REPORT ON THE FX-91 PROGRAMMING LANGUAGE

David K. Gifford
Pierre Jouvelot
Mark A. Sheldon
James W. O'Toole

February 1992
Report on the FX-91 Programming Language

DAVID K. GIFFORD, PIERRE JOUVELOT, MARK A. SHELDON, AND JAMES W. O'TOOLE
PROGRAMMING SYSTEMS RESEARCH GROUP
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

SUMMARY
This report gives a defining description of the programming language FX-91. The FX (short for FX-91) programming language is designed to support the parallel implementation of applications that perform both symbolic and scientific computations. The unique features of FX include:

• An effect system, to discover expression scheduling constraints. An effect is a static description of the side-effects an expression may perform when it is evaluated. Just as a type describes what an expression computes, an effect describes how an expression computes.

• Abstraction over any kind of description, thus permitting first-class type and effect polymorphism. Effect polymorphism makes the FX effect system more powerful than previous approaches to side-effect analysis in the presence of first-class subroutines.

• Type and effect inference, so that declaration free programs can be statically type and effect checked. FX also permits explicitly typed programs, and programs that use explicit types only for first-class polymorphic values and modules.

• First-class modules, which permit FX to serve as its own configuration language. It also includes an architecture independent module of parallel vector operators.

The introduction offers a summary of and motivation for the unique properties of FX-91.

• Chapter 1 presents the fundamental ideas of the language and describes the notational conventions used for describing the language and for writing programs in the language.

• Chapter 2 describes the FX-91 Kernel. The FX Kernel includes essential constructs and the type and effect system.

• Chapter 3 introduces built-in data types and operations, which include all of the language's data manipulation and input-output primitives.

CONTENTS

Introduction ................................ 2
1. Overview of FX .......................... 3
   1.1. Semantics .......................... 3
   1.2. Lexicon ........................... 3
   1.3. Static and Dynamic Errors ........ 4
   1.4. Conventions ....................... 4
2. The FX-91 Kernel ....................... 5
   2.1. Kinds .............................. 5
   2.2. Descriptions ...................... 5
   2.3. Values ............................. 8
   2.4. Sugars ............................. 13
3. Standard Descriptions ................. 16
   3.1. Pure ................................ 16
   3.2. Init ................................ 16
   3.3. Read ............................... 16
   3.4. Write .............................. 16
   3.5. Unit ............................... 16
   3.6. Bool ............................... 16
   3.7. Int ................................ 16
   3.8. Float .............................. 17
   3.9. Char ............................... 17
   3.10. String ............................ 18
   3.11. Sym ............................... 18
   3.12. Permutation ...................... 19
   3.13. Refof ............................ 19
   3.14. Uniqueof .......................... 19
   3.15. Listof ............................ 19
   3.16. Vectorof .......................... 20
   3.17. Sexp ............................. 21
   3.18. Stream ............................ 22
References ................................ 23
Alphabetic index of definitions of concepts, keywords, and procedures .... 24

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-89-J-1988

Pierre Jouvelot is also with the Centre de Recherche en Informatique of Ecole des Mines de Paris, France.
Report on the FX-91 Programming Language

David K. Gifford
Pierre Jouvelot
Mark A. Sheldon
James W. O'Toole

Programming Systems Research Group
Laboratory for Computer Science

February 1992

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-89-J-1988

Pierre Jouvelot is also with the Centre de Recherche en Informatique of Ecole des Mines de Paris, France.

Keywords: Programming Languages, Types, Effects, Inference, Polymorphism, Modules, Static Checking, Vector Operations

©1992 Massachusetts Institute of Technology
INTRODUCTION

FX-91 is a programming language that we designed to investigate the following questions:

- How can simple syntactic rules be used to deduce program properties beyond type information?
- How important is information about the side-effects of program expressions in a language that is designed for parallel computing, and to what extent can unambiguous side-effect information be used to schedule a program for parallel execution?
- How important are first-class polymorphic values and first-class modules in a language that provides type inference?

FX-91 is a major revision and extension of the FX-87 programming language [GJLS87]. The designs of both FX-91 and FX-87 were strongly influenced by Scheme [R86], especially in the choice of standard types and operations.

FX-87 was the first programming language to incorporate an effect system [LG88]. Experimental data from FX-87 programs show that effect information can be used to automatically schedule imperative programs for parallel execution [HG88]. However, we found that FX-87 was difficult to use because extensive declarations were required in programs.

FX-91 is designed to be easier to use than FX-87. FX-91 eliminates the requirement for most declarations [OG89, JG91], provides a less complex effect system, and provides a module system that supports programming in the large [SG90].

We have found that an effect system is useful to programmers, compiler writers, and language designers in the following respects:

- An effect system lets the programmer specify the side-effect properties of program modules in a way that is machine-verifiable. The resulting effect specifications are a natural extension of the type specifications found in conventional programming languages. We believe that effect specifications have the potential to improve the design and maintenance of imperative programs.
- An effect system lets the compiler identify optimization opportunities that are hard to detect in a conventional higher-order imperative programming language. We have focused our research on three classes of optimizations: execution time (including eager, lazy, and parallel evaluation); common subexpression elimination (including memoization); and dead code elimination. We believe that the ability to perform these optimizations effectively in the presence of side-effects represents a step towards integrating functional and imperative programming for the purpose of parallel programming.

An effect system lets the language designer express and enforce side-effect constraints in the language definition. In FX, for example, the body of a polymorphic expression must not have any side-effects. This restriction makes FX the first language known to us that permits an efficient implementation of fully orthogonal polymorphism in the presence of side-effects. In FX, any expression can be abstracted over any type and all polymorphic values are first-class. First-class values can be passed to subroutines, returned from subroutines, and placed in the store.

The FX-91 programming language was developed by the Programming Systems Research Group at MIT. In addition to the authors, Jonathan Rees and Franklyn Turbak contributed to the design of FX-91. Any information or comments about FX-91 can be submitted to the FX electronic mailing list fx@lcs.mit.edu. Send requests to be added to the list to fx-request@lcs.mit.edu.

An FX-91 interpreter written in Scheme can be obtained by sending an electronic mail request to fx-request@lcs.mit.edu.


DESCRIPTION OF THE LANGUAGE

1. Overview of \textit{FX}

\textit{FX} uses lambda abstraction and beta-reduction as the basis of its computational model, and thus it is a member of the lambda calculus family of languages. \textit{FX} uses symbolic expression (s-expression) syntax, and thus it is compatible with Lisp source maintenance tools. \textit{FX} is lexically scoped, statically checked, uses one variable namespace and implements tail-recursion. All values in \textit{FX} are first-class, including subroutines, polymorphic values and modules.

The \textit{FX} programming system is based on a \textit{kernel} language that defines the syntax and semantics of a core set of primitive \textit{FX} expressions. The kernel is primitive in the sense that it defines twenty different value expressions, and there is no simple way to express these expressions in terms of one another. Thus the \textit{FX} kernel forms the core of the \textit{FX} programming system from the point of view of both the \textit{FX} application programmer and the \textit{FX} language implementor.

The foundation provided by the \textit{FX} kernel is supplemented with a library of standard types and operators that are contained in the \textit{fx} module. The \textit{fx} module contains types and operations for booleans, integers, floating point numbers, characters, strings, symbols, permutations, unique values, lists, vectors, symbolic expressions and input-output streams. The \textit{fx} module can be defined in terms of kernel expressions, and can be replaced by programmers who wish to change the implementation of standard types.

1.1. Semantics

The semantic definition of the \textit{FX} kernel is divided into a \textit{static semantics} that is used to deduce the properties of programs before they are run and a \textit{dynamic semantics} that describes the behavior of programs at execution time.

There are two key theorems that relate the static and dynamic semantics of \textit{FX}. The \textit{type soundness} theorem guarantees that the type of an expression (the type computed by the static semantics) will be a conservative approximation of its dynamic type (the type of the value computed by the dynamic semantics). The \textit{effect soundness} theorem guarantees that the static effect of an expression will be a conservative approximation of its dynamic effect. These theorems permit results from the static semantics to be used by \textit{FX} implementations to improve dynamic performance.

The \textit{FX} static semantics is based on a hierarchical kinded type system that includes kinds, universal polymorphism, higher order types, and recursive types. The static semantics describes expressions with \textit{description expressions}. There are two principle kinds of descriptions: \textit{types}, which describe the values expressions compute, and \textit{effects}, which describe the side-effects of expressions. An expression may be polymorphic in any kind of description. Thus the type of a subroutine may depend on the effect parameters passed to it. Effect polymorphism permits the static semantics to provide tight effect bounds on higher-order functionals in a natural and simple manner.

The \textit{FX} static semantics will reconstruct omitted type and effect declarations in a manner that combines the implicit typing of ML[MT90] with the full power of the explicitly typed second-order polymorphic lambda calculus. The \textit{FX} reconstruction system relieves the programmer of the burden of providing type and effect declarations while retaining the benefits of strongly-typed languages, including superior performance, documentation, and safety. The \textit{FX} type reconstruction system will accept ML-style programs, explicitly typed programs, and programs that use explicit types only for first-class polymorphic values and modules. We offer this flexibility by providing both generic and explicitly-quantified polymorphic types in \textit{FX}, along with an operator to convert between these two forms of polymorphism.

The \textit{FX} static semantics provides complete checking of module values. The \textit{FX} module system permits types and values to be packaged as first-class module values. Because modules are first-class values, \textit{FX} does not require a separate configuration language.

1.2. Lexicon

The basic lexical entities used in the \textit{FX} programming language are the following:

- \textit{A digit} is one of $0 \ldots 9$.
- \textit{A letter} is one of $a \ldots z$ or $A \ldots Z$.
- The set of extended alphabetic characters must include: $*, /, <, =, >, !, ?, :, $, %, \ldots, \&, \pm, \ldots, [, ]$.
- \textit{A white space} is a blank space, a newline character, a tab character, or a newpage character.
- \textit{A character} is a digit, a letter, an extended alphabetic character, $+, -, \ldots$, a white space or backspace character.
- \textit{A delimiter} is a white space, a left parenthesis or a right parenthesis.
- \textit{A token} is a sequence of characters that is separated by delimiters.
- \textit{A number} is a token made of a non-empty sequence of digits, possibly including base and exponent information, a decimal point, and a sign. (see Chapter 3).
A literal is either a number, or a token that begins with ' or #, or a sequence of characters or \ enclosed in double quotes " or the symbol keyword and an identifier enclosed in parentheses.

An identifier is a token beginning with a letter or extended alphabetic character and made of a non-empty sequence of letters, digits, extended alphabetic characters, and the characters + and -. Note that + and - by themselves are also identifiers. Identifiers are case-insensitive.

FX reserves the following identifiers. Reserved identifiers must not be bound, redefined, or used as tags for sums.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>and</td>
</tr>
<tr>
<td>cond</td>
<td>begin</td>
</tr>
<tr>
<td>define-datatype</td>
<td>define-abstraction</td>
</tr>
<tr>
<td>desc</td>
<td>define-description</td>
</tr>
<tr>
<td>else</td>
<td>definetype</td>
</tr>
<tr>
<td>fx</td>
<td>effect</td>
</tr>
<tr>
<td>let</td>
<td>extract</td>
</tr>
<tr>
<td>load</td>
<td>lambda</td>
</tr>
<tr>
<td>module</td>
<td>let</td>
</tr>
<tr>
<td>or</td>
<td>match</td>
</tr>
<tr>
<td>product</td>
<td>moduleof</td>
</tr>
<tr>
<td>select</td>
<td>open</td>
</tr>
<tr>
<td>symbol</td>
<td>poly</td>
</tr>
<tr>
<td>type</td>
<td>proj</td>
</tr>
<tr>
<td>val</td>
<td>sumof</td>
</tr>
<tr>
<td>with</td>
<td>tagcase</td>
</tr>
</tbody>
</table>

Comments in FX are sequences of characters beginning with a ";" and ending with the end of the line on which the ";" is located. They are discarded by FX and treated as a single whitespace.

### 1.3. Static and Dynamic Errors

**Static errors** are detected by the FX static semantics. All syntax, type, and effect errors are detected statically and reported. The sentence "z must be y" indicates that "it is a static error if z is not y".

**Dynamic errors** may be detected by FX when a program is run. The phrase "a dynamic error is signalled" indicates that FX implementations must report the corresponding dynamic error and proceed in an implementation-dependent manner. The phrase "it is a dynamic error" indicates that FX implementations do not have to detect or report the corresponding dynamic error. The meaning of a program that contains a dynamic error is undefined.

### 1.4. Conventions

This report adheres to the following conventions:

- *FX* program text is written in **teletype font**. Program text is comprised of identifiers, literals, and delimiters.
- *Meta-expressions*, which are names for syntactic classes of expressions, are written in italic font. A programmer may replace any meta-expression by a compatible FX expression.
- Certain FX language forms have a variable number of components. A possibly empty sequence of n expressions is noted e₁...eₙ or ...eₙ. If the name of the upper bound on subscripts is not used, we write the shorter: e₁... If there is at least one expression in the sequence (i.e. n ≥ 1), we use e₁(...eₙ. We usually denote by eᵢ (or any other subscripted e) an expression belonging to such sequence. Certain parameters can have different forms. [z][y] stands for either z or y.

- The set of values z that satisfy the predicate P is noted {z | P(z)}; predicates are defined as usual. The difference of two sets S and T is noted S - T. For an ordered index set S, we note {eₛ | eₓ} the set of eₓ for each x of S. As a shorthand, {iₑ | eᵢ} is noted {eᵢ₁}. The interval of ordered values between x and y is noted [x, y]. If the lower bound is excluded, [x, y] is used instead; the same convention applies to upper bounds.

- The function that is equal to the function f, except at x (not in the domain of f) where it yields y, is noted f[x ← y]. As a shorthand, we note f₁,...,n that substitutes any free appearance of iᵢ in its argument by eᵢ; alpha-renaming of bound variables is performed to avoid name clashes. For an ordered index set S, we write [eₛ | eₓ/ₑᵢ] for the successive substitutions of iₑ by eᵢ for each x of S (the variables iₑ must be pairwise distinct). As a shorthand, [eₛ | eᵢ/ₑᵢ] is noted [e₁,...,eₙ].

- Universal quantification of a formula f(i) when i is in a given interval [1, n] is written f(i) (1 ≤ i ≤ n). This notation is straightforwardly extended to open and semi-open intervals.

- A deduction system is a set of rules written in the following way:
Declarations in description expressions are written using kind expressions. Kinds form the third and highest level of the language. Kinds are the "types" of descriptions, and every legal description expression has a kind.

A complete specification for each level of the FX Kernel follows.

2.1. Kinds

\[ k ::= \text{type} \mid \text{effect} \mid (\rightarrow \rightarrow k_1 \ldots k_n) \]

For each kind special form, we give its syntax in its section header and provide an informal description of its usage. Kinds have neither static nor dynamic semantics.

2.1.1. type

The kind expression type denotes the collection of descriptions that describe the values of computations (the so-called type expressions).

2.1.2. effect

The kind expression effect denotes the collection of descriptions that describe the side-effects of computations (the so-called effect expressions).

2.1.3. (\rightarrow \rightarrow k_1 \ldots k_n)

A \rightarrow \rightarrow expression denotes the collection of description functions that map descriptions of kind \( k_i \) to a type (the so-called type constructors).

2.2. Descriptions

\[ t_i ::= id \mid (di \; di_1 \ldots di_n) \mid (-> \; ei \; ((id_1 \; ti_1) \ldots (id_n \; ti_n)) \; ti_{n+1}) \mid (\text{productof} \; (id_1 \; ti_1) \ldots (id_n \; ti_n)) \mid (\text{sumof} \; (id_1 \; ti_1) \ldots (id_n \; ti_n)) \mid (\text{moduleof} \; (\text{abs} \; id_1 \; k_1) \ldots (\text{abs} \; id_n \; k_n) \; \text{desc} \; id_1 \; dz_1 \ldots \text{desc} \; id_p \; dz_p) \; (\text{val} \; idv_1 \; tx_1) \ldots (\text{val} \; idv_m \; tx_m)) \mid (\text{poly} \; ((id_1 \; k_1) \ldots (id_n \; k_n)) \; tz) \mid (\text{productof} \; (id_1 \; tz_1) \ldots (id_n \; tz_n)) \mid (\text{sumof} \; (id_1 \; tz_1) \ldots (id_n \; tz_n)) \mid ti \]

\[ ei ::= id \mid (\text{maxeff} \; ei_1 \ldots ei_n) \]

\[ dz ::= tz \mid (\text{dlambda} \; ((id_1 \; k_1) \ldots (id_n \; k_n)) \; tz) \mid di \]

\[ di ::= ti \mid ei \mid id \mid (\text{select} \; e \; id) \]
The syntax of expressions e is given below. Meta-variables that use i in their names (instead of z) denote description classes that can be omitted from user programs; they will be automatically inferred by the FX type and effect inference system. Such descriptions are said to be inferable. The class of inferable descriptions is contained in the class of descriptions.

The inclusion semantics is a reflexive and transitive deduction system based on the partial order defined below. Intuitively, the description \( d_z \) is included in \( d_x \) (noted \( d_z \subset d_x \)) if \( d_z \) is more constrained than \( d_x \). We note \( d_z \sim d_x \) if \( d_z \subset d_x \) and \( d_x \subset d_z \).

For each description special form, we give its syntax in its section header and provide an informal description of its usage, its static semantics and its inclusion semantics (if any). There is no dynamic semantics for descriptions.

### 2.2.1. id

A variable denotes the description to which it is bound.

There are seven constant identifiers. \( \text{fx}\_\text{unit} \) is the type of expressions used only for their side-effects. \( \text{fx}\_\text{bool} \) is the type of booleans. \( \text{fx}\_\text{pure} \), \( \text{fx}\_\text{read} \), \( \text{fx}\_\text{write} \) and \( \text{fx}\_\text{init} \) are the effects of expressions that are respectively referentially transparent, read-only, write-only and allocation-only. \( \text{fx}\_\text{refof} \) is the type of mutable references to values of type t. They are defined in the \( \text{fx} \) module (see Chapter 3) to limit the number of reserved identifiers.

#### Static Semantics

\[
\begin{align*}
\text{TK}[\text{id} :: k] & \vdash \text{id} :: k \\
\text{TK} \vdash \text{fx}\_\text{unit} :: \text{type} \\
\text{TK} \vdash \text{fx}\_\text{bool} :: \text{type} \\
\text{TK} \vdash \text{fx}\_\text{pure} :: \text{effect} \\
\text{TK} \vdash \text{fx}\_\text{read} :: \text{effect} \\
\text{TK} \vdash \text{fx}\_\text{write} :: \text{effect} \\
\text{TK} \vdash \text{fx}\_\text{init} :: \text{effect} \\
\text{TK} \vdash \text{fx}\_\text{refof} :: (\rightarrow \text{type})
\end{align*}
\]

#### 2.2.2. \( (d_z_0 d_z_1 \ldots d_z_n) \)

A description application is the type obtained by applying the type constructor \( d_z_0 \) to the descriptions \( d_z_i \).

#### Static Semantics

\[
\begin{align*}
\text{TK} \vdash d_z_0 :: (\rightarrow k_1 \ldots k_n) \\
\text{TK} \vdash d_z_i :: k_i \quad (1 \leq i \leq n) \\
\text{TK} \vdash (d_z_0 d_z_1 \ldots d_z_n) :: \text{type}
\end{align*}
\]

### Inclusion Semantics

\[
\begin{align*}
\frac{d_z_i \sim d_x' \quad (0 \leq i \leq n)}{(d_z_0 d_z_1 \ldots d_z_n) \sim (d_x'_0 d_x'_1 \ldots d_x'_n)}
\end{align*}
\]

\[
\begin{align*}
(d_z_0 d_z_1 \ldots d_z_n) & \sim [x_{i=1}^{n} d_{z_i}/d_id_i]d_x
\end{align*}
\]

\[
\begin{align*}
\text{id}_i \notin \text{FV}(d_x) \quad (1 \leq i \leq n)
\end{align*}
\]

\[
\begin{align*}
(d_z_0 d_z_1 \ldots d_z_n) \sim (d_z_0 d_z_1 \ldots d_z_n)
\end{align*}
\]

2.2.3. \( (\rightarrow e_i ((id_1 tz_1) \ldots (id_n tz_n)) tz_{n+1}) \)

The id_i must be distinct.

An \( \rightarrow \) expression is the type of subroutines that map values of type tz_i to a value of type tz_{n+1} while performing the side-effect ei. An \( \rightarrow \) expression is a binding construct.

#### Static Semantics

\[
\begin{align*}
\text{TK} \vdash e_i :: \text{effect} \\
\text{TK} \vdash ((id_1 tz_1) \ldots (id_n tz_n)) tz_{n+1} :: \text{type}
\end{align*}
\]

#### Inclusion Semantics

\[
\begin{align*}
\frac{e_i \subset e_i'} \\
(tz_i' \subset tz_i) \quad (1 \leq i \leq n)
\end{align*}
\]

\[
\begin{align*}
(tz_{n+1} \subset tz_{n+1})
\end{align*}
\]

\[
\begin{align*}
(\rightarrow e_i ((id_1 tz_1) \ldots (id_n tz_n)) tz_{n+1}) & \sim \\
(\rightarrow e_i ((id_1 tz_1') \ldots (id_n tz_n')) tz_{n+1})
\end{align*}
\]

2.2.4. \( (\text{dlambda} ((id_1 k_1) \ldots (id_n k_n)) tz) \)

The id_i must be distinct.

A \text{dlambda} expression is the type constructor that maps descriptions of kinds k_i to the type tz. A \text{dlambda} expression is a binding construct.

#### Static Semantics

\[
\begin{align*}
\text{TK} \vdash (\text{dlambda} ((id_1 k_1) \ldots (id_n k_n)) tz) :: (\rightarrow k_1 \ldots k_n)
\end{align*}
\]
Inclusion Semantics

\[
\text{tz} \sim \text{tz}' \\
\lambda((\text{id}_1, k_1)\ldots) \text{tz} \sim \lambda((\text{id}'_1, k_1)\ldots) \text{tz'}
\]

\[
\text{id}'_i \notin \text{FV(tz)} \ (1 \leq i \leq n) \\
(\lambda((\text{id}_1, k_1)\ldots) \text{tz}) \sim (\lambda((\text{id}'_1, k_1)\ldots) [\text{id}'_i/\text{id}_i] \text{tz})
\]

2.2.5. \text{(maxeff e}_1\ldots e_n\text{)}

A \text{maxeff expression} is the cumulative effect of the effects \text{e}_i.

Static Semantics

\[
\begin{align*}
\text{TK} \vdash e_i &: \text{effect} \ (1 \leq i \leq n) \\
\text{TK} \vdash (\text{maxeff e}_1\ldots e_n) &: \text{effect}
\end{align*}
\]

Inclusion Semantics

\[
(\text{maxeff}) \sim \text{fx..pure}
\]

\[
(\text{maxeff e}_i) \sim e_i
\]

\[
(\text{maxeff e}_1 e_2) \sim (\text{maxeff e}_2 e_1)
\]

\[
(\text{maxeff e}_1 (\text{maxeff e}_2 e_3)) \sim (\text{maxeff (maxeff e}_1 e_2) e_3)
\]

\[
(\text{maxeff e}_1 e_2) \sim e_i
\]

2.2.6. \text{(moduleof (abs id}_a\ k_1\ldots)\ldots (id}_d\ k_m\ldots) \text{tx)}

The \text{id}_a, \text{id}_d and \text{id}_v must be distinct.

A \text{moduleof expression} is the type of modules that export the abstract descriptions \text{id}_a, the transparent descriptions \text{id}_d and the values \text{id}_v. A \text{moduleof expression} is a binding construct.

Static Semantics

\[
\begin{align*}
\text{TK}_{[i=1]} \vdash \text{id}_a &: k_i \vdash \text{tx} \vdash \text{type} \\
\text{TK} \vdash (\text{poly ((id}_1 k_1)\ldots(id}_n k_n) \text{tx}) \vdash \text{type}
\end{align*}
\]

Inclusion Semantics

\[
\begin{align*}
\text{tx} \subset \text{tx}' \\
(\text{poly ((id}_1 k_1)\ldots(id}_n k_n) \text{tx}) \subset (\text{poly ((id}_1 k_1)\ldots(id}_n k'_n) \text{tx}') \\
(\text{poly ((id}_1 k_1)\ldots(id}_n k_n) [\text{id}'_{i=1}\text{id}'_{i}] \text{tx}) \sim (\text{poly ((id}_1 k_1)\ldots(id}_n k'_n) [\text{id}'_{i=1}\text{id}'_{i}] \text{tx}')
\end{align*}
\]

2.2.8. \text{(productof (id}_1 \text{tx}_1)\ldots(id}_n \text{tx}_n\ldots) 

The \text{id}_i must be distinct.

A \text{productof expression} is the type of aggregate values with named fields. Each field \text{id}_i corresponds to a value of type \text{tx}_i.

Static Semantics

\[
\begin{align*}
\text{TK} \vdash \text{tx}_i &: \text{type} \ (1 \leq i \leq n) \\
\text{TK} \vdash (\text{productof (id}_1 \text{tx}_1)\ldots(id}_n \text{tx}_n) \vdash \text{type}
\end{align*}
\]
Inclusion Semantics

| $t_z \subseteq t'_z$ (1 ≤ i ≤ n) |
| $n \geq m$ |
| $(\text{product of } (id_1 t_z) \ldots (id_n t_z))$ |
| $\supseteq$ |
| $(\text{product of } (id_1 t'_z) \ldots (id_n t'_z))$ |

2.2.9. (select $e$ id)

A select expression is the description named id, either abstract or transparent, that is exported by the module $e$. The effect of $e$ must be pure to prevent type abstraction violation.

Static Semantics

\[
\begin{align*}
TK \vdash e & : (\text{module of } (\text{id} \text{ of } \text{tag} \text{ of } \text{desc} \text{ of } \text{val} \text{ of } \text{select} \text{ of } \text{val} \text{ of } \text{select} \text{ of } \text{val} \text{ of } ...) e) \\
TK \vdash (\text{select } e \text{ id}) & :: k_i (1 \leq i \leq n)
\end{align*}
\]

Inclusion Semantics

\[
\begin{align*}
TK \vdash e & : (\text{module of } (\text{id} \text{ of } \text{tag} \text{ of } \text{desc} \text{ of } \text{val} \text{ of } \text{select} \text{ of } \text{val} \text{ of } \text{select} \text{ of } \text{val} \text{ of } ...) e) \\
TK \vdash (\text{select } e \text{ id}) & :: k_i (1 \leq i \leq n)
\end{align*}
\]

2.2.10. (sumof $(id_1 t_z) \ldots (id_n t_z)$)

The $id_i$ must be distinct.

A sumof expression is the type of tagged values of type $t_z$ with tag $id_i$.

Static Semantics

\[
\begin{align*}
TK \vdash t_z & :: \text{type} (1 \leq i \leq n) \\
TK \vdash (\text{sumof } (id_1 t_z) \ldots (id_n t_z)) & :: \text{type}
\end{align*}
\]

2.3. Values

\[
e ::= \text{literal} | \text{sugar} | \text{id} | (e_0 e_1 \ldots e_n) | (begin e_0 \ldots e_n) | (extend e_0 e_1) | (extract t_x e id) | (\text{load literal}) | (\text{module } (\text{define - abstraction } ida_1, \text{kan}) \ldots (\text{idv_1, e}))
\]

For each expression special form (see also the Sugars section), we give its syntax in its section header and provide an informal description of its usage, its static semantics and its dynamic semantics.

The static semantics of expressions is defined modulo the inclusion semantics of descriptions:

\[
\begin{align*}
TK \vdash e & : t_x ! e_i \\
e_i & \sim e'_i \\
t_x & \sim t_x'
\end{align*}
\]

The dynamic semantics is a deduction system based on the transitivity closed → relation defined over pairs made of values $v$ or expressions $e$, and stores $\sigma$. A value is either a literal, or a list of values or expressions in brackets $(v_1 \ldots v_n)$. Stores are functions that map locations to values.

2.3.1. Literals

There are three kernel literals: #t and #f for the fx..bool type and #u for the fx..unit type. Other literals are introduced via the fx module. A literal evaluates to itself and is a pure expression. All fx literals are immutable.

Static Semantics

\[
\begin{align*}
TK \vdash \#t & : \text{fx..bool} ! \text{fx..pure} \\
TK \vdash \#f & : \text{fx..bool} ! \text{fx..pure} \\
TK \vdash \#u & : \text{fx..unit} ! \text{fx..pure}
\end{align*}
\]
Dynamic Semantics

A literal expression evaluates to itself.

2.3.2. \textbf{id}

A variable denotes the value it is bound to.

There are three constant identifiers: the \texttt{fx.ref} subroutine allocates and returns a new reference with initial value \texttt{val}0, the \texttt{fx."} subroutine returns the value stored in \texttt{ref} and the \texttt{fx:=} subroutine replaces the value stored in \texttt{ref} with \texttt{val}1 and returns \#u.

Static Semantics

\[TK[\text{id : tz}] id \rightarrow \text{tz} \triangleq \text{fx..pure}
\]
\[TK \vdash \text{fx.ref} \rightarrow (\text{poly ((t type))})
\]
\[\rightarrow \text{fx..init}
\]
\[((\text{val}0 \ t))
\]
\[((\text{ref (fx..refof t)}))
\]
\[TK \vdash \text{fx."} \rightarrow (\text{poly ((t type))})
\]
\[\rightarrow \text{fx..read}
\]
\[((\text{ref (fx..refof t)}))
\]
\[TK \vdash \text{fx:=} \rightarrow (\text{poly ((t type))})
\]
\[\rightarrow \text{fx..write}
\]
\[((\text{ref (fx..refof t)})
\]
\[\rightarrow (\text{val}1 \ t))
\]
\[\text{fx..unit})
\]

Dynamic Semantics

\[
((\text{fx.ref } v), \sigma) \quad (\text{with } l \text{ unbound in } \sigma)
\]
\[
((\text{loc} l), \sigma[l \rightarrow v])
\]

\[
((\text{fx."} (*\text{loc} l)), \sigma)
\]
\[
(\sigma, \sigma)
\]

\[
((\text{fx:=} (*\text{loc} l) \ v), \sigma)
\]
\[
(\#u, \sigma[l \rightarrow v])
\]

2.3.3. \((e_0 e_1 \ldots e_n)\)

The expressions \(e_i\) are successively evaluated to values \(v_i\) and the value resulting from applying \(v_0\) to \(v_i\) after implicit projection if necessary, see open below) to \(v_i\) is returned.

Static Semantics

\[TK \vdash e_0 \rightarrow (\rightarrow e_i ((\text{id}_1 \text{tz}_1)(\ldots (\text{id}_n \text{tz}_n)) \text{tz}_{n+1}))
\]
\[e_0
\]
\[\text{tz}_i \rightarrow \left[\begin{array}{l}
\text{id}_i/\text{id}_j \text{tz}_i \quad (1 \leq i \leq n + 1)
\end{array}\right)
\]
\[TK \vdash \text{tz} : \text{type} \quad (1 \leq i \leq n + 1)
\]
\[TK \vdash e_i \rightarrow \text{tz} \triangleq e_i \quad (1 \leq i \leq n)
\]

Dynamic Semantics

\[
(e_0 e_1 \ldots e_n) \rightarrow \text{tz} \triangleq e_0
\]
\[\text{maxeff } e_i e_0 e_i \ldots e_i
\]
Dynamic Semantics

An *extend* expression is a special form only available in the dynamic semantics.

\[
(e_0, \sigma) \rightarrow (v_0, \sigma')
\]
\[
((\text{extend}\ e_0,\ e_1),\ \sigma) \rightarrow ((\text{*extend*}\ v_0\ \text{(with}\ v_0\ e_1)),\ \sigma')
\]
\[
(v_1, \sigma) \rightarrow (v_1, \sigma')
\]
\[
((\text{*extend*}\ v_0\ e_1),\ \sigma) \rightarrow ((\text{*extend*}\ v_0\ e_1),\ \sigma')
\]
\[
((\text{extend}\ (\text{with}\ v_0\ e_1)),\ \sigma) \rightarrow ((\text{*extend*}\ v_0\ e_1),\ \sigma')
\]

where \(\{k=1 id' k\} = \{n=1 id k\} - \{n=1 id ' k\}\).

2.3.8. (\lambda C[idi...[id, iY.]) e)

A lambda expression denotes the subroutine that, when applied to \(n\) values \(v_i\), returns the value of \(e\) with the argument values \(v_i\) substituted for the formals \(id_i\) and \(id' i\).

A lambda expression is a binding construct.

Static Semantics

\[
TK F (\text{extract}\ tz\ e\ id),\ \sigma) = \text{extract}\ tz\ e\ id,\ \sigma
\]

Dynamic Semantics

\[
((\lambda (\text{[idi...[id, iY.]}...\text{[id, iY.]})\ e),\ \sigma)
\]

2.3.9. (let ((id, ej)...(id, en)) e)

A let expression simultaneously binds each \(id_i\) to the value \(v_i\) of \(e_i\). The value of \(e\), evaluated in an augmented environment that binds \(id_i\) to \(v_i\), is returned. A let expression is a binding construct.

Static Semantics

\[
TK F - e : tz\ e i : biz,\ \sigma
\]

Dynamic Semantics

\[
((\text{extract}\ tz\ (*\text{product*}\ (id, v i)...(id, v n))\ id i),\ \sigma)
\]

\[
((\text{extract}\ tz\ v\ id),\ \sigma)
\]

\[
((\text{extract}\ tz\ (*\text{product*}\ (id, v i)...(id, v n))\ id i),\ \sigma)
\]

\[
((\text{extract}\ tz\ v\ id),\ \sigma)
\]

2.3.7. (if \(e_0\ e_1\ e_2\))

An if expression evaluates \(e_0\) to the value \(v_0\). If \(v_0\) is \(\#\) (resp. \#f), then the value of \(e_1\) (resp. \(e_2\)) is returned.

Static Semantics

\[
TK F - e_0 : bx,\ \text{bool}!\ e_0
\]
\[
TK F - e_1 : bx!\ e_1
\]
\[
TK F - e_2 : bx!\ e_2
\]
\[
TK F - (\text{if}\ e_0\ e_1\ e_2) : bx\ \text{(max eff}\ e_0\ e_1\ e_2)
\]
Dynamic Semantics

\[
((\text{let } ((id_1 e_1) ... (id_n e_n)) e), \sigma) \\
\rightarrow \\
((\text{lambda} (id_1 ... id_n) e) e_1 ... e_m), \sigma)
\]

2.3.10. (load literal)

The expression in the file named literal is produced as a value. No free variables are allowed in a load file, except if defined in the fx module (see next chapter).

Static Semantics

\[
\phi \vdash (\text{with } \text{fx (include literal)}): tz! e_i \\
TK \vdash (\text{load literal}): tz! e_i
\]

where include is an implementation-specific function that returns the expression in the file whose name is given as an argument.

Dynamic Semantics

\[
((\text{with } \text{fx (include literal)}), \sigma) \rightarrow (v, \sigma') \\
((\text{load literal}), \sigma) \rightarrow (v, \sigma')
\]

2.3.11.

(module (define-abstraction ida1 k1 dx11) ... 
    (define-abstraction ida_n k_n dxna_n) 
    (define-description idd1 dx11) ... 
    (define-description idd_m dxm_m) 
    (define idv1 e1) ...(define idv_p ep) 
    (define-typed idt1 tz1 e'_1) ... 
    (define-typed idt_q tz_q e'_q))

The ida_i, idd_j, idv_k, idt_l must be distinct.

A module expression evaluates to a module that contains the abstract descriptions ida_i, the transparent descriptions idd_j and the values of idv_k and idt_l. The representation descriptions dx_ia of ida_i can be mutually recursive. The values e_k and e'_l are successively evaluated and can be mutually recursive. For each non-effect abstract description ida_i, two subroutines are automatically defined in the scope of the module expression: up-ida_i maps from the representation description to the abstract description, while down-ida_i goes the opposite way.

Static Semantics

\[
TK_1 = TK_{i=1}^{n} \text{idai} :: k_i \\
TK_2 = [\text{poly } ((\text{lambda } (id) \text{idai}) / \text{up-ida_i})] \\
[\text{poly } ((\text{lambda } (id) \text{idd_j/idd_m}) / \text{down-ida_i})]
\]

\[
TK_1 \vdash \text{dxai} :: k_i (1 \leq i \leq n) \\
TK_1 \vdash \text{dx_ia} :: \text{kd_j} (1 \leq j \leq m) \\
TK_1 \vdash [\text{poly } ((\text{lambda } (id) \text{idd_j/idd_m}) / \text{tx_i})] \text{tx_m} :: \text{type} (1 \leq l \leq q) \\
TK_2 \vdash [\text{poly } ((\text{lambda } (id) \text{idd_j/idd_m}) / \text{e_i})] \text{e_q} :: \text{type} (1 \leq l \leq q)
\]

with the following definitions (where id_i are fresh):

\[
Up(d_1, type, d_2) = (-\rightarrow \text{fx}. \text{pure } (d_2) d_1) \\
Up(d_1, (\rightarrow \rightarrow k_1 ... k_m), d_2) = \\
(\text{poly } ((\text{lambda } (id_1 k_1) ... (id_n k_n))) \\
Up((\text{lambda } (id_1) / \text{up-ida_i}) / \text{up-ida_n}) \\
Down(d_i, k, d_2) = Up(d_2, k, d_1)
\]

Dynamic Semantics

The #module-no-rec* and #rec* expressions are special forms only available in the dynamic semantics.

\[
((\text{module}) \\
\text{(define-abstraction ida1 k1 dx11)} ... \\
\text{(define-abstraction ida_n k_n dxna_n)} \\
\text{(define-description idd1 dx11)} ... \\
\text{(define-description idd_m dxm_m)} \\
\text{(define idv1 e1)} ...(define idv_p ep) \\
\text{(define-typed idt1 tz1 e'_1)} ... \\
\text{(define-typed idt_q tz_q e'_q)}, \sigma)
\]

\[
((\text{module}) \\
\text{(define-abstraction ida1 k1 dx11)} ... \\
\text{(define-abstraction ida_n k_n dxna_n)} \\
\text{(define-description idd1 dx11)} ... \\
\text{(define-description idd_m dxm_m)} \\
\text{(define idv1 e1)} ...(define idv_p ep) \\
\text{(define-typed idt1 tz1 e'_1)} ... \\
\text{(define-typed idt_q tz_q e'_q)}, \sigma)
\]

\[
((\text{module-no-rec*}) \\
\text{ida1} [\text{lambda } (id) / \text{up-ida_i}] \\
\text{idv_p} [\text{lambda } (id) / \text{up-ida_i}] \\
\text{idt_l} [\text{lambda } (id) / \text{up-ida_i}] \\
\text{idt_q} [\text{lambda } (id) / \text{up-ida_i}]
\]

The expression in the file named literal is produced as a value. No free variables are allowed in a load file, except if defined in the fx module (see next chapter).

Static Semantics

\[
TK_1 = TK_{i=1}^{n} \text{idai} :: k_i \\
TK_2 = [\text{poly } ((\text{lambda } (id) \text{idai}) / \text{up-ida_i})] \\
[\text{poly } ((\text{lambda } (id) \text{idd_j/idd_m}) / \text{down-ida_i})]
\]

\[
TK_1 \vdash \text{dxai} :: k_i (1 \leq i \leq n) \\
TK_1 \vdash \text{dx_ia} :: \text{kd_j} (1 \leq j \leq m) \\
TK_1 \vdash [\text{poly } ((\text{lambda } (id) \text{idd_j/idd_m}) / \text{tx_i})] \text{tx_m} :: \text{type} (1 \leq l \leq q) \\
TK_2 \vdash [\text{poly } ((\text{lambda } (id) \text{idd_j/idd_m}) / \text{e_i})] \text{e_q} :: \text{type} (1 \leq l \leq q)
\]

with the following definitions (where id_i are fresh):

\[
Up(d_1, type, d_2) = (-\rightarrow \text{fx}. \text{pure } (d_2) d_1) \\
Up(d_1, (\rightarrow \rightarrow k_1 ... k_m), d_2) = \\
(\text{poly } ((\text{lambda } (id_1 k_1) ... (id_n k_n))) \\
Up((\text{lambda } (id_1) / \text{up-ida_i}) / \text{up-ida_n}) \\
Down(d_i, k, d_2) = Up(d_2, k, d_1)
\]
Dynamic Semantics

$$(((\text{plambda } \,(\text{id}_1 \ k_1)\ldots \text{id}_n \ k_n) \ e), \sigma) \rightarrow (e, \sigma)$$

2.3.14. \(\text{(product } tz \ e_1\ldots e_n)\)

The \(n\) expressions \(e_i\) are successively evaluated to values \(v_i\). A product expression evaluates to an aggregate value of product type \(tz\), with each field \(id_i\) having the value \(v_i\).

Static Semantics

$$TK \vdash tz :: \text{type}$$
$$tz \sim \text{(productof } (id_1 \ tz_1)\ldots (id_n \ tz_n))$$
$$TK \vdash e_i : tz_i ! e_i \ (1 \leq i \leq n)$$
$$TK \vdash (\text{product } tz \ e_1 \ldots e_n) : tz \ ! (\text{maxeff } e_1 \ldots e_n)$$

Dynamic Semantics

$$(((\text{product } tz \ e_1 \ldots e_n), \sigma) \rightarrow ((\text{lambda } (id_1 \ 'id_1') \ldots (id_n \ 'id_n')) e_1 \ldots e_n), \sigma)$$

where the \(id_i'\) are fresh.

2.3.15. \(\text{(proj } e \ dz_1 \ldots dz_n)\)

A proj expression projects the polymorphic expression \(e\) onto the description expressions \(dz_i\), returning the corresponding value.

Dynamic Semantics

$$(((\text{open } e), \sigma) \rightarrow (e, \sigma)$$

2.3.12. \(\text{(open } e)\)

An open expression returns, from the polymorphic expression \(e\), the value of \(e\) with the polymorphic description variables \(id_i\) of \(e\) replaced by inferred description expressions \(dz_i\). When a polymorphic value is directly applied to values, open is used to perform implicit projection.

Static Semantics

$$TK \vdash _e : (\text{poly } ((id_1 \ k_1)\ldots (id_n \ k_n)) tz) ! ei$$
$$TK \vdash dz_i :: k_i \ (1 \leq i \leq n)$$
$$TK \vdash (\text{open } e) \ : \ |_i z_i/dz_i/tz \ ! ei$$

Dynamic Semantics

$$TK \vdash (e_0 \ e_1 \ldots e_n) : tz \ ! ei$$

2.3.13. \(\text{(plambda } ((id_1 \ k_1)\ldots (id_n \ k_n)) \ e)\)

The \(id_i\) must be distinct.

A plambda expression denotes the polymorphic value that, when projected onto \(n\) description expressions \(dz_i\) of kind \(k_i\), returns the value of the pure expression \(e\) with the argument values \(dz_i\) substituted for the formals \(id_i\). A plambda expression is a binding construct.

Static Semantics

$$TK \vdash _e : tz ! f x . \text{pure}$$
$$TK \vdash (\text{plambda } ((id_1 \ k_1)\ldots (id_n \ k_n)) \ e) : (\text{poly } ((id_1 \ k_1)\ldots (id_n \ k_n)) tz) ! f x . \text{pure}$$

Dynamic Semantics

$$(((\text{sum } tz \ id \ e), \sigma) \rightarrow ((\text{sum } id \ v), \sigma)$$

2.3.16. \(\text{(sum } tz \ id \ e)\)

The expression \(e\) is evaluated to \(v\) and a tagged value of sum type \(tz\) with tag \(id\) and value \(v\) is returned.

Static Semantics

$$TK \vdash _e : tz :: \text{type}$$
$$tz \sim \text{(sumof } (id \ tz)\ldots)$$
$$TK \vdash e : tz ! ei$$
$$TK \vdash (\text{sum } tz \ id \ e) : tz \ ! ei$$

Dynamic Semantics

$$(((\text{sum } tz \ id \ e), \sigma) \rightarrow ((\text{sum } id \ v), \sigma)$$

2.3.12. \(\text{(open } e)\)

An open expression returns, from the polymorphic expression \(e\), the value of \(e\) with the polymorphic description variables \(id_i\) of \(e\) replaced by inferred description expressions \(dz_i\). When a polymorphic value is directly applied to values, open is used to perform implicit projection.

Static Semantics

$$TK \vdash _e : (\text{poly } ((id_1 \ k_1)\ldots (id_n \ k_n)) tz) ! ei$$
$$TK \vdash dz_i :: k_i \ (1 \leq i \leq n)$$
$$TK \vdash (\text{open } e) \ : \ |_i z_i/dz_i/tz \ ! ei$$

Dynamic Semantics

$$TK \vdash (e_0 \ e_1 \ldots e_n) : tz \ ! ei$$

2.3.13. \(\text{(plambda } ((id_1 \ k_1)\ldots (id_n \ k_n)) \ e)\)

The \(id_i\) must be distinct.

A plambda expression denotes the polymorphic value that, when projected onto \(n\) description expressions \(dz_i\) of kind \(k_i\), returns the value of the pure expression \(e\) with the argument values \(dz_i\) substituted for the formals \(id_i\). A plambda expression is a binding construct.

Static Semantics

$$TK \vdash _e : tz ! f x . \text{pure}$$
$$TK \vdash (\text{plambda } ((id_1 \ k_1)\ldots (id_n \ k_n)) \ e) : (\text{poly } ((id_1 \ k_1)\ldots (id_n \ k_n)) tz) ! f x . \text{pure}$$

Dynamic Semantics

$$(((\text{sum } tz \ id \ e), \sigma) \rightarrow ((\text{sum } id \ v), \sigma)$$
2.3.17. (tagcase tz e id e₁ e₂)

The expressions e, e₁ and e₂ are successively evaluated to values v₁, v₂ and v₂. The value v is a tagged value of type tz with tag id and value v'. If id is id, then the result of applying v₁ to v' is returned, otherwise the result of applying v₂ to v.

Static Semantics

\[
TK \vdash e :: \text{type}
\]
\[
TK \vdash e : tz ! ei
\]
\[
tz \sim (\text{sumof} (id₁, tz₁)...(idₙ, tzₙ))
\]
\[
TK \vdash e₁ : (\rightarrow e₁ ((id₁, tz₁)) tz₁) ! ei₁
\]
\[
TK \vdash e₂ : (\rightarrow e₂ ((id tz₁)) tz₁) ! ei₂
\]
\[
TK \vdash (\text{tagcase} tz e id e₁ e₂) : tz₁ ! \text{(maxeff} ei₁ ei₂ ei₃ ei₄\text{)}
\]

Dynamic Semantics

\[
((\text{tagcase} tz e id e₁ e₂), σ) →
((\lambda (id₁ id₂ id₃) (\text{tagcase} tz id₁ id id₂ id₃)) e₁ e₂), σ)
\]

where idᵢ are fresh.

\[
((\text{tagcase} tz (\text{sum} id v) id v₁ v₂), σ) →
((v₁ v₂), σ)
\]

2.3.18. (the tz e)

The type of e must be included in tz. The value of e is returned.

Static Semantics

\[
TK \vdash tz :: \text{type}
\]
\[
TK \vdash e : tz ! ei
\]
\[
tz \subseteq tz ! ei
\]
\[
TK \vdash (\text{the} tz e) : tz ! ei
\]

Dynamic Semantics

\[
((\text{the} tz e), σ) → (e, σ)
\]

2.3.19. (with eo e₁)

The pure expression eo is evaluated to v₀ and the value of e₁, evaluated in an environment extended with all the bindings defined in the module v₀, is returned. A with expression is a binding construct.

Static Semantics

\[
TK \vdash e₀ : (\text{moduleof}
\]
\[
(\text{abs} (id₁ k₁)...(\text{abs} idₙ kₙ)
\]
\[
(\text{desc} id₁ dz₁)...(\text{desc} idₙ dzₙ)
\]
\[
(\text{val} id₁ tz₁)...(\text{val} idₙ tzₙ))
\]
\[
! \text{fx. pure}
\]
\[
θ = \text{[[p₁=1(with eo idvₙ)/idvₙ]]}
\]
\[
[[p₁=1(select eo id₁)/id₁]]
\]
\[
[[p₁=1(idz₁/id₁)]]
\]
\[
TK \vdash (\text{with} eo idv₀) : θ tz ! \text{fx..pure} (1 ≤ k ≤ p)
\]

Dynamic Semantics

\[
(e₀, σ) → (v₀, σ)
\]
\[
((\text{with} eo e₁), σ) → ((\text{with} v₀ e₁), σ)
\]
\[
((\text{with} (*\text{module*}(idv₁ v₁)...(idvₚ vₚ)) e), σ)
\]
\[
([p₁=1(idv₁ v₁)...(idvₚ vₚ)] e), σ)
\]

2.4. Sugars

sugar ::= (and e₁...eₙ) |
\[
(\text{cond} (e₁ e₁')...(eₙ eₙ') (\text{else} eₙ₊₁)) |
\]
\[
(\text{let*} ((id₁ e₁)...(idₙ eₙ)) e) |
\]
\[
(\text{letrec} ((id₁ e₁)...(idₙ eₙ)) e) |
\]
\[
(\text{match} e (\text{pat} e₁)...(\text{patₙ eₙ})) |
\]
\[
(\text{or} e₁...eₙ) |
\]
\[
(id₁, id₂...idₙ id₁) |
\]
\[
(id₁, id₂) |
\]
\[
[e dz₁...dzₙ] |
\]
\[
(\text{define head} e) |
\]
\[
(\text{define-datatype} ([id (id₁ k₁)...(idₙ kₙ)]) | id)
\]
\[
(id₁ dz₁...dzₙ...)
\]
\[
(idₚ dzp₁...dzₚₙ)
\]
\[
(\text{do} (id e₀ e₁) (e₂ e₃) e) |
\]
\[
(\text{abs} (id₁ id₂...idₙ) k) |
\]
\[
(\text{val} (id₁ id₂...idₙ) tz)
\]

pat ::= literal |
\[
- |
\]
\[
id |
\]
\[
(e \text{pat₁...patₙ})
\]

head ::= id |
\[
(\text{head} (id₁ tz₁)...(idₙ tzₙ)) |
\]
\[
[\text{head} (id₁ k₁)...(idₙ kₙ)]
\]
For each sugar special form, we give its syntax in its section header, provide an informal description of its usage and its rewritten form in terms of kernel constructs.

2.4.1. (and e₁...eₙ)
An and expression performs a short-circuit "and" evaluation of eᵢ to vᵢ, returning #f if one of the vᵢ is #f, #t otherwise.

Rewrite Semantics
- #t (n = 0)
- (if e₁ (and e₂...eₙ) #f)

2.4.2. (cond (e₁ e'₁)...(eₙ e'ₙ) (else e'ₙ₊₁))
A cond expression is a multiple-way test expression. The tests eᵢ are successively evaluated to vᵢ and as soon as one (say j) returns #t (or else is reached), the value of e'ᵢ is returned.

Rewrite Semantics
- e'ₙ₊₁ (n = 0)
- (if e₁ e'₁ (cond e₂ e'₂)...(eₙ e'ₙ) (else e'ₙ₊₁))

2.4.3. (let* ((id₁ e₁)...(idₙ eₙ)) e)
A let* expression successively binds each idᵢ to the value of eᵢ evaluated in an augmented environment that binds idⱼ to vⱼ for j in [1, i - 1]. The value of e, evaluated in an augmented environment that binds idᵢ to vᵢ, is returned.

Rewrite Semantics
- e (n = 0)
- (let ((id₁ e₁)) (let* ((id₂ e₂)...(idₙ eₙ)) e))

2.4.4. (letrec ((id₁ e₁)...(idₙ eₙ)) e)
A letrec expression recursively binds each idᵢ to the value of eᵢ of e. The value of e, evaluated in an augmented environment that binds idᵢ to vᵢ, is returned.

Rewrite Semantics
(let ((id (module (define id₁ e₁)...(define idₙ eₙ))))
 (with id e)) where id is fresh.

2.4.5. (match e (pat₁ e₁)...(patₙ eₙ))
A match expression evaluates e to v and then performs a sequential match of v against the patterns patᵢ. As soon as a match is found with a pattern patᵢ, the value of eᵢ, evaluated in an environment in which the free variables of patᵢ are bound to the appropriate components of v, is returned.

Rewrite Semantics
(let ((id e))
 expand_clause(pat₁...patₙ, e₁...eₙ, id,
 (lambda (x) x),
 (lambda (x) unspecified)))
where id is fresh and the clause expansion function expand_clause(pat₁...patₙ, e₁...eₙ, v, s, f) is defined by:
- (f v), if n = 0
- expand_exp(patₙ, v, (s e₁), e') where e' is expand_clause(pat₂...patₙ, e₂...eₙ, v, s, f), otherwise.

The expression expansion function expand_exp(pat, v, s', f') is defined by:
- (if (= pat v) s' f'), if pat is a literal and = is the equality predicate defined on the type of the literal pat
- s', if pat is _
- (let ((id v)) s'), if pat is id
- (e v (lambda (id₁...idₙ) e') (lambda (z) f')), where the idᵢ and z are fresh and e' is expand_exp(pat₄...patₙ, id₁...idₙ, s', f'), if pat is (e pat₄...patₙ).

The pattern expansion function expand_pat(pat₁...patₙ, id₁...idₙ, s', f') is defined by:
- s', if n = 0
- expand_exp(patₙ, id₁ e', f') where e' is expand_pat(pat₂...patₙ, id₂...idₙ, s', f'), otherwise

2.4.6. (or e₁...eₙ)
An or expression performs a short-circuit "or" evaluation of eᵢ to vᵢ, returning #t if one of the vᵢ is #t, #f otherwise.

Rewrite Semantics
- #f (n = 0)
- (if e₁ #t (or e₂...eₙ))

2.4.7. id₁.id₂....idₙ.id
An infix left-associative "dot" expression returns the value of id in the module that is the value of id₁.id₂....idₙ.

Rewrite Semantics
- (with id₁ id) (n = 1)
- (with id₁ id₂....idₙ id)
2.4.8. \( id_1 \ldots id_2 \)
A "dotdot" expression denotes the description expression bound to \( id_2 \) in the module \( id_1 \).

Rewrite Semantics

\( \text{(select id}_1 \text{id}_2) \)

2.4.9. \([e \; dx_1 \ldots dx_n] \)
A \( \Box \) expression returns the value of \( e \) projected on \( dx_1 \ldots dx_n \).

Rewrite Semantics

\( \text{(proj \; e \; dx}_1 \ldots dx_n) \)

2.4.10. \((\text{define \; head} \; e) \)
A define expression with parenthesized or bracketed head respectively defines a function or a polymorphic value.

Rewrite Semantics

- \((\text{define head}' \; (\text{lambda} \; ((id_1 \; tx_1)\ldots(id_n \; tx_n)) \; e)), \quad \text{if head is} \; (\text{head}' \; (id_1 \; tx_1)\ldots(id_n \; tx_n))\)

- \((\text{define head}' \; (\text{plambda} \; ((id_1 \; k_i)\ldots(id_n \; k_i)) \; e)), \quad \text{if head is} \; [\text{head}' \; (id_1 \; k_i)\ldots(id_n \; k_i)]\)

2.4.11. \((\text{define-datatype} \; [(id \; (idl \; kD)\ldots(idn \; k)) \; | \; id]) \)

\( \text{id}_1 \; dx_{11} \ldots dx_{1m_{1}} \ldots \)

\( \text{id}_p \; dx_{p1} \ldots dx_{pm_{p}} \)

A define-datatype expression defines a possibly higher-order abstract type and a set of functions suited for creating and manipulating (via match) values of that type. A higher-order type definition introduces the following definitions in the current module binding (the case for a simple type is similar, with dlambda and plambda eliminated).

Rewrite Semantics

- \((\text{define-abstraction} \; id) \)

\( \text{dlambda} \; ((id_1 \; k_i)\ldots(id_n \; k_n)) \)

\( \text{(sumof} \; (id'_i \; \text{(productof} \; (L_1 \; dz_{11})\ldots \)

\( (L_{m_1} \; dx_{m_{1}}))) \ldots \)

\( (id'_p \; \text{(productof} \; (L_1 \; dz_{p1})\ldots \)

\( (L_{m_p} \; dx_{m_{p}}))))))))\)

- \((\text{define-description} \; id-rep) \)

\( \text{dlambda} \; ((id_1 \; k_i)\ldots(id_n \; k_n)) \)

\( \text{(sumof} \; (id'_i \; \text{(productof} \; (L_1 \; dz_{11})\ldots \)

\( (L_{m_1} \; dx_{m_{1}}))) \ldots \)

\( (id'_p \; \text{(productof} \; (L_1 \; dz_{p1})\ldots \)

\( (L_{m_p} \; dx_{m_{p}}))))))))\)

2.4.12. \((\text{do} \; (id \; c_0 \; c_i) \; (c_i \; c_i) \; e) \)
A do expression is a loop expression. The expression \( e \) is iteratively evaluated, while the value of \( c_i \) is \#f, in an environment in which \( id \) is initially bound to \( c_0 \) and then to \( c_i \) in all subsequent iterations. Once \( c_i \) evaluates to \#t, the value of \( c_i \) is returned.
3. Standard Descriptions

The fx module defines the standard effects and standard types that are provided by every FX implementation. They fill out the framework introduced by the FX Kernel with a set of useful types and subroutines.

The FX standard effects are given first. The FX standard types and type constructors appear in order of increasing complexity. There is a section for each data type or type constructor, giving its kind, a brief overview of its purpose, the syntax of literals, a list of subroutines with their types, an informal semantics and description of error conditions. In the semantic description of a subroutine, arguments are denoted by the names appearing in the type of the subroutine.

3.1. Pure

The pure effect is the effect of referentially transparent computations. It is already defined in the FX Kernel (cf. previous chapter).

3.2. Init

The init effect is the effect of computations that only initialize freshly allocated memory locations. It is already defined in the FX Kernel (cf. previous chapter).

3.3. Read

The read effect is the effect of computations that only read memory locations. It is already defined in the FX Kernel (cf. previous chapter).

3.4. Write

The write effect is the effect of computations that only write memory locations. It is already defined in the FX Kernel (cf. previous chapter).

3.5. Unit

The unit type denotes the set of values of computations that only perform side-effects. It is already defined in the FX Kernel (cf. previous chapter).

3.6. Bool

The bool type denotes the set of boolean values. It is already defined in the FX Kernel (cf. previous chapter).

There are two boolean literals: \#t (for the true boolean) and \#f (for the false boolean).

equiv? : (-> pure ((p bool) (q bool)) bool)
and? : (-> pure ((p bool) (q bool)) bool)
or? : (-> pure ((p bool) (q bool)) bool)
not? : (-> pure ((p bool)) bool)

Equiv? returns \#t if p and q are both true or both false and \#f otherwise. The subroutines and? and or? respectively return the logical "and" and logical "or" of p and q. Not? returns the negation of p.

3.7. Int

The int type denotes the set of integers.

An integer literal is formed by an optional base prefix, an optional + or - sign (+ is assumed if omitted), and a non-empty succession of digits that are defined in the given base. There are four distinct base prefixes: \#b (binary), \#o (octal), \#d (decimal) and \#x (hexadecimal). If no prefix is supplied, \#d is assumed.
The subroutines \( = \), \( < \), \( > \) and \( \geq \) respectively return \#t if \( x \) is equal to, less than, greater than, less than or equal to and greater than or equal to \( j \) and \#f otherwise. The subroutines \(+\), \(-\) and \(/\) respectively return the sum, product, difference and division of \( x \) and \( y \). \( \text{Flneg} \) returns the opposite of \( x \). \( \text{Flabs} \) returns the absolute value of \( x \). \( \text{Exp} \) returns \( e \) to the power of \( x \). \( \text{Log} \) returns the natural logarithm (in base \( e \)) of \( x \). \( \text{Sqrt} \) returns the square root of \( x \). The subroutines \( \text{sin} \), \( \text{cos} \), \( \text{tan} \), \( \text{asin} \), \( \text{acos} \) and \( \text{atan} \) respectively return the sine, cosine, tangent, arcsine (within \([-\pi/2,\pi/2]\)), arc cosine (within \([-\pi/2,\pi/2]\)) and arctangent (within \([-\pi/2,\pi/2]\)) of \( x \). The subroutines \( \text{floor} \) and \( \text{ceiling} \) respectively return the largest and smallest integer not larger and smaller than \( x \). \( \text{Truncate} \) returns the integer of absolute value not larger than \( \text{flabs} \) \( x \) and of same sign as \( x \). \( \text{Round} \) returns the closest (even if tie) integer to \( x \). \( \text{Int->float} \) returns the real \( x \) such that \((\text{floor} \ x) = (\text{ceiling} \ x) = x \).

A dynamic error is signalled in case of division by zero, overflow or underflow. The subroutines \( \text{log} \) and \( \text{sqrt} \) signal an error if \( x \) is not positive. The precision of floating point values and subroutines is unspecified: truncation may occur if the number of significant digits is too large.

3.9. Char

The char type denotes the set of characters. A character literal is formed by a \#\ prefix followed by a delimiter. The list of allowed identifiers must include: \text{backspace}, \text{newline}, \text{page}, \text{space} and \text{tab}.

\begin{verbatim}
char=? : (-> pure ((c char) (d char)) bool)
char<? : (-> pure ((c char) (d char)) bool)
char>? : (-> pure ((c char) (d char)) bool)
char-=? : (-> pure ((c char) (d char)) bool)
char-ci=? : (-> pure ((c char) (d char)) bool)
char-ci<? : (-> pure ((c char) (d char)) bool)
char-ci>? : (-> pure ((c char) (d char)) bool)
char-ci<=? : (-> pure ((c char) (d char)) bool)
char-ci>=? : (-> pure ((c char) (d char)) bool)
charocious? : (-> pure ((c char) (d char)) bool)
char-alphabetic? : (-> pure ((c char) (d char)) bool)
char-numerical? : (-> pure ((c char) (d char)) bool)
char-whitespace? : (-> pure ((c char) (d char)) bool)
char-lower-case? : (-> pure ((c char) (d char)) bool)
char-upper-case? : (-> pure ((c char) (d char)) bool)
char-upcase : (-> pure ((c char) char))
char-downcase : (-> pure ((c char) char))
char-int : (-> pure ((c char) int))
int->char : (-> pure ((c int) char))
\end{verbatim}

The subroutines \( \text{char=?} \), \( \text{char<?} \), \( \text{char>?} \) and \( \text{char-=?} \) respectively return \#t if \( c \) is equal to, less than, greater than, less than or equal to and greater than or equal to \( d \) and \#f otherwise; these tests are based on a total ordering of characters which is compatible with the ASCII standard on lower-case letters, upper-case letters and digits (without any interleaving between letters and digits). The subroutines \( \text{char-ci=?}, \text{char-ci<?}, \text{char-ci<}? \)
char-ci?>, char-ci<=> and char-ci>=? respectively return #t if c is equal to, less than, greater than, less than or equal to and greater than or equal to d and #f otherwise; these tests are case-insensitive. Char-alphabetic? returns #t when c is alphabetic; a character is alphabetic if its lower-case version is between #\a and #\z. Char-numeric? returns #t when c is a (decimal) digit. Char-whitespace? returns #t when c is a white space. The subroutine char-lower-case? (char-upper-case?) returns #t if c is between *\a (\A) and 9\z (#\Z). The string-ci<=? subroutines char-upcase and char-downcase respectively return the upper-case and lower-case version of c; non-alphabetic characters remain unchanged. Char->int returns the index of c in the character ordering mentioned above. Int->char returns the character with ordering index c.

Int->char signals an error if c is not compatible with the character ordering.

3.10. String

The string type denotes the set of mutable zero-based integer-indexed sequences of characters. Once created, a string is of constant length.

A string literal is formed by a double-quote ("), a sequence of characters (where \ is the escape character for itself and the double-quote character) and an ending double-quote.

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>make-string</td>
<td>(-&gt; init (length int) (c char)) string</td>
</tr>
<tr>
<td>string-length</td>
<td>(-&gt; pure ((s string)) int)</td>
</tr>
<tr>
<td>string-ref</td>
<td>(-&gt; read ((s string) (index int)) char)</td>
</tr>
<tr>
<td>string-set!</td>
<td>(-&gt; write ((s string) (index int) (new-c char)) unit)</td>
</tr>
<tr>
<td>string-fill!</td>
<td>(-&gt; write ((s string) (fill char)) unit)</td>
</tr>
<tr>
<td>string=?</td>
<td>(-&gt; read ((s string) (t string)) bool)</td>
</tr>
<tr>
<td>string&lt;?</td>
<td>(-&gt; read ((s string) (t string)) bool)</td>
</tr>
<tr>
<td>string&gt;?</td>
<td>(-&gt; read ((s string) (t string)) bool)</td>
</tr>
<tr>
<td>string&lt;=?</td>
<td>(-&gt; read ((s string) (t string)) bool)</td>
</tr>
</tbody>
</table>

Make-string allocates and returns a string of length characters c. String-length returns the length of s. String-ref returns the character of s that is at the index position. String-set! replaces in s the character at the index position with new-c and returns #u. String-fill! replaces each character of s with fill and returns #u. The subroutines string=? , string<?, string<=> and string>? respectively return #t if s is lexicographically equal to, less than, greater than, less than or equal to and greater than or equal to t and #f otherwise. The subroutines string-ci=?, string-ci<?, string-ci<=> and string-ci>? respectively return #t if s is lexicographically equal to, less than, greater than, less than or equal to and greater than or equal to t and #f otherwise; these tests are case-insensitive. Substring allocates and returns a string formed from the characters of s between the indices from and to (exclusive); if from and to are equal, then the substring returned is the empty string (""). String-append allocates and returns a string formed by the concatenation of head and tail. String-copy allocates and returns a string with the characters present in s.

It is a dynamic error to try to access out-of-bounds elements of strings. Substring signals a dynamic error if from is not in [0, (string-length s)], if to is not in [0, (string-length s)] and if from is not less than or equal to to.

3.11. Sym

The sym type denotes the set of values that are solely defined by their name.
A symbol literal is formed by a left parenthesis (()), the keyword symbol, a case-insensitive identifier and a right parenthesis ()).

\[
\begin{align*}
\text{sym->string} & : \quad (\text{-> pure ((s sym) (t sym)) bool}) \\
\text{string->sym} & : \quad (\text{-> pure ((s string) (t sym)) bool}) \\
\end{align*}
\]

Sym->string allocates and returns a string corresponding to the name of s. String->sym returns the symbol with name s. Sym=? returns #t if s and t have the same name and #f otherwise.

### 3.12. Permutation type

The permutation type denotes the set of one-to-one mappings on finite intervals of integers starting at 0. Other permutation operations are described with the vector operations (see below).

\[
\begin{align*}
\text{make-permutation} & : \quad (\text{-> pure ((length int) (offset int)) permutation}) \\
\text{cshift} & : \quad (\text{-> pure ((length int) (offset int)) permutation}) \\
\text{identity} & : \quad (\text{-> pure ((length int)) permutation}) \\
\end{align*}
\]

Make-permutation returns the permutation that maps every integer from in the interval \([0, \text{length}]\) to \((\pi \text{ from})\). Cshift returns the permutation that performs a circular shift (i.e. elements shifted out at one end are shifted in at the other end) on the interval \([0, \text{length}]\) by offset positions on the right if offset is positive and by \((-\text{offset})\) positions on the left otherwise. Identity returns a permutation that maps every positive integer less than length to itself.

Make-permutation signals a dynamic error if length is not positive. It is a dynamic error if \(\pi\) does not define a one-to-one mapping. The subroutines cshift and identity signal an error if length is not positive.

### 3.13. Refof type

The type (refof t) denotes the set of mutable references to values of type t. It is already defined in the FX Kernel (cf. previous chapter).

\[
\begin{align*}
\text{ref} & : \quad (\text{poly ((t type)) (-> pure ((val0 t) (refof t)) bool)}) \\
- & : \quad (\text{poly ((t type)) (-> read ((ref (refof t)) t)) bool}) \\
:= & : \quad (\text{poly ((t type)) (-> write ((ref (refof t)) (val1 t)) bool}) \\
\end{align*}
\]

Ref allocates and returns a new reference with initial value val0. ~ returns the value stored in ref. := replaces the value stored in ref with val1 and returns #u.

### 3.14. Uniqueof type

The type (uniqueof t) denotes the multiset of values of type t.

\[
\begin{align*}
\text{unique} & : \quad (\text{poly ((t type)) (-> pure ((u (uniqueof t)) bool)}) \\
\text{value} & : \quad (\text{poly ((t type)) (-> pure ((u (uniqueof t)) bool)}) \\
\text{eq}? & : \quad (\text{poly ((t type)) (-> pure ((u1 (uniqueof t)) (u2 (uniqueof t)) bool)}) \\
\end{align*}
\]

Unique allocates and returns a unique value from x; the init effect ensures that no memoization will be performed on calls to unique. Value returns the embedded value corresponding to u. Eq? returns #t when u1 and u2 have been created by the same call to unique.

### 3.15. Listof type

The type (listof t) denotes the set of mutable homogeneous lists of values of type t.

\[
\begin{align*}
\text{null} & : \quad (\text{poly ((t type)) (-> pure () (listof t)) bool}) \\
\text{null?} & : \quad (\text{poly ((t type)) (-> pure (((list (listof t)) bool)) bool}) \\
\text{cons} & : \quad (\text{poly ((t type)) (-> pure ((car t) (cdr (listof t)) (listof t)) bool}) \\
\text{car} & : \quad (\text{poly ((t type)) (-> pure (((listof t) t)) bool}) \\
\text{cdr} & : \quad (\text{poly ((t type)) (-> pure (((listof t) (listof t)) bool}) \\
\text{set-car!} & : \quad (\text{poly ((t type)) (-> write (((list (listof t)) (new t)) bool}) \\
\text{set-cdr!} & : \quad (\text{poly ((t type)) (-> write (((listof t) (new (listof t)) bool}) \\
\text{length} & : \quad (\text{poly ((t type)) (-> pure ((valof t) (valof (listof t)) bool}) \\
\text{append} & : \quad (\text{poly ((t type)) (-> write (((listof t) (new (listof t)) bool}) \\
\end{align*}
\]
### 3.16. Vectorof

The type `(vectorof t)` denotes the set of mutable, zero-based, integer-indexed, homogeneous vectors that contain elements of type `t`. Once created, a vector is of constant length.

- `make-vector`: `(poly ((t type)) (-> init read) ((length int) (value t)) (vectorof t))`
- `vector-length`: `(poly ((t type)) (-> pure ((vector (vectorof t)))))`
- `vector-ref`: `(poly ((t type)) (-> read ((vector (vectorof t)))))`
- `vector-set!`: `(poly ((t type)) (-> write ((vector (vectorof t))) ((index int)) (seed t) unit))`
- `vector-fill!`: `(poly ((t type)) (-> write ((vector (vectorof t))) ((old (vectorof t)))))`
- `vector->list`: `(poly ((t type)) (-> (maxeff init read) ((vector (vectorof t))) (listof t)))`
- `list->vector`: `(poly ((t type)) (-> (maxeff init read) ((vectorof t))))`
- `vector-map`: `(poly ((t1 type) (t2 type) (e effect)) (-> (maxeff e init read) ((vectorof t2))))`
- `vector-map2`: `(poly ((t1 type) (t2 type) (u type) (e effect)) (-> (maxeff e init read) ((vectorof t2))))`
- `vector-reduce`: `(poly ((t type) (u type) (e effect)) (-> (maxeff e read) ((vectorof t2))))`

### list-tail.

- `null`: returns the empty list. `null?` returns `#t` if the list is empty and `#f` otherwise. Cons allocates and returns a list with car as first element and cdr as remaining elements.
- `car`: returns the first element of list. Cdr returns the list after the first element of list. Set-car! replaces the first element of list with new and returns #u. Set-cdr! replaces the rest of list with new and returns #u. Length returns the number of elements in list. Append allocates and returns a list that is the concatenation of front and rear. Reverse allocates and returns a list with the elements of list in the reverse order. List-tail returns the sublist of list after omitting its first minus elements.
- `list-ref` returns the index-th element of list. Map allocates and returns a list that is obtained by consing the results of applying f on each element x of list from left to right. For-each applies f to each element x of list from left to right and returns #u. Reduce returns the result of the right-associative running (in red) applications of f with each element x of list, beginning with seed; seed is returned if list is empty. List->string allocates and returns a string made of chars. String->list allocates and returns a list made of chars.
3. Standard Descriptions

3.17. Sexp

The sexp type denote the set of values that are usually defined as "symbolic expressions". The type sexp is defined by:

\[
\text{define-datatype sexp type} \\
\text{(unit->sexp unit)} \\
\text{(bool->sexp bool)} \\
\text{(sym->sexp sym)} \\
\text{(int->sexp int)} \\
\text{(float->sexp float)} \\
\text{(char->sexp char)} \\
\text{(string->sexp string)} \\
\text{(list->sexp (listof sexp))} \\
\text{(vector->sexp (vectorof sexp)))}
\]

Sexp? (recursively) compares the two symbolic expressions s1 and s2 for equality; for each basic type, the appropriate equality function is used.

Values of type sexp can be introduced in programs by the "quote" symbol (') in front of a symbolic constant. A symbolic constant is either a literal, a sequence of symbolic constants between parentheses (preceded by a hash sign for vectors). The desugaring of a symbolic constant is defined by induction:

- if the symbolic constant is a literal l of type t (e.g., 1.3), then its desugaring is (t->sexp l) (e.g., (float->sexp 1.3)).
- if the symbolic constant is a sequence between parentheses, then the desugarings of the constituents are gathered in a list l of type (listof sexp) and its
desugaring is (list->sexp l). If the sequence is preceded by a hash sign (#), then a vector v of type (vectorof sexp) is gathered and its desugaring is (vector->sexp v).

3.18. Stream

The type stream denotes the set of values that serve as sequenced source or sink of values of type char. For programming convenience, the fx module contains operations on streams supporting the sexp type.

standard-input : stream
standard-output : stream
open-input-stream : (-> (maxeff init write)
                      (file string)) stream
open-output-stream : (-> (maxeff init write)
                        (file string)) stream
stream-write-sexp : (-> write
                     (output stream) (value sexp)) unit
write-sexp : (-> write ((value sexp)) unit)
write-char : (-> write ((value char)) unit)
stream-read-sexp : (-> write ((input stream) sexp))
read-sexp : (-> write () sexp)
stream-read-char : (-> write ((input stream) char))
read-char : (-> write () char)
stream-char-eof? : (-> write ((input stream) bool))
stream-sexp-eof? : (-> write ((input stream) bool))
close-stream : (-> write (fx stream) unit)
error : (poly ((t type))
       (-> write ((message string)) t))

open-input-stream and open-output-stream signal a dynamic error if the file cannot be opened. It is a dynamic error to perform any operation (except for testing for end of file) on a closed stream. It is a dynamic error to perform a read operation on an input stream if (stream-char-eof? input) is true. It is a dynamic error to perform a stream-sexp-read operation on an input stream if (stream-sexp-eof? input) is true. A dynamic error is signalled on attempts to read from a stream opened for output and on attempts to write to a stream opened for input. It is a dynamic error to apply an -eof? predicate to an output file. A dynamic error is signalled if a malformed s-expression is encountered by read-sexp or stream-read-sexp.

Standard-input and standard-output are implementation-defined streams (usually connected to the user terminal) on which input and output operations can be performed, respectively. Open-input-stream allocates and returns an input stream connected to the file. The interpretation of the string file is implementation-dependent. Stream-read-sexp and stream-read-char return the first value of the input stream. Read-sexp and read-char return the first value of the standard-input-stream. Open-output-stream allocates and returns an output stream connected to the file. Again, the interpretation of the string file is implementation-dependent. Stream-write-sexp and stream-write-char send the value to the output stream and return #u. Write-sexp and write-char send the value to the standard-output stream. Read operations have a write effect because they change the state of the stream. Stream-char-eof? returns #t if no more characters can be read from the input, #f otherwise. Stream-sexp-eof? returns #t if the end of the input will be reached before the start of the next s-expression, #f otherwise. Thus stream-sexp-eof? returns #f if there is only an incomplete s-expression at the end of the stream. Close-stream closes the stream st and returns #u.
REFERENCES


<table>
<thead>
<tr>
<th>Concept/Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute</td>
<td>17</td>
</tr>
<tr>
<td>acos</td>
<td>17</td>
</tr>
<tr>
<td>append</td>
<td>19</td>
</tr>
<tr>
<td>application</td>
<td></td>
</tr>
<tr>
<td>-description</td>
<td>6</td>
</tr>
<tr>
<td>-value</td>
<td>9</td>
</tr>
<tr>
<td>asin</td>
<td>17</td>
</tr>
<tr>
<td>atan</td>
<td>17</td>
</tr>
<tr>
<td>begin</td>
<td>9</td>
</tr>
<tr>
<td>bool</td>
<td>6, 16</td>
</tr>
<tr>
<td>bool-&gt;sexp</td>
<td>21</td>
</tr>
<tr>
<td>car</td>
<td>19</td>
</tr>
<tr>
<td>cdr</td>
<td>19</td>
</tr>
<tr>
<td>ceiling</td>
<td>17</td>
</tr>
<tr>
<td>char</td>
<td>17</td>
</tr>
<tr>
<td>char-&gt;?</td>
<td>17</td>
</tr>
<tr>
<td>char-&gt;&gt;=?</td>
<td>17</td>
</tr>
<tr>
<td>char-&gt;&lt;=?</td>
<td>17</td>
</tr>
<tr>
<td>char-&gt;int</td>
<td>17</td>
</tr>
<tr>
<td>char-&gt;sexp</td>
<td>21</td>
</tr>
<tr>
<td>char-alphabetic?</td>
<td>17</td>
</tr>
<tr>
<td>char-ci&gt;</td>
<td>17</td>
</tr>
<tr>
<td>char-ci-ci&gt;</td>
<td>17</td>
</tr>
<tr>
<td>char-ci-ci&gt;=?</td>
<td>17</td>
</tr>
<tr>
<td>char-ci-ci&lt;=?</td>
<td>17</td>
</tr>
<tr>
<td>char-ci-ci=?</td>
<td>17</td>
</tr>
<tr>
<td>char-downcase</td>
<td>17</td>
</tr>
<tr>
<td>char-lower-case?</td>
<td>17</td>
</tr>
<tr>
<td>char-numeric?</td>
<td>17</td>
</tr>
<tr>
<td>char-upper-case?</td>
<td>17</td>
</tr>
<tr>
<td>char-whitespace?</td>
<td>17</td>
</tr>
<tr>
<td>character</td>
<td>3</td>
</tr>
<tr>
<td>character?</td>
<td>17</td>
</tr>
<tr>
<td>close-stream</td>
<td>22</td>
</tr>
<tr>
<td>comment</td>
<td>4</td>
</tr>
<tr>
<td>compress</td>
<td>21</td>
</tr>
<tr>
<td>cond</td>
<td>14</td>
</tr>
<tr>
<td>cons</td>
<td>19</td>
</tr>
<tr>
<td>conventional</td>
<td>4</td>
</tr>
<tr>
<td>cos</td>
<td>17</td>
</tr>
<tr>
<td>cshift</td>
<td>19</td>
</tr>
<tr>
<td>defabs</td>
<td>11</td>
</tr>
<tr>
<td>defdatatype</td>
<td>15</td>
</tr>
<tr>
<td>defdesc</td>
<td>11</td>
</tr>
<tr>
<td>define</td>
<td>11, 15</td>
</tr>
<tr>
<td>deftyped</td>
<td>11</td>
</tr>
<tr>
<td>delimiter</td>
<td>3</td>
</tr>
<tr>
<td>desc</td>
<td>7</td>
</tr>
<tr>
<td>dfunc</td>
<td>5</td>
</tr>
<tr>
<td>digit</td>
<td>3</td>
</tr>
<tr>
<td>dlambda</td>
<td>6</td>
</tr>
<tr>
<td>do</td>
<td>15</td>
</tr>
<tr>
<td>dot</td>
<td></td>
</tr>
<tr>
<td>dot-dot</td>
<td></td>
</tr>
<tr>
<td>dot-notation</td>
<td>14</td>
</tr>
<tr>
<td>dot-notation</td>
<td>15</td>
</tr>
<tr>
<td>effect</td>
<td>5</td>
</tr>
<tr>
<td>environment,kind</td>
<td>5</td>
</tr>
<tr>
<td>environment,type</td>
<td>5</td>
</tr>
<tr>
<td>eoshift</td>
<td>21</td>
</tr>
<tr>
<td>eq?</td>
<td>19</td>
</tr>
<tr>
<td>equiv?</td>
<td>16</td>
</tr>
<tr>
<td>error</td>
<td>22</td>
</tr>
<tr>
<td>errors</td>
<td></td>
</tr>
<tr>
<td>dynamic</td>
<td>4</td>
</tr>
<tr>
<td>static</td>
<td>4</td>
</tr>
<tr>
<td>exp</td>
<td>17</td>
</tr>
<tr>
<td>expand</td>
<td>21</td>
</tr>
<tr>
<td>expansive</td>
<td>10</td>
</tr>
<tr>
<td>expression</td>
<td></td>
</tr>
<tr>
<td>description</td>
<td>5</td>
</tr>
<tr>
<td>kind</td>
<td>5</td>
</tr>
<tr>
<td>value</td>
<td>5</td>
</tr>
<tr>
<td>extend</td>
<td>9</td>
</tr>
<tr>
<td>extract</td>
<td>10</td>
</tr>
<tr>
<td>f1*</td>
<td>17</td>
</tr>
</tbody>
</table>
Index

<, 17
>, 17
>=, 17
<=, 17
+, 17
-, 17
/, 17
=, 17
abs, 17
neg, 17
not, 16
notations, 4
not-expansive, 10
null, 19
null?, 19
number, 3
open, 12
open-input-stream, 22
open-output-stream, 22
or, 14
or?, 16
permutation, 19
permute 21
plambda, 12
poly, 7
product, 12
productof, 7
proj, 12
projection
implicit, 12
[], 15
pure, 6, 16
read, 6, 16
read-char, 22
read-sexp, 22
reduce, 20
ref, 9, 19
refof, 6, 19
remainder, 17
Reserved identifiers, 4
reverse, 20
round, 17
scan, 21
segmented-scan, 21
select, 8
set-car!, 19
set-cdr!, 19
sexp, 21
sexp=?, 21
sin, 17
sqrt, 17
standard-input, 22
standard-output, 22
store, 8
stream, 22
stream-char-eof?, 22
stream-read-char, 22
stream-read-sexp, 22
stream-sexp-eof?, 22
stream-write-char, 22
stream-write-sexp, 22

identifier, 4
   description, 6
   reserved, 4
   value, 9
identity, 19
if, 10
inferable, 6
init, 6, 16
int, 16
int->char, 17
int->float, 17
int->sexp, 21
lambda, 10
le. a, 19
let, 10
let*, 14
letrec, 14
letter, 3
list->sexp
   list->string, 20
list->vector, 20
list-ref, 20
list-tail, 20
listof, 19
literal, 4, 8
load, 11
log, 17
make-permutation, 19
make-string, 18
make-vector, 20
map, 20
match, 14
maxeff, 7
memoization, 2
module, 11
moduleof, 7
modulo, 17
string, 18
string>, 18
string>?, 18
string>=?, 18
string<=?, 18
string->list, 20
string->sexp, 21
string->sym, 19
string-append, 18
string-ci>, 18
string-ci>?, 18
string-ci>=?, 18
string-ci<?, 18
string-ci<=?, 18
string-ci=?, 18
string-copy, 18
string-fill!, 18
string-length, 18
string-ref, 18
string-set!, 18
string=?, 18
subr, 6
substitution, 4
substring, 18
sum, 12
sumof, 8
sym, 18
sym->sexp, 21
sym->string, 19
sym=? , 19
tagcase, 13
tan, 17
the, 13	
token, 3
truncate, 17
type, 5
unique, 19
uniqueof, 19
unit, 6, 16
unit->sexp, 21
val, 7, 16, 19
value, 8
variable
  bound, 4
  free, 4, 5
vector->list, 20
vector->sexp, 21
vector-fill!, 20
vector-length, 20
vector-map, 20
vector-map2, 20
vector-reduce, 20
vector-ref, 20
vector-set!, 20
vectorof, 20
with, 13
write, 6, 16
write-char, 22
write-sexp, 22