LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ID

(Version 88.0)

Reference Manual

Computation Structures Group Memo 284

March 25, 1988

Rishiyur S. Nikhil

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Id

(Version 88.0)

Reference Manual

Rishiyur S. Nikhil

March 25, 1988

Computation Structures Group
Laboratory for Computer Science
Massachusetts Institute of Technology

545 Technology Square,
Cambridge, MA 02139, USA

Table of Contents at end of document.

1 Introduction and History

Id is a parallel programming language designed by members of the Computation Structures Group of MIT/LCS. It is used for programming dataflow and other parallel machines.

Id is a functional programming language augmented with a parallel data-structuring mechanism called *I-structures*. The purely functional subset of Id is described in Section 2, and non-functional extensions are described in Section 3.

Id traces its roots back to 1978 [1]. Since then, versions of Id have run on simulated dataflow machines, and more recently on a real multi-processor emulation of a dataflow machine. Id/83s [4] was a first cut at a major redesign of the language, based on that experience and on modern ideas in functional languages. This evolved into Id Nouveau (1986) [5], and was revised in 1987 [3]. In 1987 we also developed an abstract operational semantics for Id based on rewrite rules instead of dataflow graphs [2].

Id continues to be a research language. Current investigations include better data structures for parallelism, constructs to express non-deterministic computations, I/O, resource-management *etc.*

This document is not a tutorial on Id.

Attention: Appendix B describes current implementation restrictions, quirks, *etc.* Throughout this document, the sentence "(But see Appendix B.)" draws your attention to some such restriction.

1.1 Incompatible Changes

The following are incompatible changes from the previous version of Id ("Id Nouveau", April 1987 [3]).

- Function definitions (see Section 2.26) are now *always* introduced by the `def` keyword. Previously, function definitions at the top-level had the keyword, whereas function definitions inside blocks did not.
- The terms `array`, `vector`, `matrix`, `k_nD_arrays`, *etc.* are now keywords introducing array comprehensions (see Section 2.28.4).
- We are now serious about type-checking. Unless you explicitly disable it, every program must now pass the type-checker. Some existing programs will now be rejected by the type-checker. Typically, such programs violate the restriction that all components of a list and all components of an array must be homogeneously typed. This restriction was mentioned in the Id Nouveau document, but was not enforced by the compiler.
- For-loop syntax has changed (generalized). Instead of:

  ```
  for j from e1 to e2 by eInc do
  ```

  we now say:

  ```
  for j <- e do
  ```

  where `e` is *any* list expression. The phrase:

  ```
  e1 to e2 by eInc
  ```

  is now a full-fledged expression (*i.e.*, it can be used anywhere), and denotes a list containing the arithmetic series from `e1` through `e2` with `eInc` increment.

2 Functional Id

This section describes the purely functional (and referentially transparent) subset of Id.

2.1 Expressions, Statements and Types

Two major syntactic categories in Id are *expressions* and *statements*.

Every expression denotes a *value*. We use the generic symbols "`e`", "`e1`", *etc.* to designate arbitrary expressions.

Statements appear in the top-level of programs, in blocks, *etc.* Statements are usually type- or identifier-bindings.

Id has a polymorphic type system. Every expression and statement must "type-check", *i.e.,* satisfy certain type-rules; these are explained as each construct is introduced. Types are described by *type-expressions*. We use the generic symbols "`t`", "`t0`", *etc.* to refer to types.

Type-checking in Id is done by *type inference*, *i.e.,* the programmer is not required to declare the types of identifiers or expressions— the type-checker automatically deduces them from the context. For readability and better error-messages, however, there is a facility for declaring types of identifiers (using `typeof` statements, see Section 2.8).

For explaining the type rules in this manual, we use the notation:

```
e::t
```

to assert that the value of expression "`e`" has type "`t`", *i.e.,* lies in the set of values denoted "`t`". This notation *is not part of the language*.

2.2 Programs

A program is a collection of statements, and defines an environment:

```
STATEMENT ;
...
STATEMENT ;
```

(But see Appendix B). The order of the statements is not relevant.

2.3 Grouping

Any expression or type-expression may be enclosed in parentheses. This may be done to override precedence, or merely for visual clarity.

```
( 2 + 3 ) * ( 4 - (f x))

(btree (btree N))
```

(Parentheses are also used for "quoting" binary infix operators; see Section 2.11.)

2.4 Comments

Comments begin with "`%`" and can contain any text up to the end-of-line:

```
% anything goes till the end of the line
```

We recommend the guidelines on page 348 of the Common Lisp manual (Guy L. Steele, Jr., Digital Press, 1984) for commenting code, except that Id has "%" instead of Lisp's ";" as the comment character.

2.5 Identifiers

Identifiers may contain alphabetics, digits, "_", "?" and "`-`" in any order. Examples:

```
x
harry
desmond_2_2
2D_array
nil?
done?
```

A lexical token is an identifier only if it is not a reserved word or a number (see Sections 2.5.1 and 2.14). Upper- and lower- case letters are equivalent.

The identifier "_" consisting of only an underscore is treated specially, and is allowed only in patterns (see Section 2.21).

2.5.1 Reserved Words

The following words are reserved and may not be used as identifiers:

```
abstype      else       or
accumulate   error      rep
and          finally    seq
array        for        then
by           fun        to
call         gets       type
case         if         typeof
def          in         unless
defsubst     matrix     vector
do           next       when
downto       of         while
```

In addition, the following families of words are reserved:

```
k_nD_arrays
k_vectors
k_arrays
k_matrices
nD_array
```

for each $k \geq 1$ and $n \geq 1$.

2.5.2 Standard Identifiers

Standard identifiers are not reserved words (*i.e.*, they can be redefined by the programmer. However, to enhance readability and reusability of code, the programmer is strongly advised *not* to redefine them. See Appendix A for a listing of standard identifiers.

2.6 Types

We use the generic symbols "t", "t1", *etc.* to designate types.

Types are denoted by *type-expressions*, which are either *Type Variables*:

```
*3  *0   *13
```

or N-ary *Constructed Types* ($N \geq 0$):

```
identifier t1 ... tN
```

Pre-defined 0-ary constructed types (also called *Type Constants*):

`C`	(characters)
`N`	(numbers)
`B`	(booleans)
`S`	(character strings)
`SYM`	(symbols)

Pre-defined constructed types:

- **Array Types:**
```
(1D_array t) (vector t) (array t)
(2D_array t) (matrix t)
(3D_array t)
...
```
- **List Types:**
```
(list t)
```

Pre-defined constructed types that also have special syntax:

- **Tuple Types:**
```
t0 , ..., tN
```
- **Function Types:**
```
t0 -> t1
```
  The "->" type operator associates to the right, so that the parentheses can be omitted in the following type-expression:
```
N -> (N -> B)
```

2.6.1 Precedence in Types

In each of the following examples, the parentheses may be dropped:

```
(btree N) -> N
(list N),N
(N -> N),N
```

2.6.2 Polymorphic Types

A type containing a type variable is a *polymorphic* type. *e.g.*, the type of ":", the list constructor, is:

```
*0 -> (list *0) -> (list *0)
```

The type variable stands for "any type", indicating that ":" can construct lists of any type.

However, the type variables in a polymorphic type must be instantiated *uniformly*. These are valid instantiations:

```
N -> (list N) -> (list N)
B -> (list B) -> (list B)
(N->B) -> (list (N->B)) -> (list (N->B))
  ...
```

but this is not a valid instantiation:

```
C -> (list B) -> (list N)
```

Thus, we can have lists of numbers, lists of booleans, *etc.* but a list cannot contain both numbers and booleans (unless packaged into a disjoint union, see Section 2.9).

2.7 Overloading

There are many identifiers that are not polymorphic but overloaded. For example, the equality symbol is available at every non-abstract, non-functional type (see Section 2.12).

For each use of an overloaded identifier, the type-checker will attempt to infer the particular type at which it is used from the surrounding context. If it is unable to do so, an error will be flagged.

For the moment, only built-in symbols (like equality) are overloaded. We expect soon to be able to allow the user to overload identifiers, too.

2.8 Type Declarations

An identifier's type may be declared anywhere in its scope. The statement:

```
typeof x = t
```

asserts that identifier `x` denotes a value of type `t`. Example:

```
typeof map_list =
  (*0 -> *1) -> (list *0) -> (list *1)
```

Since Id's type-checker automatically infers the types of all identifiers, user-specified type declarations are not strictly necessary. However, we strongly recommend their plentiful use because:

- They make programs more readable;
- Error messages from the type-checker will be more localized, and hence more helpful.

Note: a type declaration statement does not *introduce* any new identifiers.

2.9 Algebraic Types

Algebraic types are also called "disjoint union" types.

New algebraic types are declared by the statement:

```
type tx tv1 ... tvN = disj1 | ... | disjM
```

Here, `tx` is the name for the new type. Its N (≥ 0) type parameters are specified by the type-variables `tvJ`. Its M (≥ 1) disjuncts are specified by the `disjJ`s, each of which has the form:

```
tcons t1 ... tL
```

`Tcons` is an identifer and represents a new L-adic (≥ 0) *Constructor*. Each `tJ` is a type-expression constraining the type of the J'th argument of the constructor. Thus,

```
tcons :: t1 -> ... -> tL -> (tx tv1 ... tvN)
```

Examples

Lists of numbers:

```
type nlist = nnil | ncons N nlist
```

The constructors thus defined are:

```
nnil  :: nlist
ncons :: N -> nlist -> nlist
```

Polymorphic lists:

```
type list *0 = nil | (:) *0 (list *0)
```

The constructors thus defined are:

```
nil :: (list *0)
(:) :: *0 -> (list *0) -> (list *0)
```

Polymorphic binary trees:

```
type btree *0 = empty_btree
              | bnode *0 (btree *0) (btree *0)
```

The constructors thus defined are:

```
empty_btree :: (btree *0)
bnode :: *0->(btree *0)->(btree *0)->(btree *0)
```

2.10 Function Applications

Every function has type "`t0 -> t1`" for some argument type "`t0`" and result type "`t1`".

Assuming:

```
ef :: (t0 -> t1)
ex :: t0
```

then the application expression:

```
ef ex :: t1
```

denotes the application of a function (the value of `ef`) to an argument (the value of `ex`).

Application associates to the left. Thus, the following two expressions are equivalent:

```
e1 e2 e3 ... eN
```

```
(((e1 e2) e3) ... eN)
```

2.11 Operators

Some functions are designated by special symbols called *operators*. Unary prefix operator expressions are written:

```
op e
```

Binary infix operator expressions are written:

```
e1 op e2
```

All binary operators can be treated as values by enclosing them in parentheses, *e.g.*,

```
(+) e1 e2
foldr_list (+) 0 list_of_N
```

This is the only special use of parentheses in Id.

The operator "`-`" is used both as a binary infix operator and as a unary prefix operator. "`(-)`" stands for the value of the binary version. To get the effect of the unary version, you can say:

```
((-) 0)
```

(partially applying the binary operator to 0).

2.11.1 Operator Precedence

In decreasing precedence:

operator	*associates*
application	L
`-` (unary)	R
`^`	R
`* /`	L
`+ -`	L
`to by`	-
`:`	R
`++`	R
`!`	L
`== <> < <= > >=`	L
`and`	L
`or`	L
`,` (comma in tuples)	-

2.12 Equality and Inequality

There are two heavily overloaded infix operators: equality `==` and inequality `<>`. They are automatically defined with type:

```
t -> t -> B
```

for all types `t` that do not contain function types and abstract types. Note: they are *not* polymorphic operators.

Inequality is simply the boolean negation of equality.

For all algebraic types `t` (including pre-defined ones like tuples and lists), two objects of type `t` are equal if they have the same structure, *i.e.*,

- They have the same constructor, and
- Their corresponding components (if any) are equal.

Two strings are equal if they are of the same length and their corresponding contents (characters) are equal.

Two objects of type `(nD_array t)` are equal if they have equal index bounds and their corresponding contents are equal.

2.13 Booleans

Booleans are defined as follows:

```
type B = false | true
```

Thus, `false` and `true` are identifiers representing boolean constants, and are also constructors (*i.e.*, they can be used in patterns).

Infix operators:

```
and :: B -> B -> B
or  :: B -> B -> B
```

Both left and right arguments are always evaluated.

Standard identifiers include `not`, the boolean negation function (see Appendix A.1).

2.14 Numbers

All numbers have type "`N`" (for the moment, we do not distinguish integers from floats in the language, though this may change in the future).

Numeric constants are written:

```
255    :: N
0.6667 :: N
1.45   :: N
2.56e4 :: N
3e-3   :: N
```

The radix and exponent are always based on 10. The decimal point must be preceded or followed by at least one digit. The "`e`" must be preceded by a number and followed by a (possibly signed) integer. Example: `2.56e4` denotes 2.56×10^4

Unary arithmetic negation operator:

```
- :: N -> N
```

Infix arithmetic operators:

```
+ :: N -> N -> N
- :: N -> N -> N
* :: N -> N -> N
/ :: N -> N -> N
^ :: N -> N -> N     (exponentiation)
```

Infix comparison operators:

```
<  :: N -> N -> B
<= :: N -> N -> B
>  :: N -> N -> B
>= :: N -> N -> B
```

See Appendix A.2 for standard numeric functions.

2.15 Characters

(Conforms to Common Lisp standard.)

All characters have type "`C`".

Character constants (expressions) are written with a "\", followed by any printable character:

```
\a      :: C
\?      :: C
\\      :: C
...
```

or by a special-character name:

```
\space     :: C
\newline   :: C
\backspace :: C
\tab       :: C
\linefeed  :: C
\page      :: C
\return    :: C
\rubout    :: C
```

Infix comparison operators:

```
<  :: C -> C -> B
<= :: C -> C -> B
>  :: C -> C -> B
>= :: C -> C -> B
```

The ordering is guaranteed only within the following subsets: digit characters, upper-case characters, and lower-case characters.

See Appendix A.3 for standard character functions.

2.16 Symbols

All symbols have type "`SYM`".

A symbol is written as a quoted identifier:

```
'A             :: SYM
'desmond_2_2   :: SYM
'c3po          :: SYM
'7am           :: SYM
```

Unlike Lisp, symbols are are *not* related to program identifiers. Each distinct symbol merely represents a unique global constant. The only primitive operations on symbols are equality and inequality.

2.17 Strings

(Conforms to Common Lisp standard.)

All strings have type "`S`" (but see Appendix B).

Constant strings (expressions) are written between double-quotation marks. "\" is used to quote any character, including "\" itself and """.

```
"Hiya"              :: S
"Say \"What?\""     :: S
```

Upper- and lower-case are distinguished.

Infix comparison operators, with lexicographic ordering:

```
<  :: S -> S -> B
<= :: S -> S -> B
>  :: S -> S -> B
>= :: S -> S -> B
```

Strings are zero-indexed (*i.e.*, the first character is at position 0).

See Appendix A.4 for standard string functions.

2.18 Tuples

An N-tuple has type "`(t1,...,tN)`" where "`tj`" is the type of the j'th component.

Assuming:

```
e1     :: t1
...
eN     :: tN
```

then the tuple expression:

```
e1, ..., eN     :: t1,...,tN
```

(where $N \geq 2$) denotes an n-tuple value.

The comma has lower precedence than any of the other operators. Examples:

```
4+5, true            :: N,B
5, (sqr x, false)    :: N,(N,B)
(5,4),"Hi",(a > b)   :: (N,N),S,B
```

The second expression is a 2-tuple whose second component is itself a 2-tuple. The nesting structure is significant— it is not equivalent to a 3-tuple.

Components of a tuple are accessed *via* pattern-matching (see Section 2.22).

There is no notation for 1-tuples— a 1-tuple of `x` is identified with `x` itself.

2.19 Conditional Expressions

Assuming:

```
e1::B    e2::t    e3::t
```

then the (two-armed) conditional expression is:

```
if e1 then e2 else e3     :: t
```

Precedence of `else`: the parentheses may be omitted in each of these examples:

```
if ... else (x,y)
if ... else (f x y)
if ... else (x and y)
```

2.20 Blocks

Assuming:

```
e::t
```

then the block expression:

```
{  STATEMENT ;
   ...
   STATEMENT
in
   e } : t
```

denotes the value of e evaluated in the environment inside the block.

Each statement must itself be well-typed. Statements usually specify *bindings* associating identifiers to types or values.

Blocks (like *all* Id constructs) follow a static scoping discipline.

The name-environment *inside* a block is the surrounding environment augmented by the names introduced by the statements of the block. A name x may be introduced at most once in a block, and hides any x in the surrounding environment. Names introduced inside a block are invisible outside the block.

Thus, the statements in a block may be recursive and mutually recursive, and the textual order of the statements is not significant.

2.21 Patterns

A *pattern* is either

- an identifier,
- a special constant (number, character, symbol), or
- a term

```
c pat1 ... patN
```

where c is an N-ary constructor name of some algebraic type t, and the patJ's are themselves patterns ($N \geq 0$). In the last case, the pattern is said to be of type t.

All identifiers in a pattern must be unique, except for the "don't-care" identifier "_", which may be repeated.

Special syntax: list patterns can be written:

```
pat1:pat2
```

Special syntax: N-tuple patterns can be written:

```
pat1,...,patN
```

2.22 Pattern-Matching

Matching a pattern to a value can either succeed and produce a set of identifier-value bindings, or it can fail (a runtime error).

A "don't-care" pattern "_" successfully matches any value, and produces no binding.

A pattern-identifier x successfully matches any value, and binds x to that value.

A pattern-constant c successfully matches only the value c, and produces no binding.

A pattern (c pat1 ... patN) successfully matches only a value of the form (c v1 ... vN), and produces the union of all the bindings obtained by matching all the patJ's to their corresponding vJ's.

2.23 Pattern-Binding Statements

The statement:

```
c pat1 ... patN = e
```

($N \geq 0$) introduces, into the current scope, the bindings obtained by matching the pattern on the left-hand side to the value produced by the right-hand side.

2.23.1 Simple Binding Statements

The degenerate case of a pattern-binding is the statement:

```
x = e
```

which introduces x as a name for the value of expression e into the current scope.

2.24 Case-expressions

Assuming:

```
e ::te    e1::t   ...    eN::t
```

and pat1 ... patN are patterns of type te, then the case-expression:

```
{case e of
   pat1 = e1
|  ...
|  patN = eN }    :: t
```

behaves as follows. Let `v` be the value of `e`. All the patterns `pat1 ... patN` are matched to `v`, in no specific order. If `patJ` matches, then the resulting bindings augment the current environment, `eJ` is evaluated in that environment, and its value is returned as the value of the whole expression.

The patterns must be disjoint, *i.e.*, at most one pattern can succesfully match any `e` (this is checked by the compiler). The order of the patterns is therefore not relevant.

The patterns need not be exhaustive— if no pattern matches, it is a runtime error.

The last clause may be preceded by ".." to designate it as a catch-all clause (this is a limited form of ordering):

```
{case e of
    pat1 = e1
    ...
|   patN = eN
|.. patF = eF }  :: t
```

Here, a match of `patF` to `v` (the value of `e`) is attempted only if all other matches fail. Thus, `patF` need not be disjoint from the other patterns.

2.25 Function Abstractions

A function abstraction expression (a form of lambda-expression) is written:

```
{fun pat11 ... pat1N = e1
|    pat21 ... pat2N = e2
     ...
|    patM1 ... patMN = eM
|..  patL1 ... patLN = eL }
```

and is equivalent to:

```
{fun x1 ... xN =
    {case (x1,...,xN) of
        (pat11,...,pat1N) = e1
    |   (pat21,...,pat2N) = e2
        ...
    |   (patM1,...,patMN) = eM
    |.. (patL1,...,patLN) = eL }}
```

and represents an "anonymous" function of arity N (≥ 1) whose formal parameters are the `xJ`s and whose body is the case-expression. As usual, static scoping rules are followed.

The final "catch-all" clause (signalled by "..") is optional.

2.26 Function Definitions

A function definition statement is written:

```
def  f pat11 ... pat1N = e1
 |   f pat21 ... pat2N = e2
     ...
 |   f patM1 ... patMN = eM
 |.. f patL1 ... patLN = eL
```

and is equivalent to the simple binding statement:

```
f = {fun x1 ... xN =
        {case (x1,...,xN) of
            (pat11,...,pat1N) = e1
        |   (pat21,...,pat2N) = e2
        |   ...
        |   (patM1,...,patMN) = eM
        |.. (patL1,...,patLN) = eL }}
```

The final "catch-all" clause (signalled by "..") is optional.

2.27 Lists

The standard list type can be defined as:

```
type list *0 = nil | (:) *0 (list *0)
```

where ":" is the infix "cons" operator.

2.27.1 Binary Infix List Operators

Appending two lists:

```
++ :: (list *0) -> (list *0) -> (list *0)
```

Indexing a list (first element is 0'th):

```
!  :: (list *0) -> N -> *0
```

2.27.2 Arithmetic Series Operators

Assuming:

```
e1::N    e2::N    eInc::N
```

evaluate to integers $v1$, $v2$ and $vInc$, respectively, then the expressions:

```
e1 to e2 by eInc        :: (list N)
e1 downto e2 by eInc    :: (list N)
```

produce lists containing ($v1$, $v1 + vInc$, $v1 + 2vInc$, ..., $v2$), and ($v1$, $v1 - vInc$, $v1 - 2vInc$, ..., $v2$), respectively.

Note: $vInc$ must always be positive.

The short forms:

```
e1 to e2     :: (list N)
e1 downto e2 :: (list N)
```

assume that $vInc$ is $+1$.

Precedence of `to`, `downto` and `by`: the parentheses may be omitted in each of these examples:

```
... to (f x)
... downto (f x)
... to (e1 + e2)
... by (f x)
```

2.27.3 List Comprehensions

A list-comprehension is written:

```
{: e || GEN1 & ... & GENn }
```

($n \geq 1$). Each generator `GEN` is written in one of two ways:

```
pat <- e FILTER1 ... FILTERm
pat  = e FILTER1 ... FILTERm
```

($m \geq 0$). Each `FILTER` is written in one of two ways:

```
when epw
unless epu
```

Generator behavior

In the first form (using `<-`), `e` must be a list of values; `pat` is matched to each element of the list, generating a sequence of environments that bind the pattern variables.

In the second form (using `=`), `pat` is matched to the value of `e`, generating an environment that binds the pattern variables.

The environments are then filtered, *i.e.,* those environments in which an `epw` evaluates false or an `epu` evaluates true are discarded. The filters are tried in sequence from left to right, *i.e.,* if a filter rejects an environment, the subsequent filter expressions are not evaluated for that environment.

Generator sequence behavior

The generators are evaluated from left to right. For each environment Env in the sequence of environments produced before `GENj`,

- `GENj` is evaluated in Env, and produces a set of environments Env_{j1}, Env_{j2}, ...
- Env is replaced in the sequence by the augmented environments $Env + Env_{j1}$, $Env + Env_{j2}$, ...

Thus, the net result of the generator sequence is a sequence of environments containing bindings for the pattern variables of all the generators.

List-comprehension behavior

The expression `e` is evaluated in each environment produced by the generator sequence, and the values are collected into a list (in the same order), which is the result of the whole expression.

Examples

A list of x-y coordinates in the first octant of a 100-square:

```
{: x,y || x <- 0 to 100 & y <- 0 to x }
```

A list of x-y coordinates in a 100-square that are not on the axes or on the diagonals:

```
{: x,y || x <- 0 to 100 when x <> 0
        & y <- 0 to 100 when y <> 0
                        unless x == y }
```

See also Appendix A.5 for standard list functions.

2.28 Arrays

Arrays are collections of uniformly-typed objects, with a constant access-time for each component.

2.28.1 Array Types

An n-dimensional array ($n \geq 1$) whose components have type `t` has type:

```
nD_array t
```

Arrays can contain objects of any type, including other arrays. The following two types are *not* equivalent:

```
2D_array t
```

```
1D_array (1D_array t)
```

(But see Appendix B).

Synonyms for `1D_array`: `vector`, `array`

Synonym for `2D_array`: `matrix`

2.28.2 Array Selection

Assuming:

```
a    :: (nD_array t)
e    :: (N,...,N)
```

then the array-selection expression:

```
a[e]    :: t
```

returns the value of the `j1,...,jn`'th component of the array "`a`", where `j1,...,jn` is the value of "`e`"

Note that the index expression can be *any* expression that returns an n-tuple of integers, *i.e.*, it does not have to be a literal tuple-expression. (but see Appendix B).

2.28.3 Array Index Bounds

For each $n \geq 1$, there is a function that returns the index bounds of n-dimensional arrays:

```
1D_bounds :: (1D_array *0) -> (N,N)
2D_bounds :: (2D_array *0) -> ((N,N),(N,N))
...
```

Synonym for `1D_bounds`: `bounds`.

2.28.4 Array Comprehensions

Array comprehensions are used to create (define) arrays.

For each $k \geq 1$ and $n \geq 1$, assuming a bounds expression (an n-tuple of integer 2-tuples):

```
eBounds :: ((N,N), ... ,(N,N))
```

and a set of index expressions (each an n-tuple of integers):

```
eJ1 :: (N,...,N)
```

and a set of component expressions (each a k-tuple):

```
eJ2 :: (t1,...,tk)
```

then the array comprehension expression:

```
{k_nD_arrays eBounds
| [e11] = e12 || gen & ... & gen
| ...
| [eM1] = eM2 || gen & ... & gen }
```

returns a k-tuple of n-dimensional arrays. Each `gen` is a generator, possibly including filters (exactly as in list-comprehensions).

Array comprehension behavior
`eBounds` is evaluated to produce lower- and upper-bounds for each of n dimensions. k arrays with these dimensions are created. Then, the subsequent clauses are all executed in parallel to fill the arrays. (The top-to-bottom order of the clauses has no significance.)

Clause behavior
See *Generator behavior* and *Generator sequence behavior* in Section 2.27.3 on list comprehensions to see how each generator sequence

```
gen & ... & gen
```

produces a sequence of environments. Now, in each such environment, `eJ1` is evaluated to produce an index into the arrays (an n-tuple). `EJ2` is evaluated to produce a k-tuple specifying the contents of that location in each of the k arrays.

A runtime error occurs if the contents of an array at some index is defined more than once, *i.e.*, if the array comprehension specifies values twice at the same index (j1,...,jN).

If, at some index, the array comprehension specifies no value at all, then that location simply remains undefined (indistinguishable from a non-terminating computation).

The generator sequences "|| gen & ... & gen" are optional. In this case, eJ1 and eJ2 specify the contents of a single location.

The k_nD_arrays keywords have synonyms for some common cases:

	$k = 1$	$k \geq 1$
$n = 1$	array vector	k_arrays k_vectors
$n = 2$	matrix	k_matrices
$n \geq 1$	nD_array	

Examples

The vector sum of two vectors A and B:

```
{array (1,N) | [i] = A[i]+B[i] || i <- 1 to N }
```

An array defined using a "wavefront" recurrence:

```
A = {matrix (1,N),(1,N)
    | [1,1] = 1
    | [i,1] = 1                 || i <- 2 to N
    | [1,j] = 1                 || j <- 2 to N
    | [i,j] = A[i-1,j] +
              A[i-1,j-1] +
              A[i,j-1]          || i <- 2 to N
                                 & j <- 2 to N }
```

An array containing the inverse of a given permutation in array A:

```
{array (1,N) | [A[i]] = i || i <- 1 to N }
```

See also Appendix A.7 for standard array functions.

2.29 Accumulators

Accumulators are an extension of arrays.

```
{k1_n1D_arrays eBounds1
 | [ei] = evs || gen & ... & gen
 | ...
 | [ei] = evs || gen & ... & gen
 k2_n2D_arrays eBounds2
 | [ei] = evs || gen & ... & gen
 | ...
 | [ei] = evs || gen & ... & gen
 .
 .
 .
 kM_nMD_arrays eBoundsM
 | [ei] = evs || gen & ... & gen
 | ...
 | [ei] = evs || gen & ... & gen

 accumulate k_ops

 | eis gets evs || gen & ... & gen
 | ...
 | eis gets evs || gen & ... & gen }
```

This returns a tuple containing k arrays ($k = k1 + k2 + ... + kM$). The first $k1$ arrays have bounds eBounds1 and are initialized according to the first set of clauses, the next $k2$ arrays have bounds eBounds2 and are initialized according to the second set of clauses, and so on.

After the keyword accumulate, the expression k_ops returns a tuple of k operators that are the accumulation operators.

The final set of clauses specifies the indices and values for the accumulation. In each clause, eis is a k-tuple of indices i1,...,ik, and evs is a k-tuple of values v1,...,vk, specifying the accumulation:

```
X1[i1] := op1 X[i1] v1
...
Xk[ik] := opk X[ik] vk
```

The number of accumulations implicitly is the total cardinality of all the environment-sequences produced by all the generators of the accumulation clauses. The array value of the entire expression is returned only after all the accumulations have been done.

Note: since the order in which the accumulation operations are performed is non-deterministic, it is the programmer's responsibility to ensure that the accumulation operators have the following property:

```
(op (op x y) z) = (op (op x z) y)
```

so that the whole construct is deterministic.

Example

A 10-category histogram of a zillion things:

```
{array (1,10)
| [i] = 0 || i <- 1 to 10
accumulate (+)
| (classify x) gets 1 || x <- zillion_things }
```

2.30 Abstract Types

A new abstract data type is declared using this statement:

```
abstype NEWTYPE
 typeof x1 = TYPE1 ;
 ...
 typeof xN = TYPEN
 rep
   REPRESENTATION-TYPE
   {
     ...
     def x1 = ... ;
     ...
     def xN = ... ;
     ...
   }
```

NEWTYPE is the (possibly parameterized) new type expression.

The subsequent typeof statements specify the *signature* (or interface) of the abstract type.

The REPRESENTATION-TYPE is a type-expression specifying the internal representation of objects of the new type.

The statements in the braces specify definitions for the identifiers in the signature. There may be other identifiers defined in the braces, but they are not exported— they are local types, local definitions *etc.* for the xJs.

The net effect of the abstype statement is to introduce the new type identifier and the identifiers x1 through xN into the current scope. Each xJ has the specified type signature and bound value.

Within the braces, the abstract type is treated as equivalent to the representation type. Outside the abstype statement, the abstract type and the representation type are treated as distinct (different) types.

Example

A stack, with a list representation:

```
abstype (stack *0)
  typeof empty  = (stack *0);
  typeof empty? = (stack *0) -> B;
  typeof push   = *0 -> (stack *0) -> (stack *0);
  typeof pop    = (stack *0) -> (stack *0);
  typeof top    = (stack *0) -> *0
 rep (list *0)
 {
   empty = nil ;
   empty? = nil? ;
   push = (:) ;

   def pop (x:s) = s
    | pop nil    = error "Stack underflow" ;

   def top (x:s) = x
    | top nil    = error "Stack underflow"
 }
```

2.31 Loops

While list- and array-comprehensions are convenient for expressing "mapping" operations over sequences, loops are convenient for expressing "reduction" operations.

The general while-loop expression form is:

```
{while eb do
   STATEMENT ;
   ...
   STATEMENT
 finally e}
```

where eb::B.

Assuming

```
eIndex :: (list N)
```

then the general for-loop expression form is:

```
{for x <- eIndex do
   STATEMENT ;
   ...
   STATEMENT
 finally e}
```

which is equivalent to:

```
{ L = eIndex
In
  {while (L <> nil) do
     x:(next L) = L ;
     STATEMENT ;
     ...
     STATEMENT
   finally e}}
```

Here, eIndex is normally an arithmetic-series expression (see Section 2.27.2).

The braces are compulsory. The type of the entire loop expression is the type of the expression in the "finally" phrase.

The loop body is a series of statements, with the following extension: a binding occurrence of an identifier (say "x") may be prefixed by the keyword "next", denoting the value to be used for "x" in the next iteration. This value is also available in the current iteration because "next x" may be used as an expression.

2.31.1 Scope of Variables in Loops

We use the phrase *loop context* to refer to the set of variables available to the loop expression from the surrounding scope. Any variable "x" from the loop context takes on a new value at each iteration if there is a "next x" binding in the loop body.

In while-loops, the predicate may only use identifiers from the loop context, and is re-evaluated each time *before* entering the loop body.

The loop is *terminated* in while loops when the predicate evaluates to false. Then, the finally e expression is evaluated and returned as the value of the loop. It may only use identifiers from the loop context.

Within the loop body, only variables from the loop context may be "next-ified". The loop body may also contain ordinary identifier bindings. The scope of all bindings is the entire loop body (this includes the next-ified variables, since "next x" may be used as an expression within the body).

For any next-ified identifier "x", the bound value becomes the value of "x" at the end of the iteration.

Examples

Successive approximations until convergence to a limit:

```
{ approx    = first_guess ;
  delx = infinity
In
  {while (delx > epsilon) do
     next approx = improve approx
     next delx = (next approx)-approx
   finally x}}
```

The n'th Fibonacci:

```
{ x,y = 1,1
In
  {for j <- 1 to n do
     next x,next y = y, x+y
   finally x}}
```

2.32 Errors

The (pseudo-) function:

```
error :: S -> *0
```

always creates a run-time error. The argument string should be a meaningful error-message.

2.33 Pragmatics

A function definition

```
def ...
```

may also be written:

```
defsubst ...
```

in which case the compiler will try to expand the function in-place wherever possible (But see Appendix B). This has no semantic consequence; it merely removes the overhead of function-calls.

The substitution is semantically transparent (it is not a macro). The function itself is still available as a value.

2.34 Annotations for Delayed Evaluation

Annotations for delayed evaluation are currently experimental features of Id to gain experience with infinite structures. Semantically, their only effect is to change the termination behavior of programs. Pragmatically, they can drastically change the run-time resource requirements of a program.

2.34.1 General Delayed Evaluation

Assuming:

```
e :: t
```

is an expression that evaluates to `v`, then the expression:

```
{# e} :: t
```

returns `d`, an unevaluated representation of `e` called a *thunk*.

The standard (pseudo-) function:

```
force :: *0 -> *0
```

takes a thunk `d`, evaluates the delayed expression in it, and returns `v`, its value. It also "memoizes" the value, so that in multiple evaluations of `(force d)`, the delayed expression itself is evaluated only once.

There is no implicit forcing. Delayed objects must be explicitly forced, and it is an error to force a non-delayed object. Thus, the first two expressions below are correct, the latter two are incorrect:

```
1 + 5
1 + (force {# 5})

1 + (force 5)  % forcing non-thunk
1 + {# 5}      % no implicit forcing
```

The function:

```
delayed? :: *0 -> B
```

may be used to test whether an object is a thunk or not. Note: this is not the same as a (non-deterministic) test of whether it has been forced yet or not.

2.35 Delayed Evaluation Tied to Data Structures

When delayed evaluation is tied to data-structures, it is often more convenient (implicit forcing) and more efficient (less space overhead for thunks).

First, we define *Contructor Terms* as applicative forms:

```
c e1 ... eN
```

where `c` is a constructor of arity N of some algebraic type. Example:

```
e1 : e2
```

but not:

```
(:) e1              % arity not satisfied
```

In a constructor term, any argument may be annotated by "#" to indicate that it should be delayed. Examples:

```
 e1 :  e2     % eager head, eager tail
 e1 : #e2     % eager head, delayed tail
#e1 :  e2     % delayed head, eager tail
#e1 : #e2     % delayed head and delayed tail
```

The delayed components will be evaluated automatically (and stored in the data structure) when an attempt is made to select it (usually in some pattern-match).

Note: the following:

```
f = (:) ;
y = f e1 #e2
```

is a *compile-time* error because `(f e1 e2)` is not a constructor term— `f` is not a constructor identifier.

The "`#`" annotations in constructor terms are *not* equivalent to using `{# ...}` and `force`. Consider:

(A) `e1:{# e2}`

(B) `e1:# e2`

In (A), the tail slot of the cons cell contains a reference to a thunk. When the tail is selected, this pointer is returned, which must then be explicitly forced to get the value of `e2`.

In (B), the thunk is stored directly in the tail slot of the cons cell. When the tail is selected, the the thunk is automatically evaluated, and the value replaces the thunk in the cell.

3 Non-functional Constructs

Warning: Programs that use constructs from this section are not likely to be referentially transparent (purely functional).

3.1 Void

Some constructs can be written in syntactic shorthand in which certain expressions may be omitted. Usually, the missing expression is equivalent to the expression

```
voidvalue :: void
```

`voidvalue` is a "useless" value— there are no interesting operations defined on it. It is an error to bind this value to an identifier, apply a function to it, store it in an array, *etc.* (But see Appendix B.)

We encourage the programmer to think of expressions of type `void` as "returning no value".

3.2 I-structures

Array comprehensions (Section 2.28.4) specify two things simultaneously— the "shape" of a data-structure (*i.e.*, index bounds) and its contents.

In I-structures, these two specifications are separated. An I-structure is an array-like data structure with empty locations which can be assigned subsequently.

3.2.1 I-structure Types

An n-dimensional I-structure whose components are of type `t` has type:

```
nD_I_array t
```

Synonyms for the type name `1D_I_array`:

`I_vector` `I_array`

Synonym for the type name `2D_I_array`:

`I_matrix`

3.2.2 I-structure Creation

An n-dimensional I-structure is created using:

```
nD_I_array::((N,N),...,(N,N)) -> (nD_I_array *0)
```

i.e., it takes an index-bounds expression (an n-tuple of integer 2-tuples) and returns an empty I-structure with those bounds.

Synonyms for `1D_I_array` I-structure allocator:
`I_vector` `I_array`

Synonym for `2D_I_array` I-structure allocator:
`I_matrix`

Example

A 2-dimensional 10×10 I-structure:

```
I_matrix ((1,10),(1,10))
```

3.2.3 I-structure Assignments

Assuming:

```
a    :: (nD_I_array t)
e1   :: (N,...,N)
e2   :: t
```

then the I-structure assignment statement:

```
a[e1] = e2
```

assigns the value of "`e2`" to the `(j1,...,jn)`'th component of the I-structure "`a`", where `(j1,...,jn)` is the value of "`e1`". (But see Appendix B).

The I-structure in the lhs *must* be designated by an identifier and not by an arbitrary expression.

A runtime error occurs if a component of an I-structure is assigned more than once. Thus, every location makes at most one transition from the "empty" state (*i.e.*, containing $\perp$) to the "full" state (*i.e.*, containing some value $v \sqsupset \perp$).

There is no race-condition between selections and I-structure assignments. A selection `a[j]` returns a value only after the location is full.

3.2.4 Delayed I-structure Assignment

This is an experimental feature of Id (See Secti ruc-ture selection expression:

```
a[e1]    :: t
```

returns the value of the `(j1,...,jn)`'th component of the I-structure "`a`", where `(j1,...,jn)` is the value of "`e1`" (But see Appendix B).

There is no race-condition between selections and I-structure assignments. A selection `a[j]` returns a value only after the location is full.

3.2.5 I-structure Index Bounds

For each $n \geq 1$, there is a function that returns the index bounds of n-dimensional I-structures:

```
1D_bounds :: (1D_I_array t) -> (N,N)
2D_bounds :: (2D_I_array t) -> ((N,N),(N,N))
...
```

Synonym for `1D_bounds`:
`bounds`

See also Appendix A.9 for standard I-structure functions.

3.3 One-Armed Conditionals

Assuming:

```
e1 :: B
e2 :: void
```

then the one-armed conditional expression is:

```
if e1 then e2    :: void
```

An `else` matches the nearest preceding unbalanced `then`. Leaving out the `else` clause is syntactic short-hand for saying
`else voidvalue`

3.4 Call Statements

Call statements contain expressions that are executed only for their side-effect (*i.e.*, their I-structure assignments):

```
call e
```

Though not required, `e` normally has type `void`.

If the expression `e` is a conditional, a loop or a block, the `call` keyword may be omitted.

3.5 Block Statements

The phrase "`in e`" in a Block-expression may be omitted. This is syntactic shorthand for saying:

```
in voidvalue
```

3.6 Loop Statements

The "`finally e`" phrase may be omitted; this is syntactic shorthand for saying

```
finally voidvalue
```

Example

An I-structure containing j^2 at the j'th index:

```
{  x = I_structure (1,10)
   {for i <- 1 to 10 do
      x[i] = j * j}
In
   x}
```

We strongly recommend against loops where variables depend on computations in future iterations, *e.g.*,

```
{ A[10] = 0 ;
  {for j from 1 to 9 do
     A[j] = f A[j+1] }}
```

Our implementation will not guarantee arbitrary look-ahead, *i.e.*, loop unfolding.

A Standard Identifiers

Id has standard libraries that implement many useful functions. The names and semantics of these functions are based on the corresponding Common Lisp functions wherever possible. The names for these functions are not reserved words, but for readability and re-usability of code, the programmer is strongly advised *not* to redefine them.

The compiler will usually expand these functions *in situ*, so that there is no procedure-calling overhead.

Since the Id libraries are a continuously growing repository of useful functions, the following list is necessarily incomplete. The libraries themselves must be consulted for the current set.

A.1 Booleans

- Truth values:

```
typeof true = B
typeof false = B
```

These are also constructors (can be used in patterns).

- Negation:

```
typeof not = B -> B
```

A.2 Numbers

- General:

```
typeof  pi = N
typeof 2pi = N
```

```
typeof odd? = N -> B
typeof even? = N -> B
```

```
typeof max = N -> N -> N
typeof min = N -> N -> N
```

- Exponentiation:

```
typeof exp = N -> N
```

where `(exp y)` $\Rightarrow e^y$

- Logarithms:

```
typeof log = N -> N
typeof log10 = N -> N
```

where `(log x)` $\Rightarrow \log_e x$,
and `(log10 x)` $\Rightarrow \log_{10} x$

- Square root, absolute value:

```
typeof sqrt = N -> N
typeof abs  = N -> N
```

- Trignometric functions (angles in radians):

```
typeof sin  = N -> N
typeof cos  = N -> N
typeof tan  = N -> N
```

```
typeof asin = N -> N
typeof acos = N -> N
typeof atan = N -> N -> N
```

where `(atan y x)` $\Rightarrow \arctan y/x$, in the range $-\pi$ to $+\pi$. The arguments cannot both be zero.

- Hyperbolic functions:

```
typeof sinh  = N -> N
typeof cosh  = N -> N
typeof tanh  = N -> N
```

```
typeof asinh = N -> N
typeof acosh = N -> N
typeof atanh = N -> N -> N
```

- Conversion to integers:

```
typeof floor = N -> N
```

where `floor` truncates towards $-\infty$.

```
typeof ceiling = N -> N
```

where `ceiling` truncates towards $+\infty$.

```
typeof truncate = N -> N
```

where `truncate` truncates towards 0.

```
typeof round = N -> N
```

where `round` truncates to the nearest integer, with $x.5$ truncated towards the even integer.

```
typeof mod = N -> N -> N
```

where `(mod x y)` $\Rightarrow x - qy$, where ($q =$ `floor (x/y)`).

```
typeof rem = N -> N -> N
```

where `(rem x y)` $\Rightarrow x-qy$, where ($q =$ `truncate(x/y)`).

A.3 Characters

- Character functions:

```
typeof digit?  = C -> B
typeof uc?     = C -> B
typeof lc?     = C -> B
typeof C_to_uc = C -> C
typeof C_to_lc = C -> C
typeof C_to_N  = C -> N
typeof N_to_C  = N -> C
```

A.4 Strings

- Convert to and from arrays of characters:

```
typeof array_to_S = (array C) -> S
```

The argument must have index bounds $(0, n-1)$ when n is the length of the string.

```
typeof S_to_array = S -> (array C)
```

The result has index bounds $(0, n-1)$ when n is the length of the string.

- Convert to and from lists of characters:

```
typeof list_to_S = (list C) -> S
typeof S_to_list = S -> (list C)
```

- Length of a string:

```
typeof S_length = S -> N
```

- Indexing a string (first character has index 0):

```
typeof S_nth = S -> N -> C
```

- Extract a substring, given a starting position and substring length:

```
typeof substring = S -> N -> N -> S
```

- Concatenate two strings:

```
typeof S_conc = S -> S -> S
```

- Map a character function over a string:

```
typeof S_map = (C -> C) -> S -> S
```

- Convert a string to upper- or lower-case:

```
typeof S_to_uc = S -> S
typeof S_to_lc = S -> S
```

A.5 Lists

- Basic functions:

```
typeof nil    = (list *0)
typeof nil?   = (list *0) -> B
typeof cons   = *0 -> (list *0) -> (list *0)
typeof hd     = (list *0) -> *0
typeof tl     = (list *0) -> (list *0)
typeof length = (list *0) -> N
```

- Last element of a list:

```
typeof last = (list *0) -> *0
```

- N'th tail of list (*i.e.*, $\texttt{tl}^n$, so 0'th tail is the list itself):

```
typeof nthtl = N -> (list *0) -> (list *0)
```

- Zipping and unzipping lists— a family of functions, for each N:

```
typeof zipN =
 (list *1) ->
 ...
 (list *N) -> (list (*1,...,*N))
```

It is an error if the N input lists are not of equal length.

```
typeof unzipN =
 (list (*1,...,*N)) ->
                (list *1,
                 ...    ,
                 list *N)
```

• Reverse a list:

```
typeof reverse = (list *0) -> (list *0)
```

• Apply a function to each member of a list, returning list of results in same order:

```
typeof map_list =
 (*0->*1) -> (list *0) -> (list *1)
```

• Filter a list, retaining only those elements that satisfy a

```
typeof filter =
 (*0 -> B) -> (list *0) -> (list *0)
```

• Left-associative reduction:

```
typeof foldl_list =
 (*0 -> *1 -> *0) ->
 *0 ->
 (list *1)          -> *0
```

Example:

```
foldl_list f z l
```

returns

```
f (f  (... (f z l0) l1) ...) ln
```

where `l0`, ..., `ln` are the elements of the list `l`.

• Right-associative reduction:

```
typeof foldr_list =
 (*0 -> *1 -> *1) ->
 *1 ->
 (list *0)          -> *1
```

Example:

```
foldr_list f z l
```

Returns

```
f l0 (... (f ln z))
```

where `l0`, ..., `ln` are the elements of the list `l`.

• Iteration:

```
typeof iterate =
 (*0 -> B) ->
 (*0 -> *0) ->
 *0          -> (list *0)
```

where `iterate p f x` returns the list containing `x`, `(f x)`, `(f (f x))`, ..., as long as `(p (`f^n` x))` is true.

• Simultaneous mapping and left-associative reduction of a list:

```
typeof map_foldl_list =
 (*0->*1->(*0,*2)) ->
 *0 ->
 (list *1)          -> (*0,list *2)
```

Example:

```
map_foldl_list f z l
```

returns `(zN,m)`, where:

```
z0,m0 = f z  l0
z1,m1 = f z0 l1
  ...
zN,mN = ...
```

`l0`, ..., `lN` are the elements of the list `l`, and `m0`, ..., `mN` are the elements of the list `m`.

For example, if `f` was

```
def f z lj = {  w = z + lj
            IN w,w} ;
```

`z` was 0, and `l` contained `1`, `2`, and `3`, then the result `m` would be a list of partial sums: `1`, `3`, and `6`, and the result `zN` would be the sum `6`.

• Simultaneous mapping and right-associative reduction of a list:

```
typeof map_foldr_list =
 (*1->*0->(*2,*0)) ->
 *0 ->
 (list *1)          -> (list *2,*0)
```

Example:

```
map_foldr_list f z l
```

returns `(m,z0)`, where:

```
m0,z0 = f l0 z1
m1,z1 = f l1 z2
  ...
mN,zN = f lN z
```

`l0`, ..., `lN` are the elements of the list `l`, and `m0`, ..., `mN` are the elements of the list `m`.

For example, if `f` was

```
def f lj z = {  w = z + lj
            IN w,w} ;
```

`z` was 0, and `l` contained `1`, `2`, and `3`, then the result `m` would be a list of partial sums (from back to front): `6`, `5`, and `3`, and the result `z0` would be the sum `6`.

A.6 Lists as Sets

All these functions require, as their first parameter, an equality function between elements of the sets.

- Conversion from list to set (remove duplicates):

```
typeof settify =
 (*0 -> *0 -> B) ->
 (list *0)           -> (list *0)
```

- Membership test:

```
typeof member? =
 (*0 -> *0 ->B) ->
 *0 ->
 (list *0)          -> B
```

- Union, intersection, difference:

```
typeof union =
 (*0 -> *0 -> B) ->
 (list *0) ->
 (list *0)           -> (list *0)
typeof intersection =
 (*0 -> *0 -> B) ->
 (list *0) ->
 (list *0)           -> (list *0)
typeof difference =
 (*0 -> *0 -> B) ->
 (list *0) ->
 (list *0)           -> (list *0)
```

- Subset test:

```
typeof subset? =
 (*0 -> *0 -> B) ->
 (list *0) ->
 (list *0)           -> B
```

- Set equality test:

```
typeof set_equal? =
 (*0 -> *0 -> B) ->
 (list *0) ->
 (list *0)           -> B
```

A.7 Arrays

In the following, we describe families of functions, for 1D arrays, 2D arrays, *etc.* We describe the entire family using the "`nD`" meta-syntax. In addition, the substring `1D_array` can always be replaced by `array` or `vector`, and the substring `2D_array` can always be replaced by `matrix`.

We refer to a sequence of indices for an array by its endpoints "`first`" and "`last`", meaning n-tuples containing the lower bounds and upper bounds, respectively, along all dimensions, and stepping the rightmost index fastest.

- Index bounds:

```
typeof nD_bounds =
 (ND_array *0) -> ((N,N),...,(N,N))
```

Synonyms:

```
1D_bounds  vector_bounds  array_bounds
2D_bounds  matrix_bounds
```

- Create k arrays, given a "filling" function:

```
typeof make_k_nD_arrays =
 ((N,N),...,(N,N)) ->
 ((N,...,N) -> (*1,...,*k) -> (nD_array *0,
                               ... ,
                               nD_array *k)
```

Example:

```
make_k_nD_arrays b f
```

returns k arrays `a1,...,ak` with bounds `b`, such that if

```
f (j1,...,jN) == (v1,...,vk)
```

then

```
ai[j1,...,jN] == vi
```

Synonyms:

	$k = 1$	$k > 1$
$n = 1$	make_array make_vector	make_k_arrays make_k_vectors
$n = 2$	make_matrix	make_k_matrices
$n \geq 1$	make_nD_array	

• Map a function over an array:

```
typeof map_nD_array =
 (*0 -> *1) ->
 (nD_array *0) -> (nD_array *1)
```

Example:

```
map_nD_array f a
```

returns an array with same bounds as array `a`, containing `(f a[j])` at each index `j`.

• Left-associative reduction over an array:

```
typeof foldl_nD_array =
 (*0 -> *1 -> *0) ->
 *0 ->
 (nD_array *1)        -> *0
```

Example:

```
foldl_nD_array f z a
```

returns:

```
f (f ... (f z a[first]) ... ) a[last]
```

• Right-associative reduction over an array:

```
typeof foldr_nD_array =
 (*0 -> *1 -> *1) ->
 *1 ->
 (nD_array *0)        -> *1
```

Example:

```
foldr_nD_array f z a
```

returns:

```
f a[first] ( ... (f a[last] z))
```

• Tree-reduction over an array:

```
typeof fold_nD_array =
 (*0 -> *1 -> *0) ->
 *0 ->
 (nD_array *1)        -> *0
```

Example:

```
fold_nD_array f z a
```

reduces the array to a value by first computing the `foldl`s of all the innermost vectors (rightmost index varying), then the `foldl`s of those results with the next innermost index varying, and so on. `fold_...` has more parallelism than `foldl_...` and `foldr_...`.

• Simultaneous mapping and left-associative reduction of an array:

```
typeof map_foldl_nD_array =
 (*0->*1->(*0,*2)) ->
 *0 ->
 (array *1)        -> (*0,array *2)
```

Example:

```
map_foldl_nD_array f z a
```

returns `(zLast,b)`, where `b` is an array with same bounds as array `a`, and

```
zFirst, b[first]  = f z           a[first]
zSecond,b[second] = f zFirst      a[second]
...
zLast,  b[last]   = f zLastButOne a[last]
```

For example, if `f` was

```
def f z aj = {  w = z + aj
               IN w,w} ;
```

z was 0, and a was a vector containing 1, 2 and 3, then the result b would be a vector of partial sums: 1, 3 and 6, and the result zLast would be the sum 6.

•Simultaneous mapping and right-associative reduction of an array:

```
typeof map_foldr_nD_array =
 (*1->*0->(*2,*0)) ->
 *0 ->
 (array *1)         -> (array *2,*0)
```

Example:

```
map_foldr_nD_array f z a
```

returns (b,zFirst), where b is an array with same bounds as array a, and

```
b[first],  zFirst  = f a[first]  zSecond
b[second],zSecond = f a[second] zThird
...
b[last],   zLast   = f a[last]   z
```

For example, if f was

```
def f z aj = {  w = z + aj
              IN w,w} ;
```

z was 0, and a was a vector containing 1, 2 and 3, then the result b would be a vector of partial sums (from last to first): 6, 5 and 3, and the result zFirst would be the sum 6.

A.8 Delayed Evaluation

```
typeof force    = *0 -> *0
typeof delayed? = *0 -> B
```

A.9 I-structures

• I-structure allocators:

```
typeof nD_I_array =
  ((N,N),...,(N,N)) -> (nD_I_array *0)
```

Synonyms for 1D_I_array: I_vector, I_array

Synonym for 2D_I_array: I_matrix

• I-structure index bounds:

```
typeof nD_bounds =
 (nD_I_array *0) -> ((N,N),...,(N,N))
```

Synonym for 1D_bounds: bounds

• Fill an empty rectangular region of k existing arrays, given a "filling" function:

```
typeof fill_k_nD_arrays =
 ((N,N),...,(N,N)) ->
 ((N,...,N) -> (*1,...,*k) ->
 (nD_array *0, ... , nD_array *k) -> void
```

Example:

```
fill_k_nD_arrays r f (a1,...,ak)
```

fills region r of arrays a1,...,ak such that if

```
f (j1,...,jN) == (v1,...,vk)
```

then

```
ai[j1,...,jN] == vi
```

B Notes on the Current Implementation

This section describes some restrictions and quirks in the current implementation.

Void: Even though it is an error to use the void value in any way, it will not currently be caught as an error.

Multi-dimensional arrays: (type (nD_array t) where $n > 1$). Currently, these are not correctly implemented. They are implemented only as nested 1-dimensional arrays.

I-structure Index Expressions: Currently, the index expression "e" for an n-dimensional array must *syntactically* be an n-tuple; it cannot be an arbitrary expression that returns an n-tuple.

Top-Level Definitions: Currently, pattern-bindings are not allowed as top-level STATEMENTS in a program.

References

[1] Arvind, K. P. Gostelow, and W. Plouffe. *An Asynchronous Programming Language and Computing Machine.* Technical Report 114a, Department of Information and Computer Science, University of California, Irvine, CA, December 1978.

[2] Arvind, R. S. Nikhil, and K. K. Pingali. *Id Nouveau Reference Manual, Part II: Semantics.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[3] R. S. Nikhil. *Id Nouveau Reference Manual, Part I: Syntax.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[4] R. S. Nikhil and Arvind. *Id/83s.* Technical Report, MIT Laboratory for Computer Science, Cambridge, MA 02139, July 1985. (Prepared for MIT Subject 6.83s).

[5] R. S. Nikhil, K. K. Pingali, and Arvind. *Id Nouveau.* Technical Report CSG Memo 265, MIT Laboratory for Computer Science, Cambridge, MA 02139, July 23 1986. (Prepared for MIT Subject 6.83s).

Contents