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.

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.
MIT/LCS/TR-531

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 fxQlcs.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.
1. Overview of $FX$

$FX$ 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. $FX$ uses symbolic expression (s-expression) syntax, and thus it is compatible with Lisp source maintenance tools. $FX$ is lexically scoped, statically checked, uses one variable namespace and implements tail-recursion. All values in $FX$ are first-class, including subroutines, polymorphic values and modules.

The $FX$ programming system is based on a kernel language that defines the syntax and semantics of a core set of primitive $FX$ 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 $FX$ kernel forms the core of the $FX$ programming system from the point of view of both the $FX$ application programmer and the $FX$ language implementor.

The foundation provided by the $FX$ kernel is supplemented with a library of standard types and operators that are contained in the $fx$ module. The $fx$ 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 $fx$ 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 $FX$ kernel is divided into a static semantics that is used to deduce the properties of programs before they are run and a dynamic semantics that describes the behavior of programs at execution time.

There are two key theorems that relate the static and dynamic semantics of $FX$. The type soundness theorem guarantees that the static 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 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 $FX$ implementations to improve dynamic performance.

The $FX$ 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 description expressions. There are two principle kinds of descriptions: types, which describe the values expressions compute, and 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 $FX$ static semantics will reconstruct omitted type and effect declarations in a manner that combines the implicit typing of ML[MTH90] with the full power of the explicitly typed second-order polymorphic lambda calculus. The $FX$ 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 $FX$ 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 $FX$, along with an operator to convert between these two forms of polymorphism.

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

1.2. Lexicon

The basic lexical entities used in the $FX$ programming language are the following:

- A digit is one of 0 ... 9.
- A letter is one of a ... z or A ... Z.
- The set of extended alphabetic characters must include: *, /, <, =, >, !, ?, :, $, %, - , ^, _ , [ , ], \, E.
- A white space is a blank space a newline character, a tab character, or a newpage character.
- A character is a digit, a letter, an extended alphabetic character, +, -, a white space or backspace character.
- A delimiter is a white space, a left parenthesis or a right parenthesis.
- A token is a sequence of characters that is separated by delimiters.
- 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).
**4 FX Report**

- 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>abs</th>
<th>and</th>
<th>begin</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>define-abstraction</td>
<td>define</td>
</tr>
<tr>
<td>define-datatype</td>
<td>define-description</td>
<td>define-typed</td>
</tr>
<tr>
<td>desc</td>
<td>dlambda</td>
<td>effect</td>
</tr>
<tr>
<td>else</td>
<td>extend</td>
<td>extract</td>
</tr>
<tr>
<td>fx</td>
<td>if</td>
<td>lambda</td>
</tr>
<tr>
<td>let</td>
<td>letrec</td>
<td>let*</td>
</tr>
<tr>
<td>load</td>
<td>match</td>
<td>maxeff</td>
</tr>
<tr>
<td>module</td>
<td>moduleof</td>
<td>open</td>
</tr>
<tr>
<td>or</td>
<td>plambda</td>
<td>poly</td>
</tr>
<tr>
<td>product</td>
<td>productof</td>
<td>proj</td>
</tr>
<tr>
<td>select</td>
<td>sum</td>
<td>sumof</td>
</tr>
<tr>
<td>symbol</td>
<td>tagcase</td>
<td>the</td>
</tr>
<tr>
<td>type</td>
<td>val</td>
<td>with</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 ε₁...εₙ. If the name of the upper bound on subscripts is not used, we write the shorter: ε₁... If there is at least one expression in the sequence (i.e. n ≥ 1), we use ε₁ε₂...εₙ. We usually denote by ε_i (or any other subscripted ε) 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 {ε ∈ S | ε} the set of ε for each z of S. As a shorthand, {ε ∈ [n]} is noted {ε ∈ [1,n]}. 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 short hand, we note f[ε ∈ [n]|ε] the function equal to f, except at each of the pairwise distinct n arguments z_i where it yields y_i. The result of the application of f to x is noted f[x].

- A variable is free in an expression e if it does not appear in any of the binding constructs within e. A variable that is not free is bound. (Binding constructs are labelled as such in their definition.) All bound variables are alpha-renamed to avoid name clashes with the surrounding context.

- The syntactic substitution s of the variable id by the expression e (noted [e/id]) is the function, defined by induction on the syntactic structure of its domain, that substitutes any free appearance of id in its argument by e; alpha-renaming of bound variables is performed to avoid name clashes. For an ordered index set S, we write [ε ∈ S | ε | e] for the successive substitutions of id by e for each z of S (the variables id must be pairwise distinct). As a shorthand, [ε ∈ [n] | e / id] is noted [ε ∈ [1,n] | e / id].

- 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:

"* 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."

"* FX program text is written in teletype font. Program text is comprised of identifiers, literals, and delimiters."

"* 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 short hand, we note f[ε ∈ [n]|ε] the function equal to f, except at each of the pairwise distinct n arguments z_i where it yields y_i. The result of the application of f to x is noted f[x]."

"* A variable is free in an expression e if it does not appear in any of the binding constructs within e. A variable that is not free is bound. (Binding constructs are labelled as such in their definition.) All bound variables are alpha-renamed to avoid name clashes with the surrounding context."

"* The syntactic substitution s of the variable id by the expression e (noted [e/id]) is the function, defined by induction on the syntactic structure of its domain, that substitutes any free appearance of id in its argument by e; alpha-renaming of bound variables is performed to avoid name clashes. For an ordered index set S, we write [ε ∈ S | ε | e] for the successive substitutions of id by e for each z of S (the variables id must be pairwise distinct). As a shorthand, [ε ∈ [n] | e / id] is noted [ε ∈ [1,n] | e / id]."

"* 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:"
which can be read as “If all the premise; are true, then each conclusion; is true.” The premises and conclusions are implicitly universally quantified over their free variables. If there are no premises, a single box is used.

- Kind, type and effect checking require a type and kind assignment function $TK$ that is the mapping of variables to their type or kind. To distinguish whether $TK$ is extended by a type or kind assignment, we respectively replace the $\rightarrow$ sign by $::$ and $\bowtie$. Kind, effect and type assertions are written in the following way:

$$TK \vdash d :: k$$
$$TK \vdash e : t ! f$$

These assertions mean that “$TK$ proves that $d$ has kind $k$, and $e$ has type $t$ and effect $f$.” The empty assignment is noted $\phi$.

- FV($z$) is the set of free variables and literals of the ordinary or description expression $z$.

2. The FX-91 Kernel

The FX Kernel is a simple programming language that is the basis of the FX programming language. All of the constructs in the FX language can be directly explained by rewriting them into the simpler kernel language. Thus, the kernel forms the core of the FX language from the point of view of both the application programmer and the language implementor.

The FX Kernel has three language levels each with its own set of expressions: value expressions, description expressions and kind expressions. In the simplest terms, programs are value (or ordinary) expressions, types are descriptions, and kinds are the “types of types”.

- Programs are written using value expressions. Value expressions form the lowest level of the language. Literals (e.g. #t) are examples of value expressions. It is possible to write sophisticated programs and only write value expressions.

- Declarations in value expressions are written using description expressions. Descriptions form the second level of the language. There are three kinds of descriptions: effect descriptions, type descriptions and description functions. As the name suggests, descriptions describe value expressions – in particular, every legal value expression has both a type and an effect description. Most omitted declarations are reconstructed by FX.

- 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

$k ::= type | effect | (\rightarrow k_1 ... k_n)$

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 k_1 ... k_n)

A $\rightarrow$ 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

$ti ::= id |$
$(di d_1 ... d_{n_1}) |$
$(\rightarrow ei ((id_1 t_1) ... (id_{n_1} t_{n_1})) t_{n_1+1}) |$
$(productof (id_1 t_1) ... (id_{n_1} t_{n_1})) |$
$(sumof (id_1 t_1) ... (id_{n_1} t_{n_1})) |$
$tx ::= (dx d_x_1 ... d_{x_n}) |$
$(\rightarrow ei ((id_1 t_x_1) ... (id_{n_1} t_{x_{n_1}})) t_{x_{n_1+1}}) |$
$(moduleof (abs id_1 k_1) ... (abs id_{n_1} k_{n_1}))$
$(desc id_1 d_z_1) ... (desc id_{n_1} d_{z_{n_1}})$
$(\text{val id}_1 t_x_1) ... (\text{val id}_{n_1} t_{x_{n_1}})) |$
$(\text{poly} ((id_1 k_1) ... (id_{n_1} k_{n_1})) t_x) |$
$(productof (id_1 t_x_1) ... (id_{n_1} t_{x_{n_1}})) |$
$(sumof (id_1 t_x_1) ... (id_{n_1} t_{x_{n_1}})) |$
$ti$,
$ei ::= id | (\maxeff ei_1 ... ei_{n_1})$
$dx ::= tx | (\dlambda ((id_1 k_1) ... (id_{n_1} k_{n_1})) t_x) | di$
$di ::= ti | ei | id | (\select e id)$
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 $\sqsubseteq$ partial order defined below. Intuitively, the description $dz_1$ is included in $dz_2$ (noted $dz_1 \sqsubseteq dz_2$) iff $dz_1$ is more constrained than $dz_2$. We note $dz_1 \sim dz_2$ if $dz_1 \sqsubseteq dz_2$ and $dz_2 \sqsubseteq dz_1$.

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

#### Static Semantics

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

### 2.2.2. $(dz_0 dz_1...dz_n)$

A description application is the type obtained by applying the type constructor $dz_0$ to the descriptions $dz_i$.

#### Static Semantics

\[
\begin{align*}
TK & \vdash dz_0 ::(\rightarrow k_1...k_n) \\
TK & \vdash dz_i :: k_i \quad (1 \leq i \leq n) \\
TK & \vdash (dz_0 dz_1...dz_n) :: type
\end{align*}
\]

### Inclusion Semantics

\[
\begin{align*}
dz_i & \sim dz'_i \quad (0 \leq i \leq n) \\
(dz_0 dz_1...dz_n) & \sim (dz'_0 dz'_1...dz'_n)
\end{align*}
\]

\[
((dlambda ((id_1 k_1)...(id_n k_n)) tx) dz_1...dz_n) \sim \\
[1\leq i \leq n] dz_i/id_i]tx
\]

\[
id_i \notin FV(tx) \quad (1 \leq i \leq n) \\
(dlambda (((id_1 k_1)...(id_n k_n)) (tx id_1...id_n)) \sim tx
\]

### 2.2.3. $(\rightarrow ei ((id_1 tz_1)...(id_n tz_n)) tz_{n+1})$

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

\[
\begin{align*}
TK[i=1...id_j:: tz_j] & \vdash tz_{j+1} :: type \quad (0 \leq i \leq n) \\
TK & \vdash (\rightarrow ei ((id_1 tz_1)...(id_n tz_n)) tz_{n+1}) :: type
\end{align*}
\]

### Inclusion Semantics

\[
\begin{align*}
et_1 & \sqsubseteq et_2 \\
tz_i & \sqsubseteq tz_i \quad (1 \leq i \leq n) \\
tz_{n+1} & \sqsubseteq tz'_{n+1} \\
(\rightarrow ei ((id_1 tz_1)...(id_n tz_n)) tz_{n+1}) & \sqsubseteq \\
(\rightarrow ei ((id_1 tz'_1)...(id_n tz'_{n+1})) tz'_{n+1})
\end{align*}
\]

\[
\begin{id_i \notin \bigcup_{i=1}^{n+1} FV(tz_i) \quad (1 \leq i \leq n)} \\
(\rightarrow ei ((id_1 tz_1)...(id_n tz_n)) tz_{n+1}) & \sim \\
(\rightarrow ei ((id_1 tz'_1)...(id_n tz'_{n+1}) \sim [id_1/id_j]tz_{n+1})
\]

### 2.2.4. $(dlambda ((id_1 k_1)...(id_n k_n)) tx)$

The $id_i$ must be distinct.

A $dlambda$ expression is the type constructor that maps descriptions of kinds $k_i$ to the type $tx$. A $dlambda$ expression is a binding construct.

#### Static Semantics

\[
\begin{align*}
TK[i=1...id_i:: k_i] & \vdash tx :: type \\
TK & \vdash (dlambda ((id_1 k_1)...(id_n k_n)) tx) :: (\rightarrow k_1...k_n)
\end{align*}
\]
### Inclusion Semantics

<table>
<thead>
<tr>
<th>$\mathcal{F} \sim \mathcal{F}'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\lambda ((i_1 k_1)...) \mathcal{F}) \sim \lambda ((i'_1 k'_1)...) \mathcal{F}'$</td>
</tr>
</tbody>
</table>

### 2.2.5. \text{maxeff} $e_1 \ldots e_n$

A \text{maxeff} expression is the cumulative effect of the effects $e_i$.

#### Static Semantics

\[
\begin{align*}
\text{TK} & \vdash e_i :: \text{effect} \quad (1 \leq i \leq n) \\
\text{TK} & \vdash \text{maxeff} \; e_1 \ldots e_n :: \text{effect}
\end{align*}
\]

### Inclusion Semantics

<table>
<thead>
<tr>
<th>$(\text{maxeff}) \sim \mathcal{F} \ldots \text{pure}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\text{maxeff} ; e) \sim e$</td>
</tr>
</tbody>
</table>

### 2.2.6. \text{moduleof} $(\text{abs} \; i_1 \; k_1)\ldots(\text{abs} \; i_n \; k_n)$

The $i_1$, $i_2$, and $i_3$ must be distinct.

A \text{moduleof} expression is the type of modules that export the abstract descriptions $i_1$, the transparent descriptions $i_2$, and the values $i_3$. A \text{moduleof} expression is a binding construct.

#### Static Semantics

\[
\begin{align*}
\text{TK} & \vdash i_1 :: k_1 \quad (1 \leq k \leq p) \\
\text{TK} & \vdash i_n :: (\text{abs} \; i_1 \; k_1)\ldots(\text{abs} \; i_n \; k_n) \quad (1 \leq j \leq m)
\end{align*}
\]

### Inclusion Semantics

<table>
<thead>
<tr>
<th>$\mathcal{F} \subseteq \mathcal{F}'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\text{poly} ; ((i_1 k_1)\ldots(i_n k_n)) ; \mathcal{F}) \subseteq \mathcal{F}'$</td>
</tr>
</tbody>
</table>

### 2.2.8. \text{productof} $(i_1 \; t_1)\ldots(i_n \; t_n)$

The $i_1$ must be distinct.

A \text{productof} expression is the type of aggregate values with named fields. Each field $i_1$ corresponds to a value of type $t_1$.

#### Static Semantics

\[
\text{TK} \vdash t_i :: \text{type} \quad (1 \leq i \leq n)
\]
Inclusion Semantics

<table>
<thead>
<tr>
<th>$t_j$</th>
<th>$t'_i$</th>
<th>$(1 \leq i \leq m)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n \geq m$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(\text{product of } (id_1 t_{z_1}) \ldots (id_n t_{z_n}))$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\subset$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(\text{product of } (id_1 t'_{z'<em>1}) \ldots (id_m t'</em>{z'_m}))$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

$$TK \vdash e : (\text{module of } (\text{define-abstraction } id_1 \ldots z_{n+1}) \ldots (\text{define-abstraction } id_m z_{n+1}))$$

$$TK \vdash (\text{select } e \text{ id}) :: \text{val } i \ (1 \leq i \leq n)$$

Inclusion Semantics

$$TK \vdash e : (\text{module of } (\text{define-abstraction } id_1 \ldots z_{n+1}) \ldots (\text{define-abstraction } id_m z_{n+1}))$$

$$TK \vdash (\text{select } e \text{ id}) :: \text{val } i \ (1 \leq i \leq m)$$

2.2.10. (sumof (id$_i$ $t_{z_i}$) $\ldots$ (id$_n$ $t_{z_n}$))

The id$_i$ must be distinct.

A sumof expression is the type of tagged values of type $t_{z_i}$ with tag id$_i$.

Static Semantics

$$TK \vdash (\text{sumof } (id_1 \ldots t_{z_i}) \ldots (id_n \ldots t_{z_n})) :: \text{type}$$

Inclusion Semantics

$$t_{z_i} \subset t'_{z'_i} \ (1 \leq i \leq n)$$

$$n \leq n'$$

$$\subset$$

$$\left(\text{sum of } (id_1 \ldots t_{z_i}) \ldots (id_n \ldots t_{z_n})\right)$$

$$\pi$$ is a permutation on $[1, n]$

$$\left(\text{sum of } (id_1 \ldots t_{z_1}) \ldots (id_n \ldots t_{z_n})\right)$$

$$\sim$$

$$\left(\text{sum of } (id_{x(1)} \ldots t_{z_{x(1)}}) \ldots (id_{x(n)} \ldots t_{z_{x(n)}})\right)$$

2.3. Values

$$e ::= \text{literal} \mid \text{sugar} \mid \text{id} \mid \text{begin} \ldots \text{end} \mid \text{extend} \ldots \text{call} \mid \text{extract} \ldots \text{id} \mid \text{lambda} \ldots \text{let} \ldots \text{e} \mid \text{load literal} \mid \text{module} \ldots \text{define-abstraction} \ldots \text{define-description} \ldots \text{lambda} \ldots \text{load} \ldots \text{define-typed} \ldots \text{define-typed}$$

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:

$$TK \vdash e : t \vdash e$$

The dynamic semantics is a deduction system based on the transitively closed $\rightarrow$ relation defined over pairs made of values $v$ or expressions $e$, and stores $s$. 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 $\text{fx..bool}$ type and #u for the $\text{fx..unit}$ type. Other literals are introduced via the $\text{fx}$ module. A literal evaluates to itself and is a pure expression. All $\text{fx}$ literals are immutable.

Static Semantics

$$TK \vdash \#t : \text{fx..bool} ! \text{fx..pure}$$

$$TK \vdash \#f : \text{fx..bool} ! \text{fx..pure}$$

$$TK \vdash \#u : \text{fx..unit} ! \text{fx..pure}$$
Dynamic Semantics

A literal expression evaluates to itself.

2.3.2. id

A variable denotes the value it is bound to.

There are three constant identifiers: the `fx.ref` subroutine allocates and returns a new reference with initial value `val0`, the `fx.` subroutine returns the value stored in `ref` and the `fx.:=` subroutine replaces the value stored in `ref` with `val1` and returns `valu`.

Static Semantics

\[
TK[: \text{id}] \vdash \text{id} : \text{tx} ! \text{fx..pure} \\
TK[: \text{fx.ref}] \vdash (\text{poly } ((\text{t type})) \\
(\rightarrow \text{fx..init} \text{,} ((\text{val0 t})) \\
(\text{fx..refof t}))) \\
TK[: \text{fx.}^-] \vdash (\text{poly } ((\text{t type})) \\
(\rightarrow \text{fx..read} \text{,} ((\text{ref (fx..refof t)}))) \\
t) \\
TK[: \text{fx.}:=] \vdash (\text{poly } ((\text{t type})) \\
(\rightarrow \text{fx..write} \text{,} ((\text{ref (fx..refof t)})) \text{,} \text{(val1 t))} \\
\text{fx..unit}))
\]

Dynamic Semantics

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

\[
((\text{fx.}^- (\text{loct id})), \sigma) \\
(\sigma, \sigma)
\]

\[
((\text{fx.}:= (\text{loct id})), \sigma) \\
(\text{#u, } \sigma[l \rightarrow v])
\]

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

The expressions \(e_i\) are successively evaluated to values \(v_i\) and the value resulting from applying \(v_0\) is returned. The expression \(e_1\) has access to the bindings of \(e_0\). In the case of conflict, the bindings of \(v_i\) take precedence. An extend expression is a binding construct.

Static Semantics

\[
TK[: e_0] \vdash (\text{moduleof} \\
(\text{abs ida}_{01} \ldots k_{01}) \ldots (\text{abs ida}_{0n} \ldots k_{0n}) \\
(\text{desc idd}_{01} \ldots dz_{01}) \ldots (\text{desc idd}_{0m} \ldots dz_{0m}) \\
(\text{val idv}_{01} \ldots tz_{01}) \ldots (\text{val idv}_{0p} \ldots tz_{0p})) \\
! e_0 \\
TK[: (with e_0 e_1)] \vdash tx ! ei \\
tx \sim (\text{moduleof} \\
(\text{abs ida}_{11} \ldots k_{11}) \ldots (\text{abs ida}_{1n} \ldots k_{1n}) \\
(\text{desc idd}_{11} \ldots dz_{11}) \ldots (\text{desc idd}_{1m} \ldots dz_{1m}) \\
(\text{val idv}_{11} \ldots tz_{11}) \ldots (\text{val idv}_{1p} \ldots tz_{1p})) \\
\{n_{11} \ldots ida_{11}\} = \{n_{12} \ldots ida_{12}\} \ldots (\text{desc idd}_{11} \ldots dz_{11}) \\
\{m_{11} \ldots idd_{11}\} = \{m_{12} \ldots idd_{12}\} \ldots (\text{desc idd}_{1m} \ldots dz_{1m}) \\
\{p_{11} \ldots idv_{11}\} = \{p_{12} \ldots idv_{12}\} \ldots (\text{val idv}_{11} \ldots tz_{11}) \\
TK[: (extend e_0 e_1)] \\
: (\text{moduleof} \\
(\text{abs ida}_{21} \ldots k_{21}) \ldots (\text{abs ida}_{2n} \ldots k_{2n}) \\
(\text{abs ida}_{11} \ldots k_{11}) \ldots (\text{abs ida}_{1n} \ldots k_{1n}) \\
(\text{desc idd}_{21} \ldots dz_{21}) \ldots (\text{desc idd}_{2m} \ldots dz_{2m}) \\
(\text{desc idd}_{11} \ldots dz_{11}) \ldots (\text{desc idd}_{2m} \ldots dz_{2m}) \\
(\text{val idv}_{21} \ldots tz_{21}) \ldots (\text{val idv}_{2p} \ldots tz_{2p}) \\
(\text{val idv}_{11} \ldots tz_{11}) \ldots (\text{val idv}_{1p} \ldots tz_{1p})) \\
! (\text{maxeff } e_0, e_1)
\]
Dynamic Semantics

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

\[
\begin{align*}
(e_0, \sigma) &\rightarrow (v_0, \sigma') \\
((\text{extend} e_0 \ e_1), \sigma) &\rightarrow ((\text{*extend*} v_0 \ (\text{with} \ v_0 \ e_1)), \sigma')
\end{align*}
\]

\[
\begin{align*}
(e_1, \sigma) &\rightarrow (v_1, \sigma') \\
((\text{*extend*} v_0 \ e_1), \sigma) &\rightarrow ((\text{*extend*} v_0 \ v_1), \sigma')
\end{align*}
\]

\[
\begin{align*}
((\text{*extend*} \ (\text{*module*} \ id_1 \ v_1) \ldots \ (id_n \ v_m)), \sigma) &\rightarrow \\
((\text{*module*} \ id'_1 \ v_1) \ldots \ (id'_m \ v_m)), \sigma)
\end{align*}
\]

\[
\begin{align*}
((\text{let} \ ((id_1 \ e_1) \ldots \ (id_n \ e_n)) \ e), \sigma) &\rightarrow \\
((\text{*lambda*} \ (id_1 \ id'_1) \ldots \ (id_n \ id'_n)) \ e), \sigma)
\end{align*}
\]

\[
\begin{align*}
\text{if } e_0 \ e_1 \ e_2 & \rightarrow \text{if } v_0 \ e_1 \ e_2 \\
((\text{if } e_0 \ e_1 \ e_2), \sigma) &\rightarrow ((\text{if } v_0 \ e_1 \ e_2), \sigma')
\end{align*}
\]

\[
\begin{align*}
((\text{if } \# e_1 \ e_2), \sigma) &\rightarrow (e_1, \sigma) \\
((\text{if } \# f e_1 \ e_2), \sigma) &\rightarrow (e_2, \sigma)
\end{align*}
\]

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

2.3.6. (extract tz e id)

The expression \(e\) of product type \(tz\) is evaluated to an aggregate value \(v\). The value of the field \(id\) of \(v\) is returned.

\[
\begin{align*}
TK &\vdash (\text{extract} \ tz \ e \ id) \ : \ tz_i \ ! \ ei \\
TK &\vdash e : \ tz \ ! \ ei \\
\text{extract} &\sim (\text{productof} \ (id_1 \ tz_1) \ldots \ (id_n \ tz_n))
\end{align*}
\]

2.3.7. (if e0 e1 e2)

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

\[
\begin{align*}
TK &\vdash e_0 : \text{fx} \ldots \text{bool} \ ! \ ei_0 \\
TK &\vdash e_1 : \text{tx} \ ! \ ei_1 \\
TK &\vdash e_2 : \text{tx} \ ! \ ei_2
\end{align*}
\]

\[
\begin{align*}
TK &\vdash (\text{if } e_0 \ e_1 \ e_2) : \text{tx} \ ! \ (\text{maxeff} e_0 \ e_1 \ e_2)
\end{align*}
\]

\[
\begin{align*}
TK &\vdash e_i : \text{tx}_i \ ! \ ei_i \ (1 \leq i \leq n) \\
G &\{i|\text{not}_\text{expansive}(e_i) \ (1 \leq i \leq n)\} \\
TK[i \in G \land \text{ID}_i] &\vdash e : \text{tx} \ ! \ ei
\end{align*}
\]

\[
\begin{align*}
TK &\vdash (\text{let} \ ((id_1 \ e_1) \ldots \ (id_n \ e_n)) \ e) \\
&\vdash \{0_{i=1} e_i/id\} \tx : \text{type}
\end{align*}
\]

where an expression is \text{not}_\text{expansive} iff it is a literal, an identifier, a lambda expression, a plambda expression or a non-application compound expression for which each value subexpression is \text{not}_\text{expansive}. 
2. The FX-91 Kernel

Dynamic Semantics

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

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

\[\varphi \vdash (\text{with } \text{fx} (\text{include } \text{literal})) : tx ! ei\]

\[\text{TK} \vdash (\text{load } \text{literal}) : tx ! ei\]

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

Dynamic Semantics

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

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

2.3.11. (module (define-abstraction ida_1 k_1 dza_1) ... (define-abstraction ida_n k_n dza_n) (define-description idd_1 dzd_1) ... (define-description idd_m dzd_m) (define idv_1 e_1) ...(define idv_p e_p) (define-typed idt_1 tz_1 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 dza_i 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

\[\text{TK}_1 = \text{TK}[(\text{idv}_1 \ldots \text{idv_p}) \rightarrow \text{idv}_1] \in (\text{fn}_1 \ldots \text{fn}_p)\]

\[\text{TK}_2 = \text{TK}[(\text{ida}_1 \ldots \text{ida}_n) \rightarrow \text{ida}_1] \in (\text{fn}_1 \ldots \text{fn}_n)\]

\[\text{TK}_3 = \text{TK}[(\text{idd}_1 \ldots \text{idd}_m) \rightarrow \text{idd}_1] \in (\text{fn}_1 \ldots \text{fn}_m)\]

\[\text{TK}_4 = \text{TK}[(\text{idt}_1 \ldots \text{idt}_q) \rightarrow \text{idt}_1] \in (\text{fn}_1 \ldots \text{fn}_q)\]

with the following definitions (where id are fresh):

\[\text{Up}(d_1, \text{type}, d_2) = (\rightarrow \text{fx}. \text{pure} (d_1) d_2)\]

\[\text{Up}(d_1, \rightarrow \rightarrow k_1 \ldots k_n, d_2) = (\text{poly} ((id_1 \ldots id_n) k_1) \ldots (id_1 \ldots id_n k_n))\]

\[\text{Down}(d_1, k, d_2) = \text{Up}(d_2, d_1, k)\]

Dynamic Semantics

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

```
((module
  (define-abstraction ida_1 k_1 dza_1)...
  (define-abstraction ida_n k_n dza_n)
  (define-description idd_1 dzd_1)...
  (define-description idd_m dzd_m)
  (define idv_1 e_1) ...(define idv_p e_p)
  (define-typed idt_1 tz_1 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 dza_i 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.
```
Dynamic Semantics

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

2.3.14. (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

\[
\begin{align*}
TK & \vdash tz :: \text{type} \\
& \vdash (\text{productof} \ (id \, tz) \ldots (id_n \, tz_n)) \\
TK & \vdash e_i : tz_i ! e_i \quad (1 \leq i \leq n) \\
TK & \vdash (\text{product} \ (e_1 \ldots e_n)) : tz \\
& \quad ! (\text{maxeff} \ e_1 \ldots e_n)
\end{align*}
\]

Dynamic Semantics

\[ ((\text{product} \ (tz \, e_1 \ldots e_n)), \sigma) \rightarrow ((\text{lambda} \ (id'_1 \ldots id'_n) \ (\text{productof} \ (id'_1 \, id'_1) \ldots (id'_n \, id'_n))) \ e_1 \ldots e_n), \sigma) \]

where the \( id'_i \) are fresh.

2.3.15. (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.

Static Semantics

\[
\begin{align*}
TK & \vdash e : (\text{poly} \ ((id_1 \, k_1) \ldots (id_n \, k_n)) \, tz) ! ei \\
& \vdash dz_i :: k_i \quad (1 \leq i \leq n) \\
& \vdash (\text{proj} \ e \, dz_1 \ldots dz_n) : tz_i ! ei
\end{align*}
\]

Dynamic Semantics

\[ ((\text{proj} \ e \, dz_1 \ldots dz_n), \sigma) \rightarrow ((\text{sum} \ (tz \, v)), a) \]

2.3.16. (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

\[
\begin{align*}
TK & \vdash tz :: \text{type} \\
& \vdash (\text{sumof} \ (id \, tz) \ldots) \\
TK & \vdash e : tz ! ei \\
& \vdash (\text{sum} \ (tz \, id \, e)) : tz ! ei
\end{align*}
\]

Dynamic Semantics

\[ ((\text{sum} \ (tz \, id \, e)), \sigma) \rightarrow ((\text{*sum} \ id \, v), \sigma') \]
2.3.17. \(\text{tagcase } tz \ e \ mid \ e_1 \ e_2\)

The expressions \(e\), \(e_1\) and \(e_2\) are successively evaluated to values \(v_1\), \(v_1\) and \(v_2\). The value \(v\) is a tagged value of type \(tz\) with tag \(id\) and value \(v'\). If \(id = id_j\), then the result of applying \(v_1\) to \(v'\) is returned, otherwise the result of applying \(v_2\) to \(v\).

\[\text{Static Semantics}\]

\[
\begin{align*}
TK & \vdash tz :: \text{type} \\
TK & \vdash e : tz ! ei \\
\text{tz} & \sim (\text{sumof } (id_1, tz_1) \ldots (id_n, tz_n)) \\
TK & \vdash e_1 : (\Rightarrow e_0 ((id, tz_1)) \text{ tz}) ! ei_1 \\
TK & \vdash e_2 : (\Rightarrow ei_4 (((id, tz_1)) \text{ tz}) ! ei_2 \\
TK & \vdash (\text{tagcase } tz \ e \ id_1, e_2) : \\
& \quad tz, \quad (\text{maxeff } ei_1 \ e_1, e_2, e_i_4)
\end{align*}
\]

\[\text{Dynamic Semantics}\]

\[
\begin{align*}
((\text{tagcase } tz \ e \ id_1, e_2), \sigma) & \rightarrow \\
((\text{lambda } (id_1, id_2, id_3) \\
\quad (\text{tagcase } tz id_1, id_2, id_3)) \ e \ e_1, e_2), \sigma)
\end{align*}
\]

where \(id_i\) are fresh.

\[
\begin{align*}
((\text{tagcase } tz (*\text{sum*} id \ v) \ id v_1, v_2), \sigma) & \rightarrow \\
((v_1 \ v), \sigma)
\end{align*}
\]

\[
\begin{align*}
((\text{tagcase } tz (*\text{sum*} id' \ v) \ id v_1, v_2), \sigma) & \rightarrow \\
((v_2 \ (*\text{sum*} id' \ v)), \sigma)
\end{align*}
\]

2.3.18. \((\text{the } tz \ e)\)

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

\[\text{Static Semantics}\]

\[
\begin{align*}
TK & \vdash tz :: \text{type} \\
TK & \vdash e : tz ! ei \\
tz & \subset tz \\
TK & \vdash (\text{the } tz \ e) : tz ! ei
\end{align*}
\]

\[\text{Dynamic Semantics}\]

\[
((\text{the } tz \ e), \sigma) \rightarrow (e, \sigma)
\]

2.3.19. \((\text{with } e_0 \ e_1)\)

The pure expression \(e_0\) is evaluated to \(v_0\) and the value of \(e_1\), evaluated in an environment extended with all the bindings defined in the module \(v_0\), is returned. A with expression is a binding construct.

\[\text{Static Semantics}\]

\[
\begin{align*}
TK & \vdash e_0 : (\text{moduleof} \\
\quad \text{(abs } id_1, k_1) \ldots (\text{abs } id_n, k_n) \\
\quad (\text{desc } id_1, dz_1) \ldots (\text{desc } id_m, dz_m) \\
\quad (\text{val } id_1, tz_1) \ldots (\text{val } id_p, tz_p)) \\
\quad ! fx. \quad \text{pure} \\
\vdash e_0 : \sigma \\
\theta & = (\theta_{e_0}(\text{with } e_0, idv_k)/idv_k) \\
\theta_{e_0}(\text{select } e_0, ida_i)/ida_i \\
\theta_{idv_k}(dx/ida_j)
\end{align*}
\]

\[\text{Dynamic Semantics}\]

\[
\begin{align*}
TK & \vdash (\text{with } e_0, e_1) : \theta \ 	ext{tz}! \text{fx} ! \text{ei}
\end{align*}
\]

2.4. Sugars

\(\text{sugar ::= (and } e_1 \ldots e_n) | \\
\quad (\text{cond } e_1 \ldots e_n (\text{else } e_{n+1}) | \\
\quad (\text{let* } (((id_1, e_1) \ldots (id_n, e_n)) \ e) | \\
\quad (\text{letrec } (((id_1, e_1) \ldots (id_n, e_n)) \ e) | \\
\quad (\text{match } e (\text{pat }_1, e_1) \ldots (\text{pat }_n, e_n)) | \\
\quad (\text{or } e_1 \ldots e_n) | \\
\quad id_1, id_2 \ldots id_n, id | \\
\quad id_1, id_2 | \\
\quad [c \ dz_1 \ldots dx_i] | \\
\quad (\text{define } head c) | \\
\quad (\text{define-datatype } [id (id_1, k_1) \ldots (id_n, k_n)] \mid id) \\
\quad (id_1, dz_1 \ldots dx_1) \\
\quad (id_p, dz_1 \ldots dx_{m_p}) \\
\quad (\text{do } (id e_0, e_1) (e_1, e_2) e) | \\
\quad (\text{abs } (id_1, id_2 \ldots id_n) \ k) | \\
\quad (\text{val } (id_1, id_2 \ldots id_n) \ tz)
\)

\(\text{pat ::= literal | \\
\quad - | \\
\quad id | \\
\quad (e \ pat_1 \ldots pat_n)
\)

\(\text{head ::= id | \\
\quad (head (id_1, tz_1) \ldots (id_n, tz_n)) | \\
\quad [\text{head } (id_1, k_1) \ldots (id_n, k_n)]\)
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) #t)

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'ₜ 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 vᵢ 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 vᵢ of eᵢ. 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(patᵢ, 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 (x) f')) where the idᵢ and z are fresh and e' is expand_exp(pat₁...patₙ, id₁...idₙ, s', f'), if pat is (e pat₁...patₙ).

The pattern expansion function expand_exp(pat₁...patₙ, id₁...idₙ, s', f') is defined by:
- s', if n = 0
- expand_exp(patₙ, id₁ e', f') where e' is expand_exp(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 \( [] \) 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. (define head \( e \))

A define expression with parenthesized or bracketed head respectively defines a function or a polymorphic value.

Rewrite Semantics

- (define head' (lambda (\( (id, kl) \ldots (id, k_n) \)) e)), if head is \( \text{head'} (id, k_1) \ldots (id, k_n) \)
- (define head' (plambda (\( (id, k_1) \ldots (id, k_n) \)) e)), if head is \( \text{head'} (id, k_1) \ldots (id, k_n) \)

2.4.11. (define-datatype \( [[id (id_1 k_1) \ldots (id_n k_n)] \mid id] \)

\[
(id'_1 \; dx_{11} \ldots dx_{1 m_1}) \ldots
(id'_p \; dx_{p1} \ldots dx_{p m_p})
\]

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

- (define-abstraction id

\[
(\Rightarrow k_1 \ldots k_n)
\]

(dlambda ((id, k_1) \ldots (id_n k_n))

\[
(\text{sumof } (id'_1 \text{ productof } (L_1 \; dx_{11}) \ldots
\; (L_{m_1} \; dx_{1 m_1})) \ldots
(id'_p \text{ productof } (L_1 \; dx_{p1}) \ldots
\; (L_{m_p} \; dx_{p m_p}))
\]

- (define-description id-rep

\[
\text{dlambda } ((id, k_1) \ldots (id_n k_n))
\]

\[
(\text{sumof } (id'_1 \text{ productof } (L_1 \; dx_{11}) \ldots
\; (L_{m_1} \; dx_{1 m_1})) \ldots
(id'_p \text{ productof } (L_1 \; dx_{p1}) \ldots
\; (L_{m_p} \; dx_{p m_p}))
\]

- (define-typed id'

\[
(p \text{ poly } ((id, k_1) \ldots (id_n k_n))
\]

\[
(\Rightarrow \text{fx..pure }
\; ((id', dx_{11}) \ldots (id'_m, dx_{1 m}))
\; (id_1 \ldots id_n))
\]

(plambda ((id, k_1) \ldots (id_n k_n))

\[
(\text{lambda } ((id', dx_{11}) \ldots (id'_m, dx_{1 m}))
\; (\text{up-id } (\text{sumof } (\text{id-rep } id_1 \ldots id_n))
\; id')
\]

\[
(\text{product } (\text{productof }
\; (L_1 \; dx_{11}) \ldots (L_{m_1} \; dx_{1 m}))
\; id'_1 \ldots id'_m))
\]

- (define-typed id'~

\[
(p \text{ poly } ((id, k_1) \ldots (id_n k_n))
\]

\[
(x_1 \text{ effect }) (x_2 \text{ effect }) (t \text{ type})
\]

\[
(\Rightarrow \text{fx..pure }
\; ((v \; (id \; id_1 \ldots id_n))
\; (s \Rightarrow x_1
\; ((id', dx_{11}) \ldots (id'_m, dx_{1 m}))
\; t))
\]

\[
(f \Rightarrow x_2 ((v \; (id \; id_1 \ldots id_n)))
\; t))
\]

\[
(\text{plambda } ((id, k_1) \ldots (id, k_n))
\; (x_1 \text{ effect }) (x_2 \text{ effect }) (t \text{ type})
\]

\[
(\text{lambda }
\; ((v \; (id \; id_1 \ldots id_n))
\; (s \Rightarrow x_1
\; ((id', dx_{11}) \ldots (id'_m, dx_{1 m}))
\; t))
\]

\[
(f \Rightarrow x_2 ((v \; (id \; id_1 \ldots id_n)))
\; t))
\]

\[
(\text{tagcase } (\text{id-rep } id_1 \ldots id_n)
\; (id-down v)
\; id'
\]

\[
(\text{lambda } (v_1)
\; (s \; (\text{extract } (\text{productof }
\; (L_1 \; dx_{11}) \ldots
\; (L_{m_1} \; dx_{1 m}))
\; v_1
\; L_1))
\]

\[
(\text{extract } (\text{productof }
\; (L_1 \; dx_{11}) \ldots
\; (L_{m_1} \; dx_{1 m}))
\; v_1
\; L_1)
\]

\[
(\text{lambda } (z)
\; (f \; v))
\]

where \( L_i, id', x_i, t, v, v_1, s, f \) and \( z \) are fresh.

2.4.12. (do (id e_0 e_1) (e_1 e_2) e)

A do expression is a loop expression. The expression \( e \) is iteratively evaluated, while the value of \( e \) is #t, in an environment in which \( id \) is initially bound to \( e_0 \) and then to \( e_1 \) in all subsequent iterations. Once \( e \) evaluates to #t, the value of \( e_2 \) is returned.
Rewrite Semantics

(letrec ((id' (lambda (id)
  (if e1
      (begin e
          (id' e0))))))

where id' is fresh.

2.4.13. (abs (id1 id2...idn) k)

An abs form with a list of identifiers denotes a sequence of abs forms for each idi.

Rewrite Semantics

- (abs id1 k) (n = 1)
- (abs id1 k) (abs (id2...idn) k)

2.4.14. (val (id1 id2...idn) tz)

A val form with a list of identifiers denotes a sequence of val forms for each idi.

Rewrite Semantics

- (val id1 tz) (n = 1)
- (val id1 tz) (val (id2...idn) tz)

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).

There is one value of type unit: the literal #u.

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)) 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.
3.8. Float

The float type denotes the set of floating point numbers. A float literal is formed by an optional + or - sign (+ is assumed if omitted), a non-empty succession of decimal digits, a decimal point, a non-empty succession of decimal digits and an optional exponent denoted by the letter E or e, an optional + or - sign (+ is assumed if omitted) and a sequence of decimal digits.

The subroutines =, <, >, <= and => respectively return #t if x is equal to, less than, greater than, less than or equal to and greater than or equal to y and #f otherwise. The subroutines +, * and - respectively return the sum, product and difference of i and j. The subroutines round and truncate respectively return the integer to x. The subroutines modulo and absolute return the integer to x. The subroutines floor and ceiling respectively return the largest and smallest integer not larger than (flabs x) and of same sign as i). Absolute returns the absolute value of i.

A dynamic error is signalled in case of division by zero, overflow or underflow. The subroutines log and sqrt signal an error if x is not positive. The precision of float literals 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 character or an identifier followed by a delimiter. The list of allowed identifiers must include: backspace, newline, page, space and tab.

The subroutines char=?, char<?, char=?, char<=? and 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 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 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).
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 *\z (*\Z). The 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.

make-string : (-> init ((length int) (c char)) string)
string-length : (-> pure ((s string) int) string)
string-ref : (-> read ((s string) (index int)) char)
string-set! : (-> write ((s string) (index int) (new-c char)) unit)
string-fill! : (-> write ((s string) (fill char)) unit)
string=? : (-> read ((s string) (t string)) bool)
string<? : (-> read ((s string) (t string)) bool)
string>? : (-> read ((s string) (t string)) bool)
string<=? : (-> read ((s string) (t string)) bool)

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?>, 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>? 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 ()).

\[
\begin{align*}
sym\rightarrow\text{string} & : \quad (\rightarrow \text{read} ((s\ \text{string}))\ \text{sym}) \\
\text{string}\rightarrow\text{sym} & : \quad (\rightarrow \text{pure} ((s\ \text{sym})\ (t\ \text{sym}))\ \text{bool}) \\
\text{sym}=? & : \quad (\rightarrow \text{pure} ((s\ \text{sym})\ (t\ \text{sym}))\ \text{bool})
\end{align*}
\]

Sym\rightarrow\text{string} allocates and returns a string corresponding to the name of s. String\rightarrow\text{sym} returns the symbol with name s. Sym=? returns #t if s and t have the same name and #f otherwise.

### 3.12. Permutation

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).

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

\text{make-permutation} returns the permutation that maps every integer from in the interval [0,length] to (\pi from). \text{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,length] by offset positions on the right if offset is positive and by (neg offset) positions on the left otherwise. \text{identity} returns a permutation that maps every positive integer less than length to itself.

\text{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 \text{cshift} and \text{identity} signal an error if length is not positive.

### 3.13. Refof

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).

\[
\begin{align*}
\text{ref} & : \quad (\rightarrow \text{init} ((\text{val0}\ t)\ (\text{refof}\ t))) \\
\text{-} & : \quad (\rightarrow \text{read} ((\text{refof}\ t)\ t)) \\
:= & : \quad (\rightarrow \text{write} ((\text{refof}\ t)\ (\text{val1}\ t))\ \text{unit})
\end{align*}
\]

\text{ref} allocates and returns a new reference with initial value val0. \text{-} returns the value stored in ref. := replaces the value stored in ref with val1 and returns #u.

### 3.14. Uniqueof

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

\[
\begin{align*}
\text{unique} & : \quad (\rightarrow \text{pure} ((t\ \text{type})\ (\text{uniqueof}\ t))) \\
\text{value} & : \quad (\rightarrow \text{pure} ((t\ \text{type}))\ (\text{uniqueof}\ t))) \\
\text{eq}? & : \quad (\rightarrow \text{pure} ((t\ \text{type}))\ (\text{uniqueof}\ t)))
\end{align*}
\]

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

### 3.15. Listof

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

\[
\begin{align*}
\text{null} & : \quad (\rightarrow \text{pure} ((t\ \text{type})\ (\text{listof}\ t))) \\
\text{null?} & : \quad (\rightarrow \text{pure} ((t\ \text{type}))\ (\text{listof}\ t))) \\
\text{cons} & : \quad (\rightarrow \text{read} ((\text{listof}\ t)\ (\text{listof}\ t)))\ (\text{listof}\ t))) \\
\text{car} & : \quad (\rightarrow \text{read} ((\text{listof}\ t)\ (\text{listof}\ t)))\ (\text{listof}\ t))) \\
\text{cdr} & : \quad (\rightarrow \text{read} ((\text{listof}\ t)\ (\text{listof}\ t)))\ (\text{listof}\ t)))
\end{align*}
\]

\text{set-car!} : \quad (\rightarrow \text{write} ((\text{listof}\ t)\ (\text{listof}\ t)))\ (\text{listof}\ t))) \\
\text{set-cdr!} : \quad (\rightarrow \text{write} ((\text{listof}\ t)\ (\text{listof}\ t)))\ (\text{listof}\ t))) \\
\text{length} : \quad (\rightarrow \text{read} ((\text{listof}\ t)\ (\text{listof}\ t)))\ (\text{listof}\ t))) \\
\text{append} : \quad (\rightarrow \text{read} ((\text{listof}\ t)\ (\text{listof}\ t)))\ (\text{listof}\ t)))
reverse : (poly ((t type))
  (-> (maxeff init read)
      (list (listof t)))
      (listof t)))

list-tail : (poly ((t type))
  (-> read
      ((list (listof t)) (minus int))
      (listof t)))

list-ref : (poly ((t type))
  (-> read
      ((listof (listof t)) (index int))
      (listof t)))

map : (poly (((t1 type) (t2 type) (e effect))
  (-> (maxeff e init read)
      ((list (listof t1)) (t2))
      (listof t2))))

for-each : (poly (((t1 type) (t2 type) (e effect))
  (-> (maxeff e init read)
      ((listof (listof t1)) (t2))
      (listof t2))))

reduce : (poly (((t1 type) (t2 type) (e effect))
  (-> (maxeff e init read)
      ((listof t1)) (t2))
      (listof t2))))

list->string : (-> (maxeff read init)
  ((chars (listof char)))
  string)

string->list : (-> (maxeff read init)
  ((chars string))
  (listof char))

list-tail.

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
      ((length int) (value t))
      (vectorof t)))

vector-length : (poly ((t type))
  (-> pure
      ((vectorof (vectorof t)))
      (index int))
      (vectorof t))

vector-ref : (poly ((t type))
  (-> read
      ((vectorof t)))
      (index int))

vector-set! : (poly ((t type))
  (-> write
      ((vectorof t))
      (index int))
      (vectorof t))

vector-fill! : (poly ((t type))
  (-> write
      ((vectorof t)))
      (new t))

vector->list : (poly ((t type))
  (-> (maxeff init read)
      ((vectorof t)))
      (listof t)))

list->vector : (poly ((t type))
  (-> (maxeff init read)
      ((list (listof t))
      (vectorof t)))

vector-map : (poly (((t1 type) (t2 type) (e effect))
  (-> (maxeff e init read)
      ((listof t1)) (t2))
      (vectorof t2)))

vector-map2 : (poly (((t1 type) (t2 type) (e effect))
  (-> (maxeff e init read)
      ((vectorof t1))
      (vectorof t2)))

vector-reduce : (poly (((t type) (u type) (e effect))
  (-> (maxeff e read)
      ((vectorof t))
      (vectorof u)))
      (seed u))

It is a dynamic error to apply access operations such as car or cdr on the empty list. A dynamic error is signalled if set-car! or set-cdr! is applied to the empty list. A dynamic error is signalled if the index is out of range in list-ref or if minus is greater than the length of list in list-tail.
vector obtained by replicating default, except for entries gathered in a list. Compress allocates and returns a vector obtained by selecting from vector the elements that have a corresponding #t value in selection. Expand allocates and returns a vector obtained by replicating default, except for entries in selection that are #t in which case the next available element of vector is chosen. Eoshift allocates and returns a vector obtained by performing an “End-Off” shift (i.e. element are shifted out at one end and default values are shifted in at the other end) of vector by offset positions on the right if offset is positive and by (neg offset) positions on the left otherwise.

The subroutines vector-ref and vector-set! signal a dynamic error if index is not in [0,(vector-length vector)]. It is a dynamic error for f not to be associative in vector-reduce, scan and segmented-scan. Segmented-scan signals a dynamic error if the lengths of segments and vector differ. Permute signals a dynamic error if the length of input differs from the domain of the mapping. Compress signals a dynamic error if the lengths of selection and vector differ. Expand signals a dynamic error if the length of selection and default differ.

3.17. Sexp
type

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

```
(define-datatype sexp
  (unit->sexp unit)
  (bool->sexp bool)
  (sym->sexp sym)
  (int->sexp int)
  (float->sexp float)
  (char->sexp char)
  (string->sexp string)
  (list->sexp (listof sexp))
  (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 \((\text{list->sexp } 1)\). If the sequence is preceded by a hash sign \((#)\), then a vector \(v\) of type \((\text{vectorof sexp})\) is gathered and its desugaring is \((\text{vector->sexp } v)\).

### 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 \(\text{fx}\) module contains operations on streams supporting the sexp type.

```lisp
standard-input  :  stream
standard-output :  stream
open-input-stream :  \((-\rightarrow\ (\text{maxeff init write})\)
                     \((\text{file string})\) \text{ stream})
open-output-stream :  \((-\rightarrow(\text{maxeff init write})\)
                     \((\text{file string})\) \text{ stream})
stream-write-sexp :  \((-\rightarrow\ \text{write}\)
                    \((\text{output stream} (\text{value sexp}))\) \text{ unit})
write-sexp      :  \((-\rightarrow\ \text{write}\)
                    \((\text{value sexp})\) \text{ unit})
write-stream-sexp :  \((-\rightarrow\ \text{write}\)
                    \((\text{value sexp})\) \text{ unit})
write-char       :  \((-\rightarrow\ \text{write}\)
                    \((\text{value char})\) \text{ unit})
stream-read-sexp :  \((-\rightarrow\ \text{write}\)
                    \((\text{input stream})\) \text{ sexp})
read-sexp        :  \((-\rightarrow\ \text{write}\)
                    \((\text{input stream})\) \text{ sexp})
stream-read-char :  \((-\rightarrow\ \text{write}\)
                    \((\text{input stream})\) \text{ char})
read-char        :  \((-\rightarrow\ \text{write}\)
                    \((\text{input stream})\) \text{ char})
stream-char-eof? :  \((-\rightarrow\ \text{write}\)
                    \((\text{input stream})\) \text{ bool})
stream-sexp-eof? :  \((-\rightarrow\ \text{write}\)
                    \((\text{input stream})\) \text{ bool})
close-stream :  \((-\rightarrow\ \text{write}\)
                \((\text{exit stream})\) \text{ unit})
error :  \((\text{poly ((t type))})\)
          \((-\rightarrow\ \text{write}\)
           \((\text{message string})\)
           \(t\))
```

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 \#t. 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 \#n. Both stream-sexp-eof? and stream-char-eof? return \#t when applied to closed streams. Error prints its message on standard-output and signals a dynamic error.

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 (apart from 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 \(-\text{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.
REFERENCES


ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

-, 9, 19
*, 17
<, 17
>, 17
>=, 17
<=, 17
+, 17
-, 17
/, 17
:=, 9, 19
=, 17
abs, 7, 16
absolute, 17
acos, 17
and, 14
and?, 16
append, 19
application
description, 6
value, 9
asin, 17
atan, 17
begin, 9
bool, 6, 16
bool->sexp, 21
car, 19
cdr, 19
ceiling, 17
char, 17
char>, 17
char=?, 17
char>=?, 17
char<=?, 17
char->int, 17
char->sexp, 21
char-alphabetic?, 17
char-ci>, 17
char-ci>=?, 17
char-ci>=?, 17
char-ci<=?, 17
char-ci=?, 17
char-downcase, 17
char-lower-case?, 17
char-numeric?, 17
char-upper-case?, 17
char-whitespace?, 17
char=??, 17
character, 3
defined, 3
white space, 3
close-stream, 22
comment, 4
compress, 21
cond, 14
cons, 19
conventional, 4
cos, 17
cshift, 19
defabs, 11
defdatatype, 15
defdesc, 11
define, 11, 15
deftyped, 11
delimiter, 3
desc, 7
dfunc, 5
digit, 3
dlambda, 6
do, 15
dot
notation, 14
dot-dot
notation, 15
effect, 5
environment, 5, 16
evironment, 5, 15
*errors

dynamic, 4
static, 4
exp, 17
expand, 21
expansive, 10
expression
description, 5
kind, 5
value, 5
extend, 9
extract, 10
f1+, 17
Index

fl<, 17
fl>, 17
fl>=, 17
fl=<, 17
fl+, 17
fl-, 17
fl/., 17
fl=, 17
flabs, 17
flneg, 17
float, 17
float->sexp, 21
floor, 17
for-each, 20
FV, 5

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

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
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-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