Abstract

Contents

1 domain Syntax

⟨*domain SYNTAX Syntax*⟩≡

```
)abbrev domain SYNTAX Syntax
++ Author: Gabriel Dos Reis
++ Date Created: November 10, 2007
++ Date Last Updated: December 05, 2007
++ Description:  This domain provides a simple, general, and arguably
++ complete representation of Spad programs as objects of a term algebra
++ built from ground terms of type boolean, integers, foats, symbols,
++ and strings.  This domain differs from InputForm in that it represents
++ any entity from a Spad program, not just expressions.
++ Related Constructors: Boolean, Integer, Float, symbol, String, SExpression.
++ See Also: SExpression, SetCategory
++ The equality supported by this domain is structural.
++ Fixme: Provide direct support for boolean values, arbritrary
++       precision float point values.
Syntax(): Public == Private where
  Public ==> Join(UnionType, SetCategory) with
    convert: % -> SExpression
      ++ convert(s) returns the s-expression representation of a syntax.

    convert: SExpression -> %
      ++ convert(s) converts an s-expression to syntax.  Note, when `s'
      ++ is not an atom, it is expected that it designates a proper list,
      ++ e.g. a sequence of cons cell ending with nil.

    coerce: Integer -> %
      ++ coerce(i) injects the integer value `i' into the Syntax domain

    coerce: % -> Integer
      ++ coerce(s) extracts and integer value from the syntax `s'
    autoCoerce: % -> Integer
      ++ autoCoerce(s) forcibly extracts an integer value from
      ++ the syntax `s'; no check performed.  To be called only
      ++ at the discretion of the compiler.

    coerce: DoubleFloat -> %
      ++ coerce(f) injects the float value `f' into the Syntax domain

    coerce: % -> DoubleFloat
      ++ coerce(s) extracts a float value from the syntax `s'.
    autoCoerce: % -> DoubleFloat
      ++ autoCoerce(s) forcibly extracts a float value from the syntax `s';
      ++ no check performed.  To be called only at the discretion of
      ++ the compiler
```

```
coerce: Symbol -> %
  ++ coerce(s) injects the symbol `s' into the Syntax domain.

coerce: % -> Symbol
  ++ coerce(s) extracts a symbol from the syntax `s'.
autoCoerce: % -> Symbol
  ++ autoCoerce(s) forcibly extracts a symbo from the Syntax
  ++ domain `s'; no check performed.  To be called only at
  ++ at the discretion of the compiler.

coerce: String -> %
  ++ coerce(s) injects the string value `s' into the syntax domain

coerce: % -> String
  ++ coerce(s) extracts a string value from the syntax `s'.
autoCoerce: % -> String
  ++ autoCoerce(s) forcibly extracts a string value from
  ++ the syntax `s'; no check performed.  To be called only at
  ++ the discretion of the compiler.

buildSyntax: (Symbol, List %) -> %
  ++ buildSyntax(op, [a1, ..., an]) builds a syntax object for op(a1,...,an).

buildSyntax: (%, List %) -> %
  ++ buildSyntax(op, [a1, ..., an]) builds a syntax object for op(a1,...,an).

nil?: % -> Boolean
  ++ nil?(s) is true when `s' is a syntax for the constant nil.

getOperator: % -> Union(Integer, DoubleFloat, Symbol, String, %)
  ++ getOperator(x) returns the operator, or tag, of the syntax `x'.
  ++ The return value is itself a syntax if `x' really is an
  ++ application of a function symbol as opposed to being an
  ++ atomic ground term.


getOperands: % -> List %
  ++ getOperands(x) returns the list of operands to the operator in `x'.

compound?: % -> Boolean
  ++ compound? x is true when not an atomic syntax.

_case: (%, [|Integer|]) -> Boolean
  ++ x case Integer is true is x really is an Integer
```

```
    _case: (%, [|DoubleFloat|]) -> Boolean
      ++ x case DoubleFloat is true is x really is a DoubleFloat

    _case: (%, [|Symbol|]) -> Boolean
      ++ x case Symbol is true is x really is a Symbol

    _case: (%, [|String|]) -> Boolean
      ++ x case String is true is x really is a String

  Private ==> SExpression add
    rep(x: %): SExpression ==
      x pretend SExpression

    per(x: SExpression): % ==
      x pretend %

    x = y ==
      EQUAL(x,y)$Lisp @ Boolean

    s case Integer ==
      integer? rep s

    s case DoubleFloat ==
      float? rep s

    s case String ==
      string? rep s

    s case Symbol ==
      symbol? rep s

    convert(x: %): SExpression ==
      rep x

    convert(x: SExpression): % ==
      per x

    coerce(i: Integer): % ==
      i pretend %

    autoCoerce(i: %): Integer ==                -- used for hard coercion
      i : Integer

    coerce(i: %): Integer ==
      i case Integer => i
      userError "invalid conversion target type"
```

```
coerce(f: DoubleFloat): % ==
  f pretend %

autoCoerce(f: %): DoubleFloat ==            -- used for hard coercion
  f : DoubleFloat

coerce(f: %): DoubleFloat ==
  f case DoubleFloat => f
  userError "invalid conversion target type"

coerce(s: Symbol): % ==
  s pretend %

autoCoerce(s: %): Symbol ==                 -- used for hard coercion
  s : Symbol

coerce(s: %): Symbol ==
  s case Symbol => s
  userError "invalid conversion target type"

coerce(s: String): % ==
  s pretend %

autoCoerce(s: %): String ==                 -- used for hard coercion
  s : String

coerce(s: %): String ==
  s case String => s
  userError "invalid conversion target type"

buildSyntax(s: Symbol, l: List %): % ==
  -- ??? ideally we should have overloaded operator ‘per’ that convert
  -- from list of syntax to syntax.  But the compiler is at the
  -- moment defective for non-exported overloaded operations.
  -- Furthermore, this direct call to ‘CONS’ is currently necessary
  -- in order to have the Syntax domain compiled as early as possible
  -- in algebra boostrapping process.  It should be removed once
  -- the bootstrap process is improved.
  CONS(s,l)$Lisp @ %

buildSyntax(op: %, l: List %): % ==
  CONS(op,l)$Lisp @ %

nil? x ==
  null? rep x
```

```
getOperator x ==
  atom? rep x => userError "atom as operand to getOperator"
  op := car rep x
  symbol? op => symbol op
  integer? op => integer op
  float? op => float op
  string? op => string op
  convert op

compound? x ==
  pair? rep x

getOperands x ==
  s := rep x
  atom? s => []
  [per t for t in destruct cdr s]
```

2 domain ConstructorCall

⟨*domain CTORCALL ConstructorCall*⟩≡

```
)abbrev domain CTORCALL ConstructorCall
++ Author: Gabriel Dos Reis
++ Date Created: January 19, 2008
++ Date Last Updated: January 19, 2008
++ Description: This domains represents a syntax object that
++ designates a category, domain, or a package.
++ See Also: Syntax, Domain
ConstructorCall(): Public == Private where
  Public ==> CoercibleTo OutputForm with
    constructorName: % -> Symbol
      ++ constructorName c returns the name of the constructor
    arguments: % -> List Syntax
      ++ arguments returns the list of syntax objects for the
      ++ arguments used to invoke the constructor.

  Private ==> add
    rep(x: %): List Syntax ==
      x pretend List(Syntax)

    constructorName x ==
      (first rep x)::Symbol

    arguments x ==
      rest rep x

    coerce x ==
      outputDomainConstructor(x)$Lisp
```

3 domain ElaboratedExpression

⟨*domain ELABEXPR ElaboratedExpression*⟩≡

```
)abbrev domain ELABEXPR ElaboratedExpression
++ Author: Gabriel Dos Reis
++ Date Created: January 19, 2008
++ Date Last Updated: January 20, 2008
++ Description: This domains an expresion as elaborated by the interpreter.
++ See Also:
ElaboratedExpression(): Public == Private where
  Public ==> CoercibleTo OutputForm with
    type: % -> ConstructorCall
      ++ type(e) returns the type of the expression as computed by
      ++ the interpreter.
    constant?: % -> Boolean
      ++ constant?(e) returns true if `e' is a constant.
    getConstant: % -> Union(SExpression,"failed")
      ++ getConstant(e) retrieves the constant value of `e'e.
    variable?: % -> Boolean
      ++ variable?(e) returns true if `e' is a variable.
    getVariable: % -> Union(Symbol,"failed")
      ++ getVariable(e) retrieves the name of the variable `e'.
    callForm?: % -> Boolean
      ++ callForm?(e) is true when `e' is a call expression.
    getOperator: % -> Union(Symbol, "failed")
      ++ getOperator(e) retrieves the operator being invoked in `e',
      ++ when `e' is an expression.
    getOperands: % -> Union(List %, "failed")
      ++ getOperands(e) returns the of operands in `e'e, assuming it
      ++ is a call form.

  Private ==> add
    immediateDataTag := INTERN("--immediateData--")$Lisp

    isAtomic(x: %): Boolean ==
      ATOM(x)$Lisp @ Boolean

    type x ==
      getMode(x)$Lisp @ ConstructorCall

    callForm? x ==
      CONSP(x)$Lisp @ Boolean

    getOperator x ==
      op: SExpression := getUnnameIfCan(x)$Lisp
      null? op => "failed"
```

```
  op pretend Symbol

constant? x ==
  isAtomic x and
    EQ(getUnnameIfCan(x)$Lisp, immediateDataTag)$Lisp : Boolean

getConstant x ==
  constant? x => getValue(x)$Lisp @ SExpression
  "failed"

variable? x ==
  isAtomic x and not constant? x

getVariable x ==
  variable? x => symbol (getUnname(x)$Lisp@SExpression)
  "failed"
```

4 License

⟨*license*⟩≡

```
--Copyright (C) 2007, Gabriel Dos Reis.
--All rights reserved.
--
--Redistribution and use in source and binary forms, with or without
--modification, are permitted provided that the following conditions are
--met:
--
--    - Redistributions of source code must retain the above copyright
--      notice, this list of conditions and the following disclaimer.
--
--    - Redistributions in binary form must reproduce the above copyright
--      notice, this list of conditions and the following disclaimer in
--      the documentation and/or other materials provided with the
--      distribution.
--
--    - Neither the name of The Numerical Algorithms Group Ltd. nor the
--      names of its contributors may be used to endorse or promote products
--      derived from this software without specific prior written permission.
--
--THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
--IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
--TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
--PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
--OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
--EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
--PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
--PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
--LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
--NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
--SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

⟨***⟩≡

⟨*license*⟩
⟨*domain SYNTAX Syntax*⟩
⟨*domain CTORCALL ConstructorCall*⟩
⟨*domain ELABEXPR ElaboratedExpression*⟩