpler instructions. To be able to perform the style checks and since the cross compiler is self-hosting, it keeps track of a shadow dynamic database of some Prolog system meta information:

```
cross_loading/1
cross_including/2
cross_currpred/3
cross_predprop/3
cross_lastpred/2
```

3.3 Folder "store"

Terms are cooperatively garbage collected with the host. The knowledge base provides non-nested rollback.

- **Section "trailing":** The execution environment of Dogelog Player provides the state to drive a Prolog engine that is based on an iterative execution.
- **Section "trimming":** The Prolog interpreter regularly performs environment trimming to help the host language garbage collector.
- **Section "meta":** There are a couple of dynamic predicates that define the Dogelog player runtime system.
- **Section "unicode":** The Dogelog player accepts Prolog texts in Unicode code points based on Unicode general categories and numeric values.

Section "trailing"

The execution environment of Dogelog Player provides the state to drive a Prolog engine that is based on an iterative execution of either call ports of new goals, or if there was a failure redo ports of goals that were successful and left a choice point. The engine state responsible for this execution is the current continuation and the current choice point.

To deal with variable binding during unification the current trail is used. Choice points can keep pointers into the trail and allow to undo sections of variable binding. Because variables might become irrelevant during execution, the Prolog specific garbage collection called environment trimming aims at compacting the trail and thus saving memory.

Prolog variables store some colouring flags to manage generational garbage collection, and some serno which is used in lexical comparison. The serno once chosen during the creation of Prolog variable doesn't change during environment trimming and therefore allows for certain algorithms, that other Prolog systems might not support.

The cross-compiler or runtime generate the following objects and calls:

new Variable(): (host language)
Create a Prolog variable.
new Compound(F, A): (host language)
Create a Prolog compound with functor F and arguments A.
new Choice(H, D, A, M): (host language)
Create a choice point with handler H, data D, index A and trail mark M.
get_cont(): (host language)
The function returns the current continuation.
cont(C): (host language)
The function sets the current continuation to C.
more(C): (host language)
The function sets the current choice point to C.
get_mark(): (host language)
The function returns the current trail.
unbind(M): (host language)
The function undoes variable binding up to the given trail mark.

Section "trimming"

The Prolog interpreter performs regularly a check point. The frequency is controlled by the parameters GC_MAX_INFERS, which has been chosen so that a rate of ca. 60 Hz is archived. If in asynchronous mode, the Prolog interpreter yields during the check point allowing it to be non-blocking in a single threaded host.

Further during this check point we decided whether to invoke a major garbage collection or a minor garbage collection. A major garbage collection is a full garbage collection where we also update the serno interval and counter. A minor garbage collection is a generational garbage collection without updating the serno interval and counter.

Since major and minor garbage collection are triggered very defensively, they might be not triggered at all, so that the serno counter might exhaust. We therefore have a further frequency parameter GC_MAX_DIRTY, which has been chosen so that a rate of ca. 1/60 Hz is archived. It will unconditionally trigger a major garbage collection.

The values are currently:

```
/* JavaScript, nodeJS */
GC_MAX_INFERS = 92000
GC_MAX_DIRTY = 331200000

/* JavaScript, Browser */
GC_MAX_INFERS = 61000
GC_MAX_DIRTY = 219600000

/* Python */
GC_MAX_INFERS = 55000
GC_MAX_DIRTY = 198000000

/* Java */
GC_MAX_INFERS = 200000
GC_MAX_DIRTY = 720000000
```

Section "meta"

The Dogelog player handling of predicates and clauses has evolved. It went from the ability to only rollback monotonic changes caused by dynamic/1, assertz/1 and assertz/2, to the rollback of non-monotonic changes as well caused by abolish/1 and retract/1. This also affects the dynamic predicates that define the Dogelog player runtime system.

To make the Prolog text loader task aware and to implement a quasi-parallel mutex around ensure_loaded/1 the built-in shield/1 is used. Temporary meta data during Prolog text loading such as sys_including/3 and sys_lastpred/3 have a task parameter, making the dynamic predicate task local.

```
sys_op/4
sys_source/3
sys_srcprop/2
sys_predprop/3
sys_lastpred/3
sys_including/3
sys_emulated/2
```

The above predicates are suppressed during listing/[0,1].

Section "unicode"

The Dogelog player accepts Prolog texts in Unicode code points based on Unicode general categories and numeric values. We rolled our own Unicode database in that we compressed the desired Unicode character properties. The following utilities written in Java are provided to generate the Unicode database:

- **Unicode:** The base class for the Unicode database generators.
- **Compress:** Generate the Unicode database for the JavaScript target.
- **CompressPy:** Generate the Unicode database for the Python target.

The output of these utilities is to be placed in the corresponding "unicode" file. It will provide the Unicode database in compressed form to reduce the file footprint and allow delivery over the wire. The compression is possible since the Unicode database needs only to provide general category and numeric value.

The Unicode database provides the following host language calls:

code_type(C): (host language)
: Retrieve a Unicode code point general category. The general category is an integer in the range 0 to 16 and 18 to 30.

code_numeric(C): (host language)
: Retrieve a Unicode code point numeric value. The numeric value is an integer in the range -1 or 0 to 36.

4 Dogelog Native

The Dogelog Player allows to code libraries in the host programming language. To implement such libraries, we provide a native API that exposes core routines. The end-user can use both include/1 and ensure_loaded1 to load native libraries. The loading and initialization mechanism is determined by its file extension.

- **Folder "entry":** This section gives some convenience entry points to build applications that embed the Prolog interpreter.

- **Folder "liblets":** The Dogelog Player provides dynamic loading of native libraries. The end-user can use both include/1 and ensure_loaded1 to load native libraries.

- **Folder "api":** We provide a native API that exposes core routines. The native API is available for the JavaScript, the Python and the Java platform.

- **Folder "cross":** Cross compilation is mainly used for bootstrapping the Dogelog Player itself. But it can be also useful to speed up the loading of Prolog texts.

4.1 Folder "entry"

This section gives some convenience entry points to build applications that embed the Prolog interpreter. Currently one convenience allows defining Prolog predicates and vice versa. We also allow unattended and attended queries.

- **Section "console":** The convenience provides both unattended and attended queries, suitable for batch or online processing.
- **Section "browser":** The Dogelog player version that targets JavaScript can be run from within nodeJS or inside a browser, the later has additional calls.
- **Section "theatre":** The Dogelog player facilitates calling Prolog predicates from within the host language.
- **Section "foreign":** The convenience facilitates defining deterministic Prolog predicates that call the host language.

Section "console"

The basic API provides both unattended and attended queries, suitable for batch or online processing. The function init() need to be only called once. The user database is organized in stages and partitions, which have the values -1 and “system” during start-up. The init() function will set the current stage to 0 and the current partition to “user”.

The following console calls are provided:

init(): (host language)
Initializes the Prolog system.

Section "browser"

The Dogelog player version that targets JavaScript can be run inside a browser and from within nodeJS. When run inside the browser the Dogelog player keeps track of a cursor that points to a DOM element. This is for example currently used in the markup library but can be also directly accessed through the native API.

Our original idea was to replicate Knuth's idea of literate programming for the web. Diverting slightly from the batch processing approach, we view HTML pages with a simple Prolog code embedding as pre-elements as the primary source. We provide plain and colorized notebook instrumentation via calling notebook() respectively notebook_async().

The following browser specific calls are provided:

set_caret(C): (host language)
Sets the input data field to the DOM element C.
set_cursor(C): (host language)
Sets the output and error data field to the DOM element C.
notebook(): (host language)
Setup the Prolog interpreter and run the notebook plain.
notebook_async(): (host language)
Setup the Prolog interpreter and run the notebook colorized.

Section "theatre"

The Dogelog player facilitates calling Prolog predicates from within the host language. The host calls perform() and perform_async() are usually called with a ground term. The client creates via the ground term using the host language data types and the Prolog term constructors. The host call post() sends a signal and interrupts the “main” task.

The following theatre calls are provided:

post(M): (host language)
Post the message M to the Prolog interpreter.
perform(G): (host language)
Run the goal G once without retaining variable bindings, returning success, failure and exceptions. The goal is run with auto-yield disabled and promises are not accepted.
perform_async(G): (host language)
Run the goal G once without retaining variable bindings, returning success, failure and exceptions. The goal is run with auto-yield enabled and promises are accepted.

Section "foreign"

The Dogelog player facilitates defining Prolog predicates that call the host language. The host language call register() adds a host language function by a given predicate indicator. The flag FFI_FUNC determines whether the functions return value should be ignored or not. After registration, the designated predicate can be called from within Prolog.

The following host language calls are provided:

FFI_FUNC: (host language)
Option that says the host language function has a return value.

register(F,N,J,K): (host language)
Add a host language function J with options K by the predicate indicator F/N to the knowledge base. The registration is staged, so clear() will remove.

4.2 Folder "liblets"

The Dogelog Player provides dynamic loading of native libraries. The end-user can use both include/1 and ensure_loaded/1 to load native libraries. The loading and initialization mechanism is determined by its file extension. Calling include/1 multiple time will initialize the native library multiple times, whereas ensure_loaded/1 does the same only on first encounter.

- **Section "neck":** As an optimization Dogelog player can directly perform neck goals if they belong to a deterministic built-in that does not modify the continuation.

- **Section "mjs":** JavaScript module files are detected by the extension “.mjs”. They are loaded asynchronously and dynamically.

- **Section "py":** Python source files are detected by the extension “.py”. They are loaded synchronously and dynamically

- **Section "class”:** Java Classes are first detected as source by the extension “.java” and then as bytecode by the extension “.class”.

Section "neck"

As an optimization the system can directly perform neck goals if they belong to a deterministic built-in that do not modify the continuation. Unlike specials, which are registered via make_special(), they need to be registered via make_check(). Further evaluable foreign functions can be registered via make_arithmetic().

This new neck API is provided in the form of exec_build(), exec_unify() and exec_eval() that should be used to implement deterministic built-in and evaluable functions. They depend on a state in the form of the current variable display and they are order dependent, in that they should be execute along the arguments in-order and without omissions.

The following neck API calls are provided:

make_check(P): (host language)
: The function returns an anonymous predicate for the deterministic built-in P.

make_arithmetic(P): (host language)
: The function returns an anonymous predicate for the deterministic built-in P.

exec_build(A): (host language)
: Return the dereferencing of the argument A.

exec_unify(A, T): (host language)
: Determine whether the argument A and the term T unify.

exec_eval(A): (host language)
: Return the evaluation of the argument A.

Section "mjs"

JavaScript module files are detected by the extension ".mjs". They are loaded asynchronously and dynamically by using the JavaScript function import(). The convention is currently that after loading the file, the Dogelog player will call a main() function of the loaded script, which needs to be exported. This leads to the following idiom:

Example:

```
import {add, make_special,
   ...} from "../nova/core.mjs";
...
export function main() {
   add("<name>", <arity>, make_special(<func>));
   ...
}
```

In the above the native API is made visible through some import. The native API is then used to register some special function by predicate indicator. The idea is that loaded scripts have exactly these side effects and do extend the Prolog interpreter by new functionality. The Dogelog player can undo register through its rollback mechanism.

Section "py"

Python source files are detected by the extension “.py”. They are loaded asynchronously and dynamically by using the Python library importlib.import_module and asyncio.to_thread. The convention is currently that after loading the file, the Dogelog player will call a main() function of the loaded script, which needs to be present. This leads to the following idiom:

Example:

```
from nova.core import (add, make_special,
    ...)
...
def main():
   add("<name>", <arity>, make_special(<func>))
   ...
```

In the above the native API is made visible through some from import statement. The native API is then used to register some special function by predicate indicator. The idea is that loaded scripts have exactly these side effects and do extend the Prolog interpreter by new functionality. The Dogelog player can undo register through its rollback mechanism.

Section "class"

Java Class are first detected as source by the extension “.java” and then as bytecode by the extension “.class”. They are loaded asynchronously and dynamically. The convention is currently that after loading the file, the Dogelog player will call a main() function of the loaded script, which needs to be present. This leads to the following idiom:

Example:

```
import nova.Store;

public final class bitlib {

   public static void main() {
     Store.add"<name>", <arity>, Special.make_special(<func>));
   }

}
```

Unlike for JavaScript and Python, we don’t have a Façade class for Java, that can be used for unqualified import of the native API. In this idiom we see qualified calls such as Store.add() etc.. The idea is again that loaded scripts have exactly these side effects and that the Dogelog player can undo register through its rollback mechanism.

4.3 Folder "api"

The Dogelog Player allows to code libraries in the host programming language. To implement such libraries, we provide a native API that exposes core routines.

- **Section** "**engine**"**:** Tasks not only have their own continuation, trail and choice points, the streams are accessible via the engine object.
- **Section** "**stream**"**:** The stream API provides access to objects representing input/output streams for custom implementations beyond the standard.
- **Section** "**stage**"**:** The clauses and predicates of the user-database are marked with a stage. The stage is used to provide progressive transactions that can be rolled back.
- **Section** "**markup**"**:** In the case of a browser environment the corresponding output and error streams point to DOM elements.

Section "engine"

Dogelog Player allows registering special foreign function foreign functions. A special foreign function has access to the execution environment of the Dogelog Player and can be used to code deterministic and non-deterministic predicates. To realize a non-deterministic predicate the special foreign function leaves a choice point.

To simplify task switching a part of the engine state is stored in an engine object. Task switching happens when a non-main task gets its execution share, and the main task is swapped in and out into a the task context. Tasks not only have their own continuation, trail and choice points, the streams are accessible via the engine object.

The following engine API calls are provided:

get_engine(): (host language)
: The function returns the current engine.

get_ctx(): (host language)
: The function returns the current task.

make_error(M): (host language)
: Create an error term from the message M.

check_nonvar(T): (host language)
: Assure that the object T is a nonvar.

check_atom(T): (host language)
: Assure that the object T is an atom.

check_number(T): (host language)
: Assure that the object T is a number.

check_integer(T): (host language)
: Assure that the object T is an integer.

check_callable(T): (host language)
: Assure that the object T is a callable.

check_atomic(T): (host language)
: Assure that the object T is atomic.

Section "stream"

The stream API provides access to the objects representing text input/output streams. This is to allow custom implementations beyond the standard. The input reader transparently handles ‘\n’, ‘\r\n’ and ‘\r’ as newline whereby also counting line numbers. The output writer does buffer before flushing and keeps track of the last written character.

Th input call-back is a function that takes no argument and returns a string. Returning an empty string indicates that end-of-stream has been reached. The output call-back is a function that takes a string and that doesn’t return anything. In both cases, to cater for call-backs that depend on some stream, appropriate host closures need to be created.

The following term API calls are provided:

new Source(T, R, C, D): (host language)
Create a text input with initial text T, call-back R, call-back C and data D.
new Sink(S, N, C, D): (host language)
Create a text output with call-back S, call-back N, call-back C and data D.
check_source(T): (host language)
Assure that the object T is a text input.
check_sink(T): (host language)
Assure that the object T is a text output.

Section "stage"

From its inception the Dogelog player had a concept of a stage giving the system the stage number -1 and the user the stage numbers 0, 1, 2, etc... The basic API calls clear() and consult() could then work accordingly. This was then enhanced to a further basic API call set_stage() which allowed to partition the user space into multiple stages.

The clauses and predicates of the user-database are marked with a stage. The stage is used to provide progressive transactions that can be rolled back. The rollback covers both the user-database clauses and predicates. For example, an abolish/1 of a dynamic predicate can be undone. The stage API provides access and modification of the stage.

The following stage API calls are provided:

clear(): (host language)
Resets the Prolog database of the current stage.
set_stage(S): (host language)
Sets the current stage.
get_stage(): (host language)
Retrieves the current stage.

Section "markup"

A Prolog system maintains a state through their input, output and error streams. In the case of a browser environment the corresponding output and error streams point to DOM elements, which can be simultaneously set via set_cursor(). There are also factory predicates in library(markup) to create more such markup streams.

Markup streams act on their cursor elem as defined in library(markup):

Prolog	JavaScript
write('foo < bar')	elem.insertAdjacentText("beforeend", "foo < bar")
tag('&')	elem.insertAdjacentHTML("beforeend", "&")
tag('<foo>')	elem.insertAdjacentHTML("beforeend", "<foo></foo>"); elem = elem.lastElementChild
tag('</foo>')	elem = elem.parentElement
tag('<foo/>')	elem.insertAdjacentHTML("beforeend", "<foo/>")

Further DOM actions are currently found in library(react):

Prolog	JavaScript
clear	elem.innerHTML = “”
goto('k7')	elem = document.getElementById(“k7”)
bind('click', E, foo(E))	elem.addEventListener("click", [E]>>foo(E))
listen('click', E, foo(E), [block(true)])	let prom = waitForEvent(elem, "click", [E]>>foo(E)) await prom

4.4 Folder "cross"

Cross compilation is mainly used for bootstrapping the Dogelog Player itself. But it can be also useful to speed up the loading of Prolog texts. It should be noted that the bootstrapping only covers the cross compilation of the Novacore into the target formats. The various libraries are not cross compiled, and deployed as Prolog texts.

- **Section "util":** There are ready made scripts to cross compile the Dogelog Player into module form and into canned form.
- **Section "player":** This cross compiler generates a JavaScript text.
- **Section "playerpy":** This cross compiler generates a Python text.
- **Section "playerj":** This cross compiler generates a Java text.

Section "util"

There are ready-made scripts to cross compile the Dogelog player into module form and into canned form. The cross compiler back-ends share the same compiler front-end, realizing the same compiler optimizations for all the programming language targets. The job of the various back-ends is to support the target specific generation of the Albufeira AST.

The following util commands are provided:

run:

The predicate succeeds in writing the module form and the canned form of the Dogelog player for all the programming language targets.

Section "player"

This cross compiler generates a JavaScript text in Unicode. Prolog strings are enclosed by the JavaScript string literal double quote (“). The code points \n, \r, \v, \t, \b, \f, \', \", \\ and / are escaped by a backslash. Control and invalid code points are escaped by the JavaScript notation \xXX and \uXXXX. Otherwise, the Prolog string is written verbatim.

The following transpile commands are provided:

transpile_begin(B):
transpile_begin(B, O):
The predicate succeeds in writing an empty JavaScript file B. The following transpile options O are recognized:

main_entry(E): E indicates whether function main() should be generated.
doge(A): A is the location of Dogelog player.

transpile_add(A, B):
transpile_add(A, B, O):
The predicate succeeds in cross compiling the Prolog text file A into JavaScript file B.
transpile_end(B):
transpile_end(B, O):
The predicate succeeds adding an epilogue to the JavaScript file B.
bundle_add(A, B):
bundle_add(A, B, O):
The predicate succeeds in appending the JavaScript files A to the JavaScript file B, converting JavaScript imports. The bundler silently overwrites an already existing JavaScript file B. The ternary predicate allows specifying bundle options.

keep_export(E): E indicates whether exports should be kept.
keep_import(I): I indicates whether imports should be kept.

Section "playerpy"

This cross compiler generates a Python text in Unicode. Prolog strings are enclosed by the Python string literal double quote (“). The code points \a, \b, \f, \n, \r, \t, \v \', \" and \\ are escaped by a backslash. Control and invalid code points are escaped by the Python notation \xXX, \uXXXX and \UXXXXXXXX. Otherwise, the Prolog string is written verbatim.

The following transpile commands are provided:

transpilepy_begin(B):
transpilepy_begin(B, O):
The predicate succeeds in writing an empty Python file B. The following transpile options O are recognized:

main_entry(E): E indicates whether function main() should be generated.
doge(A): A is the location of Dogelog player.

transpilepy_add(A, B):
transpilepy_add(A, B, O):
The predicate succeeds in cross compiling the Prolog text file A into Python file B.
transpilepy_end(B):
transpilepy_end(B, O):
The predicate succeeds adding an epilogue to the Python file B.
bundlepy_add(A, B):
bundlepy_add(A, B, O):
The predicate succeeds in appending the Python files A to the Python file B, converting Python imports. The bundler silently extends an already existing Python file B. The ternary predicate allows specifying bundle options.

keep_import(I): I indicates whether imports should be kept.

Section "playerj"

This cross compiler generates a Java text in Unicode. Prolog strings are enclosed by the Java string literal double quote (“). The code points \t, \b, \n, \r, \f, \', \" and \\ are escaped by a backslash. Control and invalid code points are escaped by the Java notation \uXXXX. Otherwise, the Prolog string is written verbatim.

The following transpile commands are provided:

transpilej_begin(B):
transpilej_begin(B, O):
The predicate succeeds in writing an empty Java file B. The following transpile options O are recognized:

main_entry(E): E indicates whether function main() should be generated.
doge(A): A is the location of Dogelog player.

transpilej_add(A, B):
transpilej_add(A, B, O):
The predicate succeeds in cross compiling the Prolog text file A into the Java file B.
transpilej_end(B):
transpilej_end(B, O):
The predicate succeeds adding an epilogue to the Java file B.
bundlej_add(A, B):
bundlej_add(A, B, O):
The predicate succeeds in appending the Java files A to the Java file B, converting Java imports. The bundler silently overwrites an already existing Java file B. The ternary predicate allows specifying bundle options.

keep_import(I): I indicates whether imports should be kept.

Acknowledgements

We are thankful to a discussant on SWI-Prolog discourse for challenging us over Prolog deterministic code execution versus imperative code execution.

Indexes

Pictures

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

Tables

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

Acronyms

ISO [1]

References

[1] ISO (1995): Prolog, Part 1: General Core, International Standard ISO/IEC 13211-1, First Edition, 1995-06-01
http://www.iso.org/standard/21413.html

[2] Clocksin, W. (1983): A portable Prolog compiler, Logic Programming Workshop, Albufeira Portugal, January 1983
http://www.softwarepreservation.org/projects/prolog/lisbon/lpw83/p74-Bowen.pdf

[3] Carlson, M. et al. (1988): Garbage collection for Prolog based on WAM. Communications of the ACM 31, 6, 719–740, June 1988
http://dl.acm.org/doi/10.1145/62959.62968

[4] Wirfs-Brock, A. (2020): JavaScript: the first 20 years, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 77. Publication date: June 2020.
http://dl.acm.org/doi/10.1145/3386327

[5] JavaScript (2020): ECMAScript® 2020 Language Specification, 11th-Edition, Ecma International, June 2020
http://262.ecma-international.org/11.0/