HS_SCSCP.historic.lhs

Jost Berthold

7. Januar 2009

_SCSCP.historic.lhs

compiled code

```
{-# OPTIONS_GHC -cpp #-}
{-# LANGUAGE Arrows, NoMonomorphismRestriction #-} -- said to be useful for HXT
module HS_SCSCP where
```

Jost Berthold, University of St.Andrews (jb@cs.st-andrews.ac.uk)

This document describes the first (top-down) implementation ideas for SCSCP in Haskell (client only). SCSCP [**?**] is a protocol for interaction between computer-algebra systems, following the OpenMath standard and developed in the SCIEnce EU project. Basis of our design is the version 1.2 of the protocol specification.

For any terminology, pls. refer to the aforementioned specification when in doubt.

Implementing SCSCP (client) in Haskell

Features summary

The goal of the implementation is to send computation requests from a Haskell program to a CA system, and to receive the answers for further processing.

Furthermore, the module provides features for starting and stopping its respective communication partners, namely SCSCP servers linked to CA systems.

Limitations

- We want to implement an interface for the SCSCP usage via sockets, thus not using a web service.
- Most likely, not all possible functionality of a client will be implemented. For the implemented features, see the subsequent description.
- Where possible, potentially large computation data should not be accessed inside the calling Haskell program. Haskell serves as a coordination layer. Thus, SCSCP messages will only be decomposed to a degree which allows to coordinate the respective computation, and included computation data otherwise passed on as opaque objects.
- we do not implement the binary OpenMath format in this version, but restrict ourselves to the XML variant.

Some imports (technical...):

compiled code

```
-- XML stuff:
import Text.XML.HXT.Arrow


import System.IO
import System.IO.Unsafe
import Control.Monad
import Control.Concurrent

#ifdef PARALLEL_HASKELL
import Eden(Trans)
import Edi
-- import Control.Parallel.Strategies -- no, exported by Edi module
#endif
```

SCSCP message types

These messages are described by the SCSCP spec., and mapped to a Haskell type (containing useful other types for encoding).

compiled code

```
data SCSCPMsg =
               -- to CA System
                PCall { pcName :: CAName
                      , pcData :: [OMObj]
                      , pcOpts :: ProcOptions }
               | PInterrupt -- empty content (!! can cause race condition !!)
               -- from CA System
               -- procedure complete: called Proc.Result here
               | PResult { prResult :: OMObj
                         , prCallID :: CallID
                         , prTime   :: CATime
                         , prMem    :: CAMem }
               | PTerminated { ptCallID :: CallID
                             , ptReturned :: Either SCSCPMsg CAError
                             , ptTime :: CATime
                             , ptMem  :: CAMem }
                deriving (Read,Show)

-- aliases
type CAName = Either String CAStandardName
type CallID = String
type CATime = Int -- ?too coarse?
type CAMem  = Int
type CARef  = String

-- encoded procedure options
data ProcOptions = PCOpts { pcCallID :: CallID
                          , pcResult :: PResultOption
                          , pcMaxTime :: Maybe CATime
                          , pcMinMem  :: Maybe CAMem
                          , pcMaxMem  :: Maybe CAMem
                          , pcDebug :: Bool }
                deriving (Read,Show)

defaultProcOptions = PCOpts undefined Result Nothing Nothing Nothing False

data PResultOption = Result | ResultRef | NoResult
               deriving (Read,Show)

-- encoded errors inside PTerminated
data CAError = CACannotCompute
```

```
              | CAInvalidRef CARef
              | CAInterrupted
              | CANoSuchProc CAName
              | CAMemExhausted
              | CATimeExhausted
                deriving (Read,Show)

-- data objects, opaque if at all possible
data OMObj = OM String -- embedding ascii XML
               deriving (Read,Show)

noObject = OM "no object!"
```

SCSCP assumes initial exchange of technical information messages (see below), after which a sequence of dialogs between the client and SCSCP server is performed, where the client sends a sequence of `PCall` messages, and the server responds each of them, in their original order, with a corresponding `PResult` or `PTerminated` message. The specification describes `CallID` as a convenience and debug feature only.

Clients can also send an `PInterrupt` message, containing no data. Its semantics on receiver (server) side is to stop the *current* procedure immediately, further ones might be in the message queue of the server (and are not affected), there is no way of addressing one of the several `PCalls` using `PInterrupt`, and the server immediately reacts.

0.1 Assumed standard procedures

SCSCP assumes a set of standard procedures (e.g. to request supported operations from a server), which we encode as follows and may be used as `pcName` in a `PCall`.

compiled code

```
data CAStandardName = GetAllowedHeads -- returns "symbol_sets"
                    | GetSignature    -- returns "signature"
                                      -- (OMSymbol,minarg,maxarg,(list of) symbol_sets )
                                      --      JB: pTerminated if not supported? Guess so.
                    | GetTransientCD  -- returns server-specific Content Dictionary
                                      --      (should prefix SCSCP_transient_)
                    | GetServiceDescr -- returns "service_description",
                                      -- containing 3 Strings:
                                      --      CA system name, version, and descriptive text
                    | StoreObj           -- computes an object, stores it and returns CARef
                    | RetrieveObj CARef  -- takes ref, returns the OMObj
                    | UnbindObj CARef    -- deletes a CARef'ed object from the server
                deriving (Read,Show)
```

The XML format specification

Part 4 of the spec. contains an informal specification of the messages, which we reproduce here for documentation.

- Procedure Call (`PCall`)

```
<OMOBJ>
 <OMATTR>
 <!-- beginning of attribution pairs -->
```

```
 <OMATP>
  <OMS cd="scscp1" name="call_ID" />
  <OMSTR>call_identifier</OMSTR>
  <OMS cd="scscp1" name="option_runtime" />
  <OMI>runtime_limit_in_milliseconds</OMI>
  <OMS cd="scscp1" name="option_min_memory" />
  <OMI>minimal_memory_required_in_bytes</OMI>
  <OMS cd="scscp1" name="option_max_memory" />
  <OMI>memory_limit_in_bytes</OMI>
  <OMS cd="scscp1" name="option_debuglevel" />
  <OMI>debuglevel_value</OMI>
  <OMS cd="scscp1" name="option_return_object" />
  <!-- another possibility is "option_return_cookie" -->
  <OMSTR></OMSTR>
 </OMATP>
 <!-- Attribution pairs finished, now the procedure call -->
 <OMA>
  <OMS cd="scscp1" name="procedure_call" />
  <OMA>
   <!-- "SCSCP_transient_" is an obligatory prefix
        in the name of a transient CD -->
   <OMS cd="SCSCP_transient_identifier"
       name="NameOfTheProcedureRegisteredAsWebService" />
   <!-- Argument 1 -->
   <!-- ... -->
   <!-- Argument M -->
  </OMA>
 </OMA>
</OMATTR>
</OMOBJ>
```

♠**JB:** So why are these call attributes not XML attributes? Must be because OpenMath had to be left unmodified. IMHO, an extension by a new tag would have been the way to go... including all PC attributes as real XML attributes, and containing the argument list only♠

- Interrupt Signal (`PInterrupt`): no content at all.
- Procedure Completed (`PResult`)

```
<OMOBJ>
 <OMATTR>
  <!-- Attribution pairs, dependently on the debugging level
       may include procedure name, OM object for the original
       procedure call, etc. -->
  <OMATP>
   <OMS cd="scscp1" name="call_ID" />
   <OMSTR>call_identifier</OMSTR>
   <OMS cd="scscp1" name="info_runtime" />
   <OMI>runtime_in_milliseconds</OMI>
   <OMS cd="scscp1" name="info_memory" />
   <OMI>used_memory_in_bytes</OMI>
  </OMATP>
  <!-- Attribution pairs finished, now the result -->
  <OMA>
   <OMS cd="scscp1" name="procedure_completed" />
   <!-- The result itself, may be OM symbol for cookie -->
   <!-- OM_object_corresponding_to_the_result -->
  </OMA>
 </OMATTR>
</OMOBJ>
```

And for referenced data (stored in CA), the following (called a cookie):

```
<OMOBJ>
 <OMATTR>
  <OMATP>
   <OMS cd="scscp1" name="call_ID" />
   <OMSTR>call_identifier</OMSTR>
  </OMATP>
  <OMA>
   <OMS cd="scscp1" name="procedure_completed"/>
   <OMR xref="CAS_variable_identifier" />
  </OMA>
 </OMATTR>
</OMOBJ>
```

- Procedure Terminated (`PTerminated`)

```
<OMOBJ>
 <OMATTR>
  <!-- beginning of attribution pairs -->
  <OMATP>
   <OMS cd="scscp1" name="call_ID" />
   <OMSTR>call_identifier</OMSTR>
   <OMS cd="scscp1" name="info_runtime" />
   <OMI>runtime_in_milliseconds</OMI>
   <OMS cd="scscp1" name="info_memory" />
   <OMI>used_memory_in_bytes</OMI>
  </OMATP>
  <!-- end of attribution pairs -->
  <!-- now the application part of the OM object -->
  <OMA>
   <OMS cd="scscp1" name="procedure_terminated" />
   <OME>
    <OMS cd="scscp1" name="name_of_standard_error"/>
    <!-- Error description depends on error type -->
    <OMSTR>Error_message</OMSTR>
   </OME>
  </OMA>
 </OMATTR>
</OMOBJ>
```

Message exchange between client and server. A ref. implementation(?)

This is mostly done using XML processing instructions. The initialisation is a sequence of messages as follows, where things like attribute order and format is strictly fixed.

Examples:

```
Server -> Client:
<?scscp service_name="MuPADserver" service_version="1.1"
service_id="host:26133" scscp_versions="1.0 3.4.1 1.2special" ?>

Client -> Server:
<?scscp version="1.0" ?>

Server -> Client:
<?scscp version="1.0" ?>
```

```
Server -> Client:
<?scscp service_name="MuPADserver" service_version="1.1"
```

```
service_id="host:26133" scscp_versions="1.0 3.4.1 1.2special" ?>

Client -> Server:
<?scscp version="2.0" ?>

Server -> Client:
<?scscp quit reason="not supported version 2.0" ?>

OR JUST: <?scscp quit ?>
```

The actual data messages are enclosed in processing instruction blocks:

```
<?scscp start ?>
... message (OpenMath object), formats see above...
<?scscp end ?>
```

The exception is the interrupt signal, which is just SIGUSR2 to the server.

Messages can be canceled using `<?scscp cancel ?>` to close the block. The server should not process the message.

Sessions are terminated using `<?scscp quit ?>`, optionally giving a reason as above.