Dogelog Player Host

Version 1.1.5, December 17, 2023

XLOG Technologies AG

Dogelog Prolog

Dogelog Player 1.1.5

Host Interface

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

Date: December 17, 2023
Version: 0.1

Warranty & Liability

To the extent permitted by applicable law and unless explicitly otherwise agreed upon, XLOG Technologies AG makes no warranties regarding the provided information. XLOG Technologies AG assumes no liability that any problems might be solved with the information provided by XLOG Technologies AG.

Rights & License

Trademarks

Table of Contents

Change history

Jan Burse, December 17, 2023, 0.1:

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

1 Introduction

t.b.d.

- **Dogelog Deployment:** t.b.d.
- **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 Deployment

t.b.d.:

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

2.1 t.b.d.

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

t.b.d.

3 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.

- **Native API:** We provide a native API that exposes core routines. The native API is available for the JavaScript and the Python platform.

- **Host Specifics:** This section is concerned with those API calls that do not agree on availability on all target platforms.

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

3.1 Native 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. The native API is available for the JavaScript and the Python platform. This section is concerned with those API calls that agree on availability on all target platforms.

- **Registry API:** Dogelog Player allows registering special foreign functions. They decide on their own whether they are deterministic or non-deterministic.

- **Term API:** The term API provides the creation and basic operations of Prolog terms. Only Prolog variables and Prolog compounds have their own classes.

- **Stream API:** The stream API provides access to objects representing input/output streams for custom implementations beyond the standard.

- **Stage API:** 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.

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

Registry API

The registry API overlaps with the embedding API documented in the language reference of the Dogelog Player, in that the same add() host language call is used to either add Prolog clauses to the Prolog database or register foreign functions. What the native API then exposes is mainly registering foreign functions:

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

The following registry API calls are provided:

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

get_cont(): (host language)
: The function returns the current continuation.

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

cont(C): (host language)
: The function sets the current continuation to C.

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.

Term API

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.

new Variable(): (host language)
: Create a Prolog variable.

new Compound(F, A): (host language)
: Create a Prolog compound with functor F and arguments A.

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.

Stream API

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.

Stage API

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.

Neck API

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.

3.2 Host Specifics

This section is concerned with those API calls that do not agree on availability on all target platforms. Differences between platforms are for example due to the modelling of Prolog terms where our targets JavaScript and Python differ. Further differences might be found related to the runtime environment, for example the browser has a live DOM.

JavaScript and Python can freely invoke function pointers and there is no type system that checks compatibility at compile time. However, Java requires object type declarations. Therefore, whenever functions were dynamically invoked, we had to solve this for Java via objects. We list the additionally defined foreign interfaces.

- **JavaScript Specifics:** The specifics mainly deal with the number blending that is applied in the Dogelog Player for the JavaScript platform.

- **Python Specifics:** The specifics mainly deal with a number blending problem on the Python platform, that the Dogelog Player needs to master.

- **Browser Specifics:** The Dogelog player version that targets JavaScript can be run from within nodeJS or inside a browser, the later has additional calls.

- **Java Specifics:** The specifics mainly deal with the number blending that is applied in the Dogelog Player for the Java platform.

JavaScript Specifics

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. The mapping is documented in the language reference of the Dogelog Player.

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.

Python Specifics

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 Python specific calls are provided:

atomic_equal(S, T): (host language)
Determine whether the two atomics S and T are equal.

Browser Specifics

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 current use in the markup library but can also be directly accessed through the native API.

The following browser specific calls are provided:

set_cursor(C): (host language)
Sets the output and error cursor to the DOM element C.

Java Specifics

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. The mapping is documented in the language reference of the Dogelog Player.

The following Java specific calls are provided:

is_bigint(T): (host language)
Check whether the object T is a Java bigint.
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.

3.3 Native Loader

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.

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

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

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

JavaScript Module

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 "dogelog.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.

Python Source

Python source files are detected by the extension “.py”. They are loaded synchronously and dynamically by using the Python library importlib. 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 dogelog import (add, make_special,
    ...)
...
def main():
   add("<name>", <arity>, make_special(<func>))
   ...
```

In the above the native API is made visible through some from. 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.

Java Class

Java Class are first detected as source by the extension “.java” and then as bytecode by the extension “.class”.

t.b.d.

4 Dogelog Transpiler

Concerning the files "compile" and "loader", these files need to be cross-compiled into the Albufeira embedding to build the Dogelog player. On the other hand, the files "index", "special" and "machine" are already genuinely host language and need not be touched. The main entry point for the cross compiler is the file "main".

- **Folder "util":** There are ready made scripts to cross compile the Dogelog player.
- **Folder "albufeira":** The instruction set of the Dogelog player is rather minimal. The same instruction set is used in cross compilation and at runtime.
- **Folder "player":** Typically, a list of Prolog terms, representing a clause, is made printable to the JavaScript target.
- **Folder "playerpy":** Typically, a list of Prolog terms, representing a clause, is made printable to the Python target.
- **Folder "playerj":** Typically, a list of Prolog terms, representing a clause, is made printable to the Java target.

4.1 Folder "util"

There are ready-made scripts to cross compile the Dogelog player into module form and into canned form. More details are explained here:

- **Section "main":** There are ready made scripts to cross compile the Dogelog Player into module form and into canned form.

- **Section "meta":** During cross compilation the Prolog texts receives the same style check as when they would be consulted by the Dogelog player..

- **Section "Unicode":** The Dogelog player accepts Prolog texts in Unicode code points based on Unicode general categories and numeric values.

- **Section "helper":** The Dogelog player transpiler makes here and then usage of more general file operations, such as copying a file.

Section "main"

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 front-end compile.p. We use simple Prolog write statements to emit the text of the target host language. Depending on the target, strings of the host language will use their specific escape mechanism.

- **player/cross/main.p:** The Prolog text for the scripts that target JavaScript.
- **playerpy/cross/main.p:** The Prolog text for the scripts that targets Python.

Either the Dogelog runtime or the Dogelog player itself can currently perform the compilation. The ready-made scripts do also generate canned versions of the Dogelog player. In a canned version all the imported files are stripped from their exports and bundled together with the main file to form a new single file.

The scripts provide the following commands:

run:
: The predicate succeeds in writing the module form and the canned form of the Dogelog player for all the host languages.

cross_main:
cross_mainpy:
: The predicate succeeds in writing the module form and the canned form of the Dogelog player of the corresponding host language.

Section "meta"

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
```

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.

Section "helper"

The Dogelog player transpiler makes here and then usage of more general file operations, such as copying a file. The Prolog predicate copy_file/2 copies an arbitrary text file.

The following helper predicates are provided:

copy_file(A, B):
copy_file(A, B, O):

The predicate succeeds. As side effect it copies the file A into the file B. An already existing file B is silently overwritten. The ternary predicate allows specifying copy options.

append(B): B is the append file flag.

canonical_add(A, B):

The predicate succeeds. As side effect it canonifies the Prolog text A into the file B. An already existing file B is silently extended.

4.2 Folder "albufeira"

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 "trimming":** The Prolog interpreter regularly performs environment trimming to help the host language garbage collector.

- **Section "stages":** The Dogelog player handling of predicates and clauses has evolved to the ability to rollback of non-monotonic changes.

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 generates the following 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.

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

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 generates the following objects:

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

Place(V), V < -1:
: 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.

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

Skeleton(F, L):
: 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.

otherwise C:
: In building mode gives the embedded constant C. 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 following built-ins belong to the Albufeira instructions:

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

The values of GC_MAX_INFERS is currently:

```
GC_MAX_INFERS = 92000 /* JavaScript */
GC_MAX_INFERS = 55000 /* Python */
GC_MAX_INFERS = 200000 /* Java */
```

Section "stages"

From its inception the Dogelog player had a concept of a stage giving the system the stage number -1 and the user the stage number 0. 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 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 roll-back 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:

```
sys_op/4
sys_source/1
sys_predprop/3
sys_lastpred/2
sys_loading/1
sys_including/2
```

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

4.3 Folder "player"

The Dogelog player is available for the JavaScript platform. Both the common JavaScripot and the module JavaScript dialect variants have been tested. Also, both the browser and NodeJS runtime variants have been tested. The following incarnations are pre bundled within the distribution archive and their support is shown:

- **player/drawer/index.mjs:** The entry point for the un-canned Dogelog player. Mainly suited for module JavaScript and for both the browser and NodeJS.

- **player/canned/dogelog.mjs:** The entry point for the canned async Dogelog player. Mainly suited for module JavaScript and for both the browser and NodeJS.

Typically, a list of Prolog terms, representing clauses, is made either printable:

- **Section "transpile":** The transpiler generates the corresponding Albufeira instruction streams for the JavaScript target.

- **Section "bundle":** The bundler creates a canned form for the JavaScript target.

- **Section "platform":** A transpiler needs to generate printable Prolog literals. We describe the JavaScript representation of Prolog atomics in more detail.

Section "transpile"

The Prolog predicates transpile_begin/[2,3], transpile_add/[2,3] and transpile_end/[1,2] cross compiles a Prolog text to a host language. It does so in that it generates the corresponding Albufeira instruction streams and in that it places the corresponding host language calls.

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.

Section "bundle"

The Prolog predicates bundle_add/[2,3] creates a canned form for the JavaScript target. It does so by mainly removing JavaScript comments. It can be also used to modify JavaScript exports and imports. More details are explained below.

The following bundle commands are provided:

bundle_add(A, B):
bundle_add(A, B, O):

The predicate succeeds in appending the host language files A to the host language file B, converting host language modules to host language single file. The bundler silently extends an already existing host language file B. The ternary predicate allows specifying bundle options. The following bundle options are supported:

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

Section "platform"

A transpiler needs to generate host language literals to create the Prolog atomics demanded by the corresponding Albufeira instructions. We describe the host language literals of Prolog atomics for JavaScript in more detail. The basic data types are atom, number and reference. The number data type is further divided into integer and float.

The 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 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
```

4.4 Folder "playerpy"

The Dogelog player is available for the Python platform. Both CPython and PyPy have been tested. PyPy can run the Dogelog player much faster than CPython, and the Dogelog player garbage collection parameters are set for PyPy. The following incarnations are pre bundled within the distribution archive and their support is shown:

- **playerpy/drawer/index.py:** The entry point for the un-canned Dogelog player. Suited for both CPython and PyPy.

- **playerpy/canned/dogelog.py:** The entry point for the canned Dogelog player. Suited for both CPython and PyPy.

Typically, a list of Prolog terms, representing clauses, is made either printable:

- **Section "transpile":** The transpiler generates the corresponding Albufeira instruction streams for the Python target.

- **Section "bundle":** The bundler creates a canned form for the JavaScript target.

- **Section "platform":** A transpiler needs to generate printable Prolog literals. We describe the Python representation of Prolog atomics in more detail.

Section "transpile"

The Prolog predicates transpilepy_begin/[2,3], transpilepy_add/[2,3] and transpilepy_end/[1,2] cross compiles a Prolog text to a host language. It does so in that it generates the corresponding Albufeira instruction streams and in that it places the corresponding host language calls.

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.

Section "bundle"

The Prolog predicates bundlepy_add/[2,3] creates a canned form for the Python target. It does so by mainly removing JavaScript comments. It can be also used to modify Python imports. More details are explained below.

The following bundle commands are provided:

bundlepy_add(A, B):
bundlepy_add(A, B, O):

The predicate succeeds in appending the host language files A to the host language file B, converting host language modules to host language single file. The bundler silently extends an already existing host language file B. The ternary predicate allows specifying bundle options. The following bundle options are supported:

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

Section "platform"

A transpiler needs to generate host language literals to create the Prolog atomics demanded by the corresponding Albufeira instructions. We describe the host language literals of Prolog atomics for Python in more detail. The basic data types are atom, number and reference. The number data type is further divided into integer and float.

The 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 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
```

4.5 Folder "playerj"

The Dogelog player is available for the Java platform. Both JDK 8 and JDK 21 have been tested. JDK 8 can run the Dogelog player a little faster than JDK 21, and the Dogelog player garbage collection parameters are set for JDK 8. The following incarnations are pre bundled within the distribution archive and their support is shown:

- **playerj/drawer/index.jar:** The entry point for the un-canned Dogelog player. Suited for both JDK 8 and JDK 21.

- **playerj/canned/dogelog.jar:** The entry point for the canned Dogelog player. Suited for both JDK 8 and JDK 21.

Typically, a list of Prolog terms, representing clauses, is made either printable:

- **Section "transpile":** The transpiler generates the corresponding Albufeira instruction streams for the Java target.

- **Section "platform":** A transpiler needs to generate printable Prolog literals. We describe the Python representation of Prolog atomics in more detail.

- **Section "handlers":** Unlike JavaScript and Python, Java requires object type declarations. We list the additionally defined interfaces.

Section "transpile"

The Prolog predicates transpilej_begin/[2,3], transpilej_add/[2,3] and transpilej_end/[1,2] cross compiles a Prolog text to a host language. It does so in that it generates the corresponding Albufeira instruction streams and in that it places the corresponding host language calls.

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.

Section "platform"

A transpiler needs to generate host language literals to create the Prolog atomics demanded by the corresponding Albufeira instructions. We describe the host language literals of Prolog atomics for Java in more detail. The basic data types are atom, number and reference. The number data type is further divided into integer and float.

The 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 mapping from Prolog to Python 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
```

Section “handlers”

Unlike JavaScript and Python, Java requires object type declarations. We list the additionally defined interfaces. What came handy are the new JDK 8 functional interfaces and method references. Among the interfaces one finds definitions for the different types of foreign functions and a definition for the callback stored in a choice point.

The following Java specific classes are provided:

```
interface Builtin { Object run(Object[] args); }
interface Check { boolean run(Object[] args); }
interface Funktion { Number eval(Object[] obj); }
interface Callback { Object run(Object rope, int at, Choice choice); }
interface Sender {  void perform(Sink sink) throws IOException; }
interface Receiver { void perform(Source source) throws IOException; }
interface Releaser { void perform(Object obj) throws IOException; }
```

5 Appendix Deployment Listings

t.b.d.

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

5.1 t.b.d.

t.b.d.

- **tictac.p:** The Tic-Tac-Toe game search.

Prolog t.b.d.

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

Tables

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/