LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-531

2

REPORT ON THE FX-91 PROGRAMMING LANGUAGE

David K. Gifford

Pierre Jouvelot

Mark A. Sheldon

James W. O'Toole

February 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

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

This research was supported by the Defense Advanced Research Projects Agency of the Departement 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 Departement 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

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 *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: `*`, `/`, `<`, `=`, `>`, `!`, `?`, `:`, `$`, `%`, `_`, `&`, `^`, `~`, `[`, `]`, `@`.
- 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).

- 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.

`abs`	`and`	`begin`
`cond`	`define-abstraction`	`define`
`define-datatype`	`define-description`	`define-typed`
`desc`	`dlambda`	`effect`
`else`	`extend`	`extract`
`fx`	`if`	`lambda`
`let`	`letrec`	`let*`
`load`	`match`	`maxeff`
`module`	`moduleof`	`open`
`or`	`plambda`	`poly`
`product`	`productof`	`proj`
`select`	`sum`	`sumof`
`symbol`	`tagcase`	`the`
`type`	`val`	`with`

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 "x must be y" indicates that "it is a static error if x 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 $e_1 \ldots e_n$ or $\ldots_{i=1}^{n} e_i$. If the name of the upper bound on subscripts is not used, we write the shorter: $e_1 \ldots$. If there is at least one expression in the sequence (*i.e.* $n \geq 1$), we use $e_1 e_2 \ldots e_n$. We usually denote by e_i (or any other subscripted e) an expression belonging to such sequence. Certain parameters can have different forms; $[x|y]$ stands for either x or y.

- The set of values x that satisfy the predicate P is noted $\{x \mid P(x)\}$; 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 $\{_{x \in S} e_x\}$ the set of e_x for each x of S. As a shorthand, $\{_{i \in [1,n]} e_i\}$ is noted $\{_{i=1}^{n} e_i\}$. 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 $f[x \mapsto y]$. As a short hand, we note $f[_{i=1}^{n} x_i \mapsto y_i]$ the function equal to f, except at each of the pairwise distinct n arguments x_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 $[_{x \in S} e_x / id_x]$ for the successive substitutions of id_x by e_x for each x of S (the variables id_x must be pairwise distinct). As a shorthand, $[_{i \in [1,n]} e_i / id_i]$ is noted $[_{i=1}^{n} e_i / id_i]$.

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

- A deduction system is a set of rules written in the following way:

$$\boxed{\dfrac{premise_1 ... premise_n}{conclusion_1 ... conclusion_m}}$$

which can be read as "If all the $premise_i$ are true, then each $conclusion_j$ 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 $\mapsto$ sign by : and :: . 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 ϕ.

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

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` | `(->>` 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. `(->>` $k_1 ... k_n$`)`

A `->>` 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 di1...din) |
       (-> ei ((id1 ti1)...(idn tin)) tin+1) |
       (productof (id1 ti1)...(idn tin)) |
       (sumof (id1 ti1)...(idn tin)) |
tx ::= (dx dx1...dxn) |
       (-> ei ((id1 tx1)...(idn txn)) txn+1) |
       (moduleof (abs ida1 k1)...(abs idan kn)
                 (desc idd1 dx1)...(desc iddp dxp)
                 (val idv1 tx1)...(val idvm txm)) |
       (poly ((id1 k1)...(idn kn)) tx) |
       (productof (id1 tx1)...(idn txn)) |
       (sumof (id1 tx1)...(idn txn)) |
       ti
ei ::= id | (maxeff ei1...ein)
dx ::= tx | (dlambda ((id1 k1)...(idn kn)) tx) | 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 x) 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 $\sqsubset$ partial order defined below. Intuitively, the description dx_1 is included in dx_2 (noted $dx_1 \sqsubset dx_2$) iff dx_1 is more constrained than dx_2. We note $dx_1 \sim dx_2$ if $dx_1 \sqsubset dx_2$ and $dx_2 \sqsubset dx_1$.

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` t) 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

$$
\begin{array}{rcl}
TK[id :: k] & \vdash & id :: k \\
TK & \vdash & \texttt{fx..unit} :: \texttt{type} \\
TK & \vdash & \texttt{fx..bool} :: \texttt{type} \\
TK & \vdash & \texttt{fx..pure} :: \texttt{effect} \\
TK & \vdash & \texttt{fx..read} :: \texttt{effect} \\
TK & \vdash & \texttt{fx..write} :: \texttt{effect} \\
TK & \vdash & \texttt{fx..init} :: \texttt{effect} \\
TK & \vdash & \texttt{fx..refof} :: (\texttt{->>}\ \texttt{type})
\end{array}
$$

2.2.2. $(dx_0\ dx_1 \ldots dx_n)$

A description application is the type obtained by applying the type constructor dx_0 to the descriptions dx_i.

Static Semantics

$$
\dfrac{TK \vdash dx_0 :: (\texttt{->>}\ k_1 \ldots k_n) \qquad TK \vdash dx_i :: k_i \quad (1 \le i \le n)}{TK \vdash (dx_0\ dx_1 \ldots dx_n) :: \texttt{type}}
$$

Inclusion Semantics

$$
\dfrac{dx_i \sim dx'_i \quad (0 \le i \le n)}{(dx_0\ dx_1 \ldots dx_n) \sim (dx'_0\ dx'_1 \ldots dx'_n)}
$$

$$
\boxed{\begin{array}{c}((\texttt{dlambda}\ ((id_1\ k_1) \ldots (id_n\ k_n))\ tx)\ dx_1 \ldots dx_n) \\ \sim \\ [^n_{i=1} dx_i / id_i] tx\end{array}}
$$

$$
\dfrac{id_i \notin \mathrm{FV}(tx) \quad (1 \le i \le n)}{(\texttt{dlambda}\ ((id_1\ k_1) \ldots (id_n\ k_n))\ (tx\ id_1 \ldots id_n)) \sim tx}
$$

2.2.3. $(\texttt{->}\ ei\ ((id_1\ tx_1) \ldots (id_n\ tx_n))\ tx_{n+1})$

The id_i must be distinct.

An `->` expression is the type of subroutines that map values of type tx_i to a value of type tx_{n+1} while performing the side-effect ei. An `->` expression is a binding construct.

Static Semantics

$$
\dfrac{\begin{array}{rcl} TK & \vdash & ei :: \texttt{effect} \\ TK[^i_{j=1} id_j : tx_j] & \vdash & tx_{i+1} :: \texttt{type} \quad (0 \le i \le n)\end{array}}{TK \vdash (\texttt{->}\ ei\ ((id_1\ tx_1) \ldots (id_n\ tx_n))\ tx_{n+1}) :: \texttt{type}}
$$

Inclusion Semantics

$$
\dfrac{\begin{array}{rcl} ei & \sqsubset & ei' \\ tx'_i & \sqsubset & tx_i \quad (1 \le i \le n) \\ tx_{n+1} & \sqsubset & tx'_{n+1}\end{array}}{\begin{array}{c}(\texttt{->}\ ei\ ((id_1\ tx_1) \ldots (id_n\ tx_n))\ tx_{n+1}) \\ \sqsubset \\ (\texttt{->}\ ei'\ ((id_1\ tx'_1) \ldots (id_n\ tx'_n))\ tx'_{n+1})\end{array}}
$$

$$
\dfrac{id'_i \notin \bigcup_{j=1}^{n+1} \mathrm{FV}(tx_j) \quad (1 \le i \le n)}{\begin{array}{c}(\texttt{->}\ ei\ ((id_1\ tx_1) \ldots (id_n\ tx_n))\ tx_{n+1}) \\ \sim \\ (\texttt{->}\ ei\ ((id'_1\ tx_1) \ldots (id'_n\ [^{n-1}_{j=1} id'_j / id_j] tx_n)) \\ [^n_{j=1} id'_j / id_j] tx_{n+1})\end{array}}
$$

2.2.4. $(\texttt{dlambda}\ ((id_1\ k_1) \ldots (id_n\ k_n))\ tx)$

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

$$
\dfrac{TK[^n_{i=1} id_i :: k_i] \vdash tx :: \texttt{type}}{\begin{array}{l}TK \vdash \ (\texttt{dlambda}\ ((id_1\ k_1) \ldots (id_n\ k_n))\ tx) \\ \qquad\qquad :: (\texttt{->>}\ k_1 \ldots k_n)\end{array}}
$$

Inclusion Semantics

$$\frac{tx \sim tx'}{\begin{array}{c}\texttt{(dlambda ((}id_1\ k_1\texttt{)...)}\ tx\texttt{)}\\ \sim \\ \texttt{(dlambda ((}id_1\ k_1\texttt{)...)}\ tx'\texttt{)}\end{array}}$$

$$\frac{id'_i \notin \mathrm{FV}(tx) \quad (1 \le i \le n)}{\begin{array}{c}\texttt{(dlambda ((}id_1\ k_1\texttt{)...(}id_n\ k_n\texttt{))}\ tx\texttt{)}\\ \sim \\ \texttt{(dlambda ((}id'_1\ k_1\texttt{)...(}id'_n\ k_n\texttt{))}\ [^n_{i=1} id'_i/id_i]tx\texttt{)}\end{array}}$$

2.2.5. (maxeff $ei_1 ... ei_n$)

A **maxeff** expression is the cumulative effect of the effects ei_i.

Static Semantics

$$\frac{TK \ \vdash\ ei_i :: \texttt{effect} \quad (1 \le i \le n)}{TK \ \vdash\ \texttt{(maxeff}\ ei_1 ... ei_n\texttt{)} :: \texttt{effect}}$$

Inclusion Semantics

$$\boxed{\texttt{(maxeff)} \sim \texttt{fx..pure}}$$

$$\boxed{\texttt{(maxeff}\ ei\texttt{)} \sim ei}$$

$$\boxed{\texttt{(maxeff}\ ei_1\ ei_2\texttt{)} \sim \texttt{(maxeff}\ ei_2\ ei_1\texttt{)}}$$

$$\boxed{\begin{array}{c}\texttt{(maxeff}\ ei_1\ \texttt{(maxeff}\ ei_2\ ei_3\texttt{))}\\ \sim \\ \texttt{(maxeff (maxeff}\ ei_1\ ei_2\texttt{)}\ ei_3\texttt{)}\end{array}}$$

$$\boxed{\texttt{(maxeff}\ ei\ ei\texttt{)} \sim ei}$$

2.2.6. (moduleof (abs $ida_1\ k_1$)...(abs $ida_n\ k_n$) (desc $idd_1\ dx_1$)...(desc $idd_p\ dx_p$) (val $idv_1\ tx_1$)...(val $idv_m\ tx_m$))

The ida_i, idd_k and idv_j must be distinct.

A **moduleof** expression is the type of modules that export the abstract descriptions ida_i, the transparent descriptions idd_k and the values idv_j. A **moduleof** expression is a binding construct.

Static Semantics

$$\frac{\begin{array}{l}TK[^n_{i=1} ida_i :: k_i] \vdash dx_k :: kd_k \quad (1 \le k \le p)\\ TK[^n_{i=1} ida_i :: k_i] \vdash [^p_{k=1} dx_k/idd_k]tx_j :: \texttt{type}\\ \qquad\qquad\qquad\qquad (1 \le j \le m)\end{array}}{\begin{array}{l}TK \vdash \texttt{(moduleof}\\ \quad \texttt{(abs}\ ida_1\ k_1\texttt{)...(abs}\ ida_n\ k_n\texttt{)}\\ \quad \texttt{(desc}\ idd_1\ dx_1\texttt{)...(desc}\ idd_p\ dx_p\texttt{)}\\ \quad \texttt{(val}\ idv_1\ tx_1\texttt{)...(val}\ idv_m\ tx_m\texttt{))} :: \texttt{type}\end{array}}$$

Inclusion Semantics

$$\frac{\pi,\ \sigma,\ \tau \text{ are permutations on } [1,n],\ [1,p],\ [1,m]}{\begin{array}{l}\texttt{(moduleof}\\ \quad \texttt{(abs}\ ida_1\ k_1\texttt{)...(abs}\ ida_n\ k_n\texttt{)}\\ \quad \texttt{(desc}\ idd_1\ dx_1\texttt{)...(desc}\ idd_p\ dx_p\texttt{)}\\ \quad \texttt{(val}\ idv_1\ tx_1\texttt{)...(val}\ idv_m\ tx_m\texttt{))}\\ \qquad\qquad \sim \\ \texttt{(moduleof}\\ \quad \texttt{(abs}\ ida_{\pi(1)}\ k_{\pi(1)}\texttt{)...(abs}\ ida_{\pi(n)}\ k_{\pi(n)}\texttt{)}\\ \quad \texttt{(desc}\ idd_{\sigma(1)}\ dx_{\sigma(1)}\texttt{)...(desc}\ idd_{\sigma(p)}\ dx_{\sigma(p)}\texttt{)}\\ \quad \texttt{(val}\ idv_{\tau(1)}\ tx_{\tau(1)}\texttt{)...(val}\ idv_{\tau(m)}\ tx_{\tau(m)}\texttt{))}\end{array}}$$

$$\frac{\begin{array}{rcl} dx_i & \sqsubset & dx'_i \quad (1 \le i \le p')\\ tx_j & \sqsubseteq & tx'_j \quad (1 \le j \le m')\\ n \text{ (resp. } m,p) & \ge & n' \text{ (resp. } m',p')\end{array}}{\begin{array}{l}\texttt{(moduleof}\\ \quad \texttt{(abs}\ ida_1\ k_1\texttt{)...(abs}\ ida_n\ k_n\texttt{)}\\ \quad \texttt{(desc}\ idd_1\ dx_1\texttt{)...(desc}\ idd_p\ dx_p\texttt{)}\\ \quad \texttt{(val}\ idv_1\ tx_1\texttt{)...(val}\ idv_m\ tx_m\texttt{))}\\ \qquad\qquad \sqsubset \\ \texttt{(moduleof}\\ \quad \texttt{(abs}\ ida_1\ k_1\texttt{)...(abs}\ ida_n\ k_{n'}\texttt{)}\\ \quad \texttt{(desc}\ idd_1\ dx'_1\texttt{)...(desc}\ idd_{p'}\ dx'_{p'}\texttt{)}\\ \quad \texttt{(val}\ idv_1\ tx'_1\texttt{)...(val}\ idv_{m'}\ tx'_{m'}\texttt{))}\end{array}}$$

2.2.7. (poly (($id_1\ k_1$)...($id_n\ k_n$)) tx)

The id_i must be distinct.

A **poly** expression is the type of polymorphic expressions abstracted over descriptions of kind k_i. A **poly** expression is a binding construct.

Static Semantics

$$\frac{TK[^n_{i=1} id_i :: k_i] \vdash tx :: \texttt{type}}{TK \vdash \texttt{(poly ((}id_1\ k_1\texttt{)...(}id_n\ k_n\texttt{))}\ tx\texttt{)} :: \texttt{type}}$$

Inclusion Semantics

$$\frac{tx \sqsubset tx'}{\begin{array}{c}\texttt{(poly ((}id_1\ k_1\texttt{)...(}id_n\ k_n\texttt{))}\ tx\texttt{)}\\ \sqsubset \\ \texttt{(poly ((}id_1\ k_1\texttt{)...(}id_n\ k_n\texttt{))}\ tx'\texttt{)}\end{array}}$$

$$\frac{id'_i \notin \mathrm{FV}(tx) \quad (1 \le i \le n)}{\begin{array}{c}\texttt{(poly ((}id_1\ k_1\texttt{)...(}id_n\ k_n\texttt{))}\ tx\texttt{)}\\ \sim \\ \texttt{(poly ((}id'_1\ k_1\texttt{)...(}id'_n\ k_n\texttt{))}\ [^n_{i=1} id'_i/id_i]tx\texttt{)}\end{array}}$$

2.2.8. (productof ($id_1\ tx_1$)...($id_n\ tx_n$))

The id_i must be distinct.

A **productof** expression is the type of aggregate values with named fields. Each field id_i corresponds to a value of type tx_i.

Static Semantics

$$\frac{TK \vdash\ tx_i :: \texttt{type} \quad (1 \le i \le n)}{TK \vdash\ \texttt{(productof (}id_1\ tx_1\texttt{)...(}id_n\ tx_n\texttt{))} :: \texttt{type}}$$

Inclusion Semantics

$$\frac{\begin{array}{c} tx_i \sqsubset tx'_i \quad (1 \le i \le m) \\ n \ge m \end{array}}{\begin{array}{c}(\texttt{productof}\ (id_1\ tx_1)\ldots(id_n\ tx_n)) \\ \sqsubset \\ (\texttt{productof}\ (id_1\ tx'_1)\ldots(id_m\ tx'_m))\end{array}}$$

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

$$\frac{\begin{array}{l} TK \vdash e : (\texttt{moduleof} \\ \quad (\texttt{abs}\ ida_1\ k_1)\ldots(\texttt{abs}\ ida_n\ k_n) \\ \quad (\texttt{desc}\ idd_1\ dx_1)\ldots \\ \quad (\texttt{val}\ idv_1\ tx_1)\ldots) \\ \quad !\ \texttt{fx..pure} \end{array}}{TK \vdash (\texttt{select}\ e\ ida_i) :: k_i \quad (1 \le i \le n)}$$

$$\frac{\begin{array}{l} TK \vdash e : (\texttt{moduleof} \\ \quad (\texttt{abs}\ ida_1\ k_1)\ldots(\texttt{abs}\ ida_n\ k_n) \\ \quad (\texttt{desc}\ idd_1\ dx_1)\ldots(\texttt{desc}\ idd_m\ dx_m) \\ \quad (\texttt{val}\ idv_1\ tx_1)\ldots)\ !\ \texttt{fx..pure} \\ TK[^n_{i=1} ida_i :: k_i] \vdash dx_j :: kd_j \quad (1 \le j \le m) \end{array}}{TK \vdash (\texttt{select}\ e\ idd_j) :: kd_j \quad (1 \le j \le m)}$$

Inclusion Semantics

$$\frac{\begin{array}{l} TK \vdash e : (\texttt{moduleof} \\ \quad (\texttt{abs}\ ida_1\ k_1)\ldots(\texttt{abs}\ ida_n\ k_n) \\ \quad (\texttt{desc}\ idd_1\ dx_1)\ldots(\texttt{desc}\ idd_m\ dx_m) \\ \quad (\texttt{val}\ idv_1\ tx_1)\ldots) \end{array}}{\begin{array}{c} (\texttt{select}\ e\ idd_i) \\ \sim \\ [^n_{j=1}(\texttt{select}\ e\ ida_j)/ida_j]dx_i \quad (1 \le i \le m) \end{array}}$$

2.2.10. (sumof (id_1 tx_1)...(id_n tx_n))

The id_i must be distinct.

A **sumof** expression is the type of tagged values of type tx_i with tag id_i.

Static Semantics

$$\frac{TK \vdash tx_i :: \texttt{type} \quad (1 \le i \le n)}{TK \vdash (\texttt{sumof}\ (id_1\ tx_1)\ldots(id_n\ tx_n)) :: \texttt{type}}$$

Inclusion Semantics

$$\frac{\begin{array}{c} tx_i \sqsubset tx'_i \quad (1 \le i \le n) \\ n \le n' \end{array}}{\begin{array}{c}(\texttt{sumof}\ (id_1\ tx_1)\ldots(id_n\ tx_n)) \\ \sqsubset \\ (\texttt{sumof}\ (id_1\ tx'_1)\ldots(id_{n'}\ tx'_{n'}))\end{array}}$$

$$\frac{\pi \text{ is a permutation on } [1,n]}{\begin{array}{c}(\texttt{sumof}\ (id_1\ tx_1)\ldots(id_n\ tx_n)) \\ \sim \\ (\texttt{sumof}\ (id_{\pi(1)}\ tx_{\pi(1)})\ldots(id_{\pi(n)}\ tx_{\pi(n)}))\end{array}}$$

2.3. Values

$$\begin{array}{rl} e ::= & literal \mid \\ & sugar \mid \\ & id \mid \\ & (e_0\ e_1 \ldots e_n) \mid \\ & (\texttt{begin}\ e_0\ e_1 \ldots e_n) \mid \\ & (\texttt{extend}\ e_0\ e_1) \mid \\ & (\texttt{extract}\ tx\ e\ id) \mid \\ & (\texttt{if}\ e_0\ e_1\ e_2) \mid \\ & (\texttt{lambda}\ ([id_1 \mid (id'_1\ tx'_1)]\ldots[id_n \mid (id'_n\ tx'_n)])\ e) \mid \\ & (\texttt{let}\ ((id_1\ e_1)\ldots(id_n\ e_n))\ e) \\ & (\texttt{load}\ literal) \mid \\ & (\texttt{module}\ (\texttt{define-abstraction}\ ida_1\ ka_1\ dxa_1)\ldots \\ & \qquad\qquad (\texttt{define-abstraction}\ ida_n\ ka_n\ dxa_n) \\ & \qquad\quad (\texttt{define-description}\ idd_1\ dxd_1)\ldots \\ & \qquad\qquad (\texttt{define-description}\ idd_m\ dxd_m) \\ & \qquad\quad (\texttt{define}\ idv_1\ e_1)\ldots(\texttt{define}\ idv_p\ e_p) \\ & \qquad\quad (\texttt{define-typed}\ idt_1\ tx_1\ e'_1)\ldots \\ & \qquad\qquad (\texttt{define-typed}\ idt_q\ tx_q\ e'_q)) \mid \\ & (\texttt{open}\ e) \mid \\ & (\texttt{plambda}\ ((id_1\ k_1)\ldots(id_n\ k_n))\ e) \mid \\ & (\texttt{product}\ tx\ e_1 \ldots e_n) \mid \\ & (\texttt{proj}\ e\ dx_1 \ldots dx_n) \mid \\ & (\texttt{sum}\ tx\ id\ e) \mid \\ & (\texttt{tagcase}\ tx\ e\ id\ e_1\ e_2) \mid \\ & (\texttt{the}\ tx\ e) \mid \\ & (\texttt{with}\ e_0\ e_1) \end{array}$$

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:

$$\frac{\begin{array}{lcl} TK & \vdash & e : tx\ !\ ei \\ ei & \sim & ei' \\ tx & \sim & tx' \end{array}}{TK \vdash e : tx'\ !\ ei'}$$

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* σ. A value is either a literal, or a list of values or expressions in brackets $\langle v_1 \ldots v_n \rangle$. 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

$$\begin{array}{lcl} TK & \vdash & \texttt{\#t} : \texttt{fx..bool}\ !\ \texttt{fx..pure} \\ TK & \vdash & \texttt{\#f} : \texttt{fx..bool}\ !\ \texttt{fx..pure} \\ TK & \vdash & \texttt{\#u} : \texttt{fx..unit}\ !\ \texttt{fx..pure} \end{array}$$

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 `#u`.

Static Semantics

$$\begin{array}{rcl}
TK[id : tx] \vdash id & : & tx \ \texttt{!}\ \texttt{fx..pure} \\
TK \vdash \texttt{fx.ref} & : & \texttt{(poly ((t type))} \\
 & & \quad \texttt{(-> fx..init} \\
 & & \qquad \texttt{((val0 t))} \\
 & & \qquad \texttt{(fx..refof t)))} \\
TK \vdash \texttt{fx.\^{}} & : & \texttt{(poly ((t type))} \\
 & & \quad \texttt{(-> fx..read} \\
 & & \qquad \texttt{((ref (fx..refof t)))} \\
 & & \qquad \texttt{t))} \\
TK \vdash \texttt{fx.:=} & : & \texttt{(poly ((t type))} \\
 & & \quad \texttt{(-> fx..write} \\
 & & \qquad \texttt{((ref (fx..refof t))} \\
 & & \qquad \ \texttt{(val1 t))} \\
 & & \qquad \texttt{fx..unit))}
\end{array}$$

Dynamic Semantics

$$\frac{(\texttt{(fx.ref}\ v\texttt{)}, \sigma) \quad (\text{with } l \text{ unbound in } \sigma)}{(\langle \texttt{*loc*}\ l \rangle, \sigma[l \mapsto v])}$$

$$\frac{(\texttt{(fx.\^{}}\ \langle \texttt{*loc*}\ l \rangle\texttt{)}, \sigma)}{(\sigma l, \sigma)}$$

$$\frac{(\texttt{(fx.:=}\ \langle \texttt{*loc*}\ l \rangle\ v\texttt{)}, \sigma)}{(\texttt{\#u}, \sigma[l \mapsto v])}$$

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 (after implicit projection if necessary, see open below) to v_i is returned.

Static Semantics

$$\frac{\begin{array}{rcl}
TK \vdash e_0 & : & \texttt{(->}\ ei\ \texttt{((}id_1\ tx_1\texttt{)}\ldots\texttt{(}id_n\ tx_n\texttt{))}\ tx_{n+1}\texttt{)} \\
 & ! & ei_0 \\
tx'_i & \sim & [_{j=1}^{i-1} e_j / id_j] tx_i \quad (1 \le i \le n+1) \\
TK \vdash tx'_i & :: & \texttt{type} \quad (1 \le i \le n+1) \\
TK \vdash e_i & : & tx'_i \ \texttt{!}\ ei_i \quad (1 \le i \le n)
\end{array}}{\begin{array}{rcl}
TK \vdash (e_0\ e_1 \ldots e_n) & : & tx'_{n+1} \\
 & ! & \texttt{(maxeff}\ ei\ ei_0\ ei_1 \ldots ei_n\texttt{)}
\end{array}}$$

Dynamic Semantics

$$\frac{(e_i, \sigma) \rightarrow (v_i, \sigma')}{((v_0 \ldots v_{i-1}\ e_i \ldots e_n), \sigma) \rightarrow ((v_0 \ldots v_i\ e_{i+1} \ldots e_n), \sigma')}$$

$$\begin{array}{c}
((\langle \texttt{*lambda*}\ (id_1 \ldots id_n)\ e \rangle\ v_1 \ldots v_n), \sigma) \\
\rightarrow \\
([_{i=1}^{n} v_i / id_i] e, \sigma)
\end{array}$$

2.3.4. (begin $e_0\ e_1 \ldots e_n$)

The expressions e_i are successively evaluated to values v_i and v_n is returned.

Static Semantics

$$\frac{TK \ \vdash \ e_i : tx_i \ \texttt{!}\ ei_i \quad (0 \le i \le n)}{\begin{array}{rcl}
TK \vdash \texttt{(begin}\ e_0\ e_1 \ldots e_n\texttt{)} & : & tx_n \\
 & ! & \texttt{(maxeff}\ ei_0\ ei_1 \ldots ei_n\texttt{)}
\end{array}}$$

Dynamic Semantics

$$\frac{(e, \sigma) \rightarrow (v, \sigma')}{(\texttt{(begin}\ e\texttt{)}, \sigma) \rightarrow (v, \sigma')}$$

$$\frac{(e_0, \sigma) \rightarrow (v_0, \sigma')}{\begin{array}{c}
(\texttt{(begin}\ e_0\ e_1\ e_2 \ldots\texttt{)}, \sigma) \\
\rightarrow \\
(\texttt{(begin}\ e_1\ e_2 \ldots\texttt{)}, \sigma')
\end{array}}$$

2.3.5. (extend $e_0\ e_1$)

The expression e_0 is evaluated to a module v_0 and the value of module e_1, extended with all the bindings introduced by 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_1 take precedence. An `extend` expression is a binding construct.

Static Semantics

$$\frac{\begin{array}{l}
TK \vdash e_0 : \texttt{(moduleof} \\
\qquad \texttt{(abs}\ ida_{01}\ k_{01}\texttt{)}\ldots\texttt{(abs}\ ida_{0n_0}\ k_{0n_0}\texttt{)} \\
\qquad \texttt{(desc}\ idd_{01}\ dx_{01}\texttt{)}\ldots\texttt{(desc}\ idd_{0m_0}\ dx_{0m_0}\texttt{)} \\
\qquad \texttt{(val}\ idv_{01}\ tx_{01}\texttt{)}\ldots\texttt{(val}\ idv_{0p_0}\ tx_{0p_0}\texttt{))} \\
\quad \texttt{!}\ ei_0 \\
TK \vdash \texttt{(with}\ e_0\ e_1\texttt{)} : tx \ \texttt{!}\ ei \\
tx \sim \texttt{(moduleof} \\
\qquad \texttt{(abs}\ ida_{11}\ k_{11}\texttt{)}\ldots\texttt{(abs}\ ida_{1n_1}\ k_{1n_1}\texttt{)} \\
\qquad \texttt{(desc}\ idd_{11}\ dx_{11}\texttt{)}\ldots\texttt{(desc}\ idd_{1m_1}\ dx_{1m_1}\texttt{)} \\
\qquad \texttt{(val}\ idv_{11}\ tx_{11}\texttt{)}\ldots\texttt{(val}\ idv_{1p_1}\ tx_{1p_1}\texttt{))} \\
\{_{i=1}^{n_2} ida_{2i}\} = \{_{i=1}^{n_0} ida_{0i}\} - \{_{i=1}^{n_1} ida_{1i}\} \\
\{_{i=1}^{m_2} idd_{2i}\} = \{_{i=1}^{m_0} idd_{0i}\} - \{_{i=1}^{m_1} idd_{1i}\} \\
\{_{i=1}^{p_2} idv_{2i}\} = \{_{i=1}^{p_0} idv_{0i}\} - \{_{i=1}^{p_1} idv_{1i}\}
\end{array}}{\begin{array}{l}
TK \vdash \texttt{(extend}\ e_0\ e_1\texttt{)} \\
\quad : \texttt{(moduleof} \\
\qquad \texttt{(abs}\ ida_{21}\ k_{21}\texttt{)}\ldots\texttt{(abs}\ ida_{2n_2}\ k_{2n_2}\texttt{)} \\
\qquad \texttt{(abs}\ ida_{11}\ k_{11}\texttt{)}\ldots\texttt{(abs}\ ida_{1n_1}\ k_{1n_1}\texttt{)} \\
\qquad \texttt{(desc}\ idd_{21}\ dx_{21}\texttt{)}\ldots\texttt{(desc}\ idd_{2m_2}\ dx_{2m_2}\texttt{)} \\
\qquad \texttt{(desc}\ idd_{11}\ dx_{11}\texttt{)}\ldots\texttt{(desc}\ idd_{1m_1}\ dx_{1m_1}\texttt{)} \\
\qquad \texttt{(val}\ idv_{21}\ tx_{21}\texttt{)}\ldots\texttt{(val}\ idv_{2p_2}\ tx_{2p_2}\texttt{)} \\
\qquad \texttt{(val}\ idv_{11}\ tx_{11}\texttt{)}\ldots\texttt{(val}\ idv_{1p_1}\ tx_{1p_1}\texttt{))} \\
\quad \texttt{!}\ \texttt{(maxeff}\ ei_0\ ei\texttt{)}
\end{array}}$$

Dynamic Semantics

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

$$\frac{(e_0,\sigma) \rightarrow (v_0,\sigma')}{((\texttt{extend}\ e_0\ e_1),\sigma) \rightarrow ((\texttt{*extend*}\ v_0\ (\texttt{with}\ v_0\ e_1)),\sigma')}$$

$$\frac{(e_1,\sigma) \rightarrow (v_1,\sigma')}{((\texttt{*extend*}\ v_0\ e_1),\sigma) \rightarrow ((\texttt{*extend*}\ v_0\ v_1),\sigma')}$$

$$\begin{array}{c} ((\texttt{*extend*}\ \langle\texttt{*module*}\ (id_1\ v_1)...(id_n\ v_n)\rangle \\ \langle\texttt{*module*}\ (id'_1\ v'_1)...(id'_m\ v'_m)\rangle),\sigma) \\ \rightarrow \\ (\langle\texttt{*module*}\ (id''_1\ v_1)...(id''_p\ v_p) \\ (id'_1\ v'_1)...(id'_m\ v'_m)\rangle,\sigma) \end{array}$$

where $\{_{k=1}^{p} id''_k\} = \{_{k=1}^{n} id_k\} - \{_{k=1}^{m} id'_k\}$.

2.3.6. (extract *tx e id*)

The expression e of product type tx is evaluated to an aggregate value v. The value of the field id of v is returned.

Static Semantics

$$\frac{\begin{array}{rcl} TK & \vdash & tx :: \texttt{type} \\ TK & \vdash & e : tx\ !\ ei \\ tx & \sim & (\texttt{productof}\ (id_1\ tx_1)...(id_n\ tx_n)) \end{array}}{TK \vdash (\texttt{extract}\ tx\ e\ id_i) : tx_i\ !\ ei}$$

Dynamic Semantics

$$\frac{(e,\sigma) \rightarrow (v,\sigma')}{\begin{array}{c} ((\texttt{extract}\ tx\ e\ id),\sigma) \\ \rightarrow \\ ((\texttt{extract}\ tx\ v\ id),\sigma') \end{array}}$$

$$\begin{array}{c} ((\texttt{extract}\ tx\ \langle\texttt{*product*}\ (id_1\ v_1)...(id_n\ v_n)\rangle\ id_i),\sigma) \\ \rightarrow \\ (v_i,\sigma) \end{array}$$

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 **#t** (resp. **#f**), then the value of e_1 (resp. e_2) is returned.

Static Semantics

$$\frac{\begin{array}{rcl} TK & \vdash & e_0 : \texttt{fx..bool}\ !\ ei_0 \\ TK & \vdash & e_1 : tx\ !\ ei_1 \\ TK & \vdash & e_2 : tx\ !\ ei_2 \end{array}}{TK \vdash (\texttt{if}\ e_0\ e_1\ e_2) : tx\ !\ (\texttt{maxeff}\ ei_0\ ei_1\ ei_2)}$$

Dynamic Semantics

$$\frac{(e_0,\sigma) \rightarrow (v_0,\sigma')}{((\texttt{if}\ e_0\ e_1\ e_2),\sigma) \rightarrow ((\texttt{if}\ v_0\ e_1\ e_2),\sigma')}$$

$$((\texttt{if}\ \texttt{\#t}\ e_1\ e_2),\sigma) \rightarrow (e_1,\sigma)$$

$$((\texttt{if}\ \texttt{\#f}\ e_1\ e_2),\sigma) \rightarrow (e_2,\sigma)$$

2.3.8. (lambda ([id_1 | (id'_1 tx'_1)]...[id_n | (id'_n tx'_n)]) e)

The id_i and id'_i must be distinct.

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

$$\frac{\begin{array}{c} TK_i = TK[_{j=1}^{i-1} id'_j : tx'_j] \quad (1 \leq i \leq n+1) \\ TK_i \vdash ti_i :: \texttt{type} \quad (1 \leq i \leq n) \\ TK_i \vdash tx'_i :: \texttt{type} \quad (1 \leq i \leq n) \\ TK_{n+1}[_{i=1}^{n} id_i : ti_i] \vdash e : tx\ !\ ei \end{array}}{\begin{array}{l} TK \vdash (\texttt{lambda}\ ([id_1 \mid (id'_1\ tx'_1)]...[id_n \mid (id'_n\ tx'_n)]) \\ \quad e) \\ \quad : (\texttt{->}\ ei\ ([(id_1\ ti_1) \mid (id'_1\ tx'_1)]... \\ \quad\quad [(id_n\ ti_n) \mid (id'_n\ tx'_n)])\ tx) \\ \quad !\ \texttt{fx..pure} \end{array}}$$

Dynamic Semantics

$$\begin{array}{c} ((\texttt{lambda}\ ([id_1 \mid (id'_1\ tx'_1)]...[id_n \mid (id'_n\ tx'_n)])\ e),\sigma) \\ \rightarrow \\ (\langle\texttt{*lambda*}\ ([id_1 \mid id'_1]...[id_n \mid id'_n])\ e\rangle,\sigma) \end{array}$$

2.3.9. (let ((id_1 e_1)...(id_n e_n)) 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

$$\frac{\begin{array}{c} TK \vdash e_i : tx_i\ !\ ei_i \quad (1 \leq i \leq n) \\ G = \{i \mid not\_expansive(e_i) \quad (1 \leq i \leq n)\} \\ TK[_{i \in [1,n]-G} id_i : tx_i] \vdash [_{i \in G} e_i/id_i]e : tx\ !\ ei \\ TK \vdash [_{i=1}^{n} e_i/id_i]tx :: \texttt{type} \end{array}}{\begin{array}{l} TK \vdash \quad (\texttt{let}\ ((id_1\ e_1)...(id_n\ e_n))\ e) \\ \quad\quad : [_{i=1}^{n} e_i/id_i]tx \\ \quad\quad !\ (\texttt{maxeff}\ ei_1...ei_n\ ei) \end{array}}$$

where an expression is *not_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 *not_expansive*.

Dynamic Semantics

$$\begin{array}{c}\texttt{((let ((}id_1\ e_1\texttt{)}\ldots\texttt{(}id_n\ e_n\texttt{))}\ e\texttt{)},\sigma\texttt{)}\\ \longrightarrow \\ \texttt{((}\langle\texttt{*lambda* (}id_1\ldots id_n\texttt{)}\ e\rangle\ e_1\ldots e_n\texttt{)},\sigma\texttt{)}\end{array}$$

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

$$\frac{\phi \vdash \texttt{(with fx (include } literal\texttt{))}\ \texttt{:}\ tx\ \texttt{!}\ ei}{TK \vdash \texttt{(load } literal\texttt{)}\ \texttt{:}\ tx\ \texttt{!}\ ei}$$

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

Dynamic Semantics

$$\frac{\texttt{((with fx (include } literal\texttt{))},\sigma\texttt{)} \rightarrow (v,\sigma')}{\texttt{((load } literal\texttt{)},\sigma\texttt{)} \rightarrow (v,\sigma')}$$

2.3.11.

```
(module (define-abstraction ida_1 k_1 dxa_1)...
          (define-abstraction ida_n k_n dxa_n)
        (define-description idd_1 dxd_1)...
          (define-description idd_m dxd_m)
        (define idv_1 e_1)...(define idv_p e_p)
        (define-typed idt_1 tx_1 e'_1)...
          (define-typed idt_q tx_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 dxa_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

$$\begin{array}{lcl}
TK_1 & = & TK[_{i=1}^{n} ida_i :: k_i] \\
TK_2 & = & TK_1[_{k=1}^{p} idv_k : ti_k][_{l=1}^{q} idt_l : [_{j=1}^{m} dxd_j/idd_j]tx_l] \\
 & & \quad [_{i=1}^{n}\texttt{up-}ida_i : Up(ida_i, k_i, dxa_i)] \\
 & & \quad [_{i=1}^{n}\texttt{down-}ida_i : Down(ida_i, k_i, dxa_i)] \\
TK_1 & \vdash & dxa_i :: k_i \quad (1 \le i \le n) \\
TK_1 & \vdash & dxd_j :: kd_j \quad (1 \le j \le m) \\
TK_1 & \vdash & [_{j=1}^{m} dxd_j/idd_j]tx_l :: \texttt{type} \quad (1 \le l \le q) \\
TK_2 & \vdash & [_{j=1}^{m} dxd_j/idd_j]e_k : ti_k\ \texttt{!}\ ei_k \quad (1 \le k \le p) \\
TK_2 & \vdash & [_{j=1}^{m} dxd_j/idd_j]e'_l : tx_l\ \texttt{!}\ ei'_l \quad (1 \le l \le q)
\end{array}$$

```
TK ⊢ (module
        (define-abstraction ida_1 k_1 dxa_1)...
          (define-abstraction ida_n k_n dxa_n)
        (define-description idd_1 dxd_1)...
          (define-description idd_m dxd_m)
        (define idv_1 e_1)...(define idv_p e_p)
        (define-typed idt_1 tx_1 e'_1)...
          (define-typed idt_q tx_q e'_q))
      : (moduleof
          (abs ida_1 k_1)...(abs ida_n k_n)
          (desc idd_1 dxd_1)...(desc idd_m dxd_m)
          (val idv_1 ti_1)...(val idv_p ti_p)
          (val idt_1 tx_1)...(val idt_q tx_q))
      ! (maxeff ei_1...ei_p ei'_1...ei'_q)
```

with the following definitions (where id_i are fresh):

$$\begin{array}{l}
Up(d_1, \texttt{type}, d_2) = \texttt{(-> fx..pure (}d_2\texttt{)}\ d_1\texttt{)} \\
Up(d_1, \texttt{(->>}\ k_1\ldots k_n\texttt{)}, d_2) = \\
\quad \texttt{(poly ((}id_1\ k_1\texttt{)}\ldots\texttt{(}id_n\ k_n\texttt{))} \\
\quad\quad Up(\texttt{(}d_1\ id_1\ldots id_n\texttt{)}, \texttt{type}, \texttt{(}d_2\ id_1\ldots id_n\texttt{)))} \\
Down(d_1, k, d_2) = Up(d_2, k, d_1)
\end{array}$$

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 dxa_1)...
     (define-abstraction ida_n k_n dxa_n)
   (define-description idd_1 dxd_1)...
     (define-description idd_m dxd_m)
   (define idv_1 e_1)...(define idv_p e_p)
   (define-typed idt_1 tx_1 e'_1)...
     (define-typed idt_q tx_q e'_q)),σ)
                    →
((*module-no-rec*
   (idv_1 [i=1..n (lambda (id) id)/up-ida_i]
          [i=1..n (lambda (id) id)/down-ida_i]e_1)...
   (idv_p [i=1..n (lambda (id) id)/up-ida_i]
          [i=1..n (lambda (id) id)/down-ida_i]e_p)
   (idt_1 [i=1..n (lambda (id) id)/up-ida_i]
          [i=1..n (lambda (id) id)/down-ida_i]e'_1)...
   (idt_q [i=1..n (lambda (id) id)/up-ida_i]
          [i=1..n (lambda (id) id)/down-ida_i]e'_q)),σ)
```

$$\frac{(e_k,\sigma) \rightarrow (v_k,\sigma')}{\begin{array}{l}((\mathtt{*module\text{-}no\text{-}rec*}\ (id_1\ v_1)\ldots(id_{k-1}\ v_{k-1})\\ \qquad (id_k\ e_k)\ldots(id_p\ e_p)),\sigma)\\ \qquad \rightarrow\\ ((\mathtt{*module\text{-}no\text{-}rec*}\ (id_1\ v_1)\ldots(id_k\ v_k)\\ \qquad (id_{k+1}\ [v_k/id_k]e_{k+1})\ldots\\ \qquad (id_p\ [v_k/id_k]e_p)),\sigma')\end{array}}$$

$$\begin{array}{l}((\mathtt{*module\text{-}no\text{-}rec*}\ (id_1\ v_1)\ldots(id_p\ v_p)),\sigma)\\ \qquad \rightarrow\\ ((\mathtt{*module*}\\ \quad (id_1\ [{}^{p}_{i=1}\langle\mathtt{*rec*}\ (\ldots^{p}_{j=1}(id_j\ v_j))\ id_i\rangle/id_i]v_1)\ldots\\ \quad (id_p\ [{}^{p}_{i=1}\langle\mathtt{*rec*}\ (\ldots^{p}_{j=1}(id_j\ v_j))\ id_i\rangle/id_i]v_p)\rangle,\sigma)\end{array}$$

$$\begin{array}{c}(\langle\mathtt{*rec*}\ ((id_1\ v_1)\ldots(id_p\ v_p))\ id_k\rangle,\sigma)\\ \rightarrow\\ ([{}^{p}_{i=1}\langle\mathtt{*rec*}\ ((id_1\ v_1)\ldots(id_p\ v_p))\ id_i\rangle/id_i]v_k,\sigma)\end{array}$$

2.3.12. (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 di_i. When a polymorphic value is directly applied to values, open is used to perform implicit projection.

Static Semantics

$$\frac{\begin{array}{lll}TK & \vdash & e : (\mathtt{poly}\ ((id_1\ k_1)\ldots(id_n\ k_n))\ tx)\ !\ ei\\ TK & \vdash & di_i :: k_i \quad (1 \le i \le n)\end{array}}{TK \vdash (\mathtt{open}\ e) : [{}^{n}_{i=1} di_i/id_i]tx\ !\ ei}$$

$$\frac{\begin{array}{lll}TK & \vdash & e_0 : (\mathtt{poly}\ ((id_1\ k_1)\ldots(id_m\ k_m))\ tx')\ !\ ei'\\ TK & \vdash & ((\mathtt{open}\ e_0)\ e_1\ldots e_n) : tx\ !\ ei\end{array}}{TK \vdash (e_0\ e_1\ldots e_n) : tx\ !\ ei}$$

Dynamic Semantics

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

2.3.13. (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 dx_i of kind k_i, returns the value of the pure expression e with the argument values dx_i substituted for the formals id_i. A plambda expression is a binding construct.

Static Semantics

$$\frac{TK[{}^{n}_{i=1} id_i :: k_i] \ \vdash\ e : tx\ !\ \mathtt{fx..pure}}{\begin{array}{l}TK \vdash (\mathtt{plambda}\ ((id_1\ k_1)\ldots(id_n\ k_n))\ e)\\ \qquad : (\mathtt{poly}\ ((id_1\ k_1)\ldots(id_n\ k_n))\ tx)\\ \qquad !\ \mathtt{fx..pure}\end{array}}$$

Dynamic Semantics

$$((\mathtt{plambda}\ ((id_1\ k_1)\ldots)\ e),\sigma) \rightarrow (e,\sigma)$$

2.3.14. (product tx $e_1\ldots e_n$)

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

Static Semantics

$$\frac{\begin{array}{lll}TK & \vdash & tx :: \mathtt{type}\\ tx & \sim & (\mathtt{productof}\ (id_1\ tx_1)\ldots(id_n\ tx_n))\\ TK & \vdash & e_i : tx_i\ !\ ei_i \quad (1 \le i \le n)\end{array}}{\begin{array}{lll}TK \vdash (\mathtt{product}\ tx\ e_1\ldots e_n) & : & tx\\ & ! & (\mathtt{maxeff}\ ei_1\ldots ei_n)\end{array}}$$

Dynamic Semantics

$$\begin{array}{c}((\mathtt{product}\ tx\ e_1\ldots e_n),\sigma)\\ \rightarrow\\ (((\mathtt{lambda}\ (id'_1\ldots id'_n)\ \langle\mathtt{*product*}\ (id_1\ id'_1)\ldots(id_n\ id'_n)\rangle)\\ e_1\ldots e_n),\sigma)\end{array}$$

where the id'_i are fresh.

2.3.15. (proj e $dx_1\ldots dx_n$)

A **proj** expression projects the polymorphic expression e onto the description expressions dx_i, returning the corresponding value.

Static Semantics

$$\frac{\begin{array}{lll}TK & \vdash & e : (\mathtt{poly}\ ((id_1\ k_1)\ldots(id_n\ k_n))\ tx)\ !\ ei\\ TK & \vdash & dx_i :: k_i \quad (1 \le i \le n)\end{array}}{TK \vdash (\mathtt{proj}\ e\ dx_1\ldots dx_n) : [{}^{n}_{i=1} dx_i/id_i]tx\ !\ ei}$$

Dynamic Semantics

$$((\mathtt{proj}\ e\ d_1\ldots),\sigma) \rightarrow (e,\sigma)$$

2.3.16. (sum tx id e)

The expression e is evaluated to v and a tagged value of sum type tx with tag id and value v is returned.

Static Semantics

$$\frac{\begin{array}{lll}TK & \vdash & tx :: \mathtt{type}\\ tx & \sim & (\mathtt{sumof}\ (id\ tx)\ldots)\\ TK & \vdash & e : tx\ !\ ei\end{array}}{TK \vdash (\mathtt{sum}\ tx\ id\ e) : tx\ !\ ei}$$

Dynamic Semantics

$$\frac{(e,\sigma) \rightarrow (v,\sigma')}{((\mathtt{sum}\ tx\ id\ e),\sigma) \rightarrow (\langle\mathtt{*sum*}\ id\ v\rangle,\sigma')}$$

2.3.17. $(\mathtt{tagcase}\ tx\ e\ id\ e_1\ e_2)$

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

Static Semantics

$$\frac{\begin{array}{lcl} TK & \vdash & tx :: \mathtt{type} \\ TK & \vdash & e : tx\ !\ ei \\ tx & \sim & (\mathtt{sumof}\ (id_1\ tx_1)\ldots(id_n\ tx_n)) \\ TK & \vdash & e_1 : (\texttt{->}\ ei_3\ ((id_j\ tx_j))\ tx_r)\ !\ ei_1 \\ TK & \vdash & e_2 : (\texttt{->}\ ei_4\ ((id\ tx))\ tx_r)\ !\ ei_2 \end{array}}{\begin{array}{lcl} TK & \vdash & (\mathtt{tagcase}\ tx\ e\ id_j\ e_1\ e_2) \\ & & \quad : tx_r \\ & & \quad !\ (\mathtt{maxeff}\ ei\ ei_1\ ei_2\ ei_3\ ei_4) \end{array}}$$

Dynamic Semantics

$$\begin{array}{c} ((\mathtt{tagcase}\ tx\ e\ id\ e_1\ e_2),\sigma) \\ \longrightarrow \\ (((\mathtt{lambda}\ (id_1\ id_2\ id_3) \\ \quad (\mathtt{tagcase}\ tx\ id_1\ id\ id_2\ id_3))\ e\ e_1\ e_2),\sigma) \end{array}$$

where id_i are fresh.

$$((\mathtt{tagcase}\ tx\ \langle\mathtt{*sum*}\ id\ v\rangle\ id\ v_1\ v_2),\sigma) \longrightarrow ((v_1\ v),\sigma)$$

$$\begin{array}{c} ((\mathtt{tagcase}\ tx\ \langle\mathtt{*sum*}\ id'\ v\rangle\ id\ v_1\ v_2),\sigma) \\ \longrightarrow \\ ((v_2\ \langle\mathtt{*sum*}\ id'\ v\rangle),\sigma) \end{array}$$

2.3.18. $(\mathtt{the}\ tx\ e)$

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

Static Semantics

$$\frac{\begin{array}{rcl} TK \vdash tx & :: & \mathtt{type} \\ TK \vdash e & : & tx'\ !\ ei \\ tx' & \sqsubset & tx \end{array}}{TK \vdash (\mathtt{the}\ tx\ e) : tx\ !\ ei}$$

Dynamic Semantics

$$((\mathtt{the}\ tx\ e),\sigma) \longrightarrow (e,\sigma)$$

2.3.19. $(\mathtt{with}\ e_0\ e_1)$

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.

Static Semantics

$$\frac{\begin{array}{l} TK \vdash e_0 : (\mathtt{moduleof} \\ \qquad (\mathtt{abs}\ ida_1\ k_1)\ldots(\mathtt{abs}\ ida_n\ k_n) \\ \qquad (\mathtt{desc}\ idd_1\ dx_1)\ldots(\mathtt{desc}\ idd_m\ dx_m) \\ \qquad (\mathtt{val}\ idv_1\ tx_1)\ldots(\mathtt{val}\ idv_p\ tx_p)) \\ \qquad !\ \mathtt{fx..pure} \\ \theta = [^p_{k=1}(\mathtt{with}\ e_0\ idv_k)/idv_k] \\ \qquad [^n_{i=1}(\mathtt{select}\ e_0\ ida_i)/ida_i] \\ \qquad [^m_{j=1} dx_j/idd_j] \end{array}}{TK \vdash (\mathtt{with}\ e_0\ idv_k) : \theta tx_k\ !\ \mathtt{fx..pure} \quad (1 \le k \le p)}$$

$$\frac{\begin{array}{l} TK \vdash e_0 : (\mathtt{moduleof} \\ \qquad (\mathtt{abs}\ ida_1\ k_1)\ldots(\mathtt{abs}\ ida_n\ k_n) \\ \qquad (\mathtt{desc}\ idd_1\ dx_1)\ldots(\mathtt{desc}\ idd_m\ dx_m) \\ \qquad (\mathtt{val}\ idv_1\ tx_1)\ldots(\mathtt{val}\ idv_p\ tx_p)) \\ \qquad !\ \mathtt{fx..pure} \\ \theta = [^p_{k=1}(\mathtt{with}\ e_0\ idv_k)/idv_k] \\ \qquad [^n_{i=1}(\mathtt{select}\ e_0\ ida_i)/ida_i] \\ \qquad [^m_{j=1} dx_j/idd_j] \\ TK \vdash \theta e_1 : tx\ !\ ei \end{array}}{TK \vdash (\mathtt{with}\ e_0\ e_1) : \theta tx\ !\ \theta ei}$$

Dynamic Semantics

$$\frac{(e_0,\sigma) \longrightarrow (v_0,\sigma')}{((\mathtt{with}\ e_0\ e_1),\sigma) \longrightarrow ((\mathtt{with}\ v_0\ e_1),\sigma')}$$

$$\begin{array}{|c|} \hline ((\mathtt{with}\ \langle\mathtt{*module*}\ (idv_1\ v_1)\ldots(idv_p\ v_p)\rangle\ e),\sigma) \\ \hline ([^p_{i=1} v_i/idv_i]e,\sigma) \\ \hline \end{array}$$

2.4. Sugars

$$\begin{array}{lcl} sugar & ::= & (\mathtt{and}\ e_1\ldots e_n)\ | \\ & & (\mathtt{cond}\ (e_1\ e'_1)\ldots(e_n\ e'_n)\ (\mathtt{else}\ e'_{n+1}))\ | \\ & & (\mathtt{let*}\ ((id_1\ e_1)\ldots(id_n\ e_n))\ e)\ | \\ & & (\mathtt{letrec}\ ((id_1\ e_1)\ldots(id_n\ e_n))\ e)\ | \\ & & (\mathtt{match}\ e\ (pat_1\ e_1)\ldots(pat_n\ e_n))\ | \\ & & (\mathtt{or}\ e_1\ldots e_n)\ | \\ & & id_1.id_2.\ldots.id_n.id\ | \\ & & id_1..id_2\ | \\ & & [e\ dx_1\ldots dx_n]\ | \\ & & (\mathtt{define}\ head\ e)\ | \\ & & (\mathtt{define\text{-}datatype}\ [(id\ (id_1\ k_1)\ldots(id_n\ k_n))\ |\ id] \\ & & \qquad (id'_1\ dx_{11}\ \ldots dx_{1m_1})\ldots \\ & & \qquad (id'_p\ dx_{p1}\ \ldots dx_{pm_p})) \\ & & (\mathtt{do}\ (id\ e_0\ e_i)\ (e_t\ e_r)\ e)\ | \\ & & (\mathtt{abs}\ (id_1\ id_2\ldots id_n)\ k)\ | \\ & & (\mathtt{val}\ (id_1\ id_2\ldots id_n)\ tx) \end{array}$$

$$\begin{array}{lcl} pat & ::= & literal\ | \\ & & \texttt{-}\ | \\ & & id\ | \\ & & (e\ pat_1\ldots pat_n) \end{array}$$

$$\begin{array}{lcl} head & ::= & id\ | \\ & & (head\ (id_1\ tx_1)\ldots(id_n\ tx_n))\ | \\ & & [head\ (id_1\ k_1)\ldots(id_n\ k_n)] \end{array}$$

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` $e_1 ... e_n$`)`

An `and` expression performs a short-circuit "and" evaluation of e_i to v_i, returning `#f` if one of the v_i is `#f`, `#t` otherwise.

Rewrite Semantics

- `#t` $(n = 0)$
- `(if` e_1 `(and` $e_2 ... e_n$`) #f)`

2.4.2. `(cond (`e_1 e'_1`)...(`e_n e'_n`) (else` e'_{n+1}`))`

A `cond` expression is a multiple-way test expression. The tests e_i are successively evaluated to v_i and as soon as one (say j) returns `#t` (or `else` is reached), the value of e'_j is returned.

Rewrite Semantics

- e'_{n+1} $(n = 0)$
- `(if` e_1 e'_1 `(cond (`e_2 e'_2`)...(`e_n e'_n`) (else` e'_{n+1}`)))`

2.4.3. `(let* ((`id_1 e_1`)...(`id_n e_n`))` e`)`

A `let*` expression successively binds each id_i to the value v_i of e_i evaluated in an augmented environment that binds id_j to v_j for j in $[1, i-1]$. The value of e, evaluated in an augmented environment that binds id_i to v_i, is returned.

Rewrite Semantics

- e $(n = 0)$
- `(let ((`id_1 e_1`)) (let* ((`id_2 e_2`)...(`id_n e_n`))` e`))`

2.4.4. `(letrec ((`id_1 e_1`)...(`id_n e_n`))` e`)`

A `letrec` expression recursively 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.

Rewrite Semantics

`(let ((`id `(module (define` id_1 e_1`)...(define` id_n e_n`))))`
`  (with` id e`))`
where id is fresh.

2.4.5. `(match` e `(`pat_1 e_1`)...(`pat_n e_n`))`

A `match` expression evaluates e to v and then performs a sequential match of v against the patterns pat_i. As soon as a match is found with a pattern pat_i, the value of e_i, evaluated in an environment in which the free variables of pat_i are bound to the appropriate components of v, is returned.

Rewrite Semantics

`(let ((`id e`))`
$\quad expand_{clause}(pat_1 ... pat_n,$
$\quad\quad e_1 ... e_n,$
$\quad\quad id,$
$\quad\quad$ `(lambda (x) x),`
$\quad\quad$ `(lambda (x)` *unspecified*`)))`

where id is fresh and the clause expansion function $expand_{clause}(pat_1 ... pat_n, e_1 ... e_n, v, s, f)$ is defined by:

- `(`f v`)`, if $n = 0$
- $expand_{exp}(pat_1,\ v,\ $`(`$s$ e_1`)`$,\ e')$ where e' is $expand_{clause}(pat_2 ... pat_n, e_2 ... e_n, v, s, f)$, 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 (`$id_1 ... id_n$`)` e'`) (lambda (`x`)` f'`))`, where the id_i and x are fresh and e' is $expand_{pat}(pat_1 ... pat_n, id_1 ... id_n, s', f')$, if pat is `(`e $pat_1 ... pat_n$`)`.

The pattern expansion function $expand_{pat}(pat_1 ... pat_n, id_1 ... id_n, s', f')$ is defined by:

- s', if $n = 0$
- $expand_{exp}(pat_1, id_1, e', f')$ where e' is $expand_{pat}(pat_2 ... pat_n, id_2 ... id_n, s', f')$, otherwise

2.4.6. `(or` $e_1 ... e_n$`)`

An `or` expression performs a short-circuit "or" evaluation of e_i to v_i, returning `#t` if one of the v_i is `#t`, `#f` otherwise.

Rewrite Semantics

- `#f` $(n = 0)$
- `(if` e_1 `#t (or` $e_2 ... e_n$`))`

2.4.7. $id_1.id_2....id_n.id$

An infix left-associative "dot" expression returns the value of id in the module that is the value of $id_1.id_2....id_n$.

Rewrite Semantics

- `(with` id_1 id`)` $(n = 1)$
- `(with` id_1 $id_2....id_n.id$`)`

2.4.8. $id_1..id_2$

A "dotdot" expression denotes the description expression bound to id_2 in the module id_1.

Rewrite Semantics

```
(select id1 id2)
```

2.4.9. $[e\ dx_1 ... dx_n]$

A [] expression returns the value of e projected on $dx_1 ... dx_n$.

Rewrite Semantics

```
(proj e dx1...dxn)
```

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 ((id1 tx1)...(idn txn)) e))`, if *head* is `(head' (id1 tx1)...(idn txn))`
- `(define head' (plambda ((id1 k1)...(idn kn)) e))`, if *head* is `[head' (id1 k1)...(idn kn)]`

2.4.11.

```
(define-datatype [(id (id1 k1)...(idn kn)) | id]
                 (id'1 dx11 ...dx1m1)...
                 (id'p dxp1 ...dxpmp))
```

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
    (->> k1...kn)
    (dlambda ((id1 k1)...(idn kn))
      (sumof (id'1 (productof (L1 dx11)...
                              (Lm1 dx1m1)))...
             (id'p (productof (L1 dxp1)...
                              (Lmp dxpmp))))))
  ```
- ```
  (define-description id-rep
    (dlambda ((id1 k1)...(idn kn))
      (sumof (id'1 (productof (L1 dx11)...
                              (Lm1 dx1m1)))...
             (id'p (productof (L1 dxp1)...
                              (Lmp dxpmp))))))
  ```
- ```
  (define-typed id'i
    (poly ((id1 k1)...(idn kn))
     (-> fx..pure
         ((id''1 dxi1)...(id''mi dximi))
         (id id1...idn)))
    (plambda ((id1 k1)...(idn kn))
     (lambda ((id''1 dxi1)...(id''mi dximi))
      (up-id (sum (id-rep id1...idn)
                  id'i
                  (product (productof
                             (L1 dxi1)...(Lmi dximi))
                           id''1...id''mi))))))
  ```
- ```
  (define-typed id'i~
    (poly ((id1 k1)...(idn kn)
           (x1 effect) (x2 effect) (t type))
     (-> fx..pure
         ((v (id id1...idn))
          (s (-> x1
                 ((id''1 dxi1)...(id''mi dximi))
                 t))
          (f (-> x2 ((v (id id1...idn))) t)))
         t))
    (plambda ((id1 k1)...(idn kn)
              (x1 effect) (x2 effect) (t type))
     (lambda
      ((v (id id1...idn))
       (s (-> x1 ((id''1 dxi1)...(id''mi dximi)) t))
       (f (-> x2 ((v (id id1...idn))) t)))
      (tagcase (id-rep id1...idn)
               (id-down v)
               id'i
               (lambda (vt)
                 (s (extract (productof
                               (L1 dxi1)...
                               (Lmi dximi))
                             vt
                             L1)...
                    (extract (productof
                               (L1 dxi1)...
                               (Lmi dximi))
                             vt
                             Lmi)))
               (lambda (x) (f v))))))
  ```

where L_i, id''_i, x_i, t, v, v_t, s, f and x are fresh.

2.4.12. (do (*id* e_0 e_i) (e_t e_r) *e*)

A do expression is a loop expression. The expression e is iteratively evaluated, while the value of e_t is `#f`, in an environment in which id is initially bound to e_0 and then to e_i in all subsequent iterations. Once e_t evaluates to `#t`, the value of e_r is returned.

Rewrite Semantics

```
(letrec ((id' (lambda (id)
                (if e_t
                    e_r
                    (begin e
                           (id' e_i))))))
  (id' e_0))
```

where *id'* is fresh.

2.4.13. (abs (id_1 id_2...id_n) k)

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

Rewrite Semantics

- (abs id_1 k) $(n = 1)$
- (abs id_1 k) (abs (id_2...id_n) k)

2.4.14. (val (id_1 id_2...id_n) tx)

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

Rewrite Semantics

- (val id_1 tx) $(n = 1)$
- (val id_1 tx) (val (id_2...id_n) tx)

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 *effect*

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

3.2. Init *effect*

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 *effect*

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 *effect*

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 *type*

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 *type*

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 *type*

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.

```
=          :    (-> pure ((i int) (j int)) bool)
<          :    (-> pure ((i int) (j int)) bool)
>          :    (-> pure ((i int) (j int)) bool)
<=         :    (-> pure ((i int) (j int)) bool)
>=         :    (-> pure ((i int) (j int)) bool)
+          :    (-> pure ((i int) (j int)) int)
*          :    (-> pure ((i int) (j int)) int)
-          :    (-> pure ((i int) (j int)) int)
/          :    (-> pure ((i int) (j int)) int)
neg        :    (-> pure ((i int)) int)
remainder  :    (-> pure ((i int) (j int)) int)
modulo     :    (-> pure ((i int) (j int)) int)
absolute   :    (-> pure ((i int)) int)
```

The subroutines =, <, >, <= and >= respectively return **#t** if **x** is equal, 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 and difference of i and j. / returns the truncated division of i by j. **Neg** returns the opposite of i. The subroutines **remainder** and **modulo** both return the rest of the number-theoretic integer division of i by j; they differ on negative arguments (the value returned by **remainder** has the same sign as i). **Absolute** returns the absolute value of i.

A dynamic error is signalled in case of division by zero or overflow. The range of integer values and subroutines is unspecified.

3.8. Float type

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.

```
fl=         :   (-> pure ((x float) (y float)) bool)
fl<         :   (-> pure ((x float) (y float)) bool)
fl>         :   (-> pure ((x float) (y float)) bool)
fl<=        :   (-> pure ((x float) (y float)) bool)
fl>=        :   (-> pure ((x float) (y float)) bool)
fl+         :   (-> pure ((x float) (y float)) float)
fl*         :   (-> pure ((x float) (y float)) float)
fl-         :   (-> pure ((x float) (y float)) float)
fl/         :   (-> pure ((x float) (y float)) float)
flneg       :   (-> pure ((x float)) float)
flabs       :   (-> pure ((x float)) float)
exp         :   (-> pure ((x float)) float)
log         :   (-> pure ((x float)) float)
sqrt        :   (-> pure ((x float)) float)
sin         :   (-> pure ((x float)) float)
cos         :   (-> pure ((x float)) float)
tan         :   (-> pure ((x float)) float)
asin        :   (-> pure ((x float)) float)
acos        :   (-> pure ((x float)) float)
atan        :   (-> pure ((x float)) float)
floor       :   (-> pure ((x float)) int)
ceiling     :   (-> pure ((x float)) int)
truncate    :   (-> pure ((x float)) int)
round       :   (-> pure ((x float)) int)
int->float  :   (-> pure ((x int)) float)
```

The subroutines **fl=**, **fl<**, **fl>**, **fl<=** and **fl>=** 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 **fl+**, **fl***, **fl-** and **fl/** respectively return the sum, product, difference and division of **x** and **y**. **Flneg** returns the opposite of **x**. **Flabs** returns the absolute value of **x**. **Exp** returns e to the power of **x**. **Log** returns the natural logarithm (in base e) of **x**. **Sqrt** returns the square root of **x**. The subroutines **sin**, **cos**, **tan**, **asin**, **acos** and **atan** respectively return the sine, cosine, tangent, arcsine (within $]-\pi/2,\pi/2]$), arccosine (within $]-\pi/2,\pi/2]$) and arctangent (within $]-\pi/2,\pi/2]$) of **x**. The subroutines **floor** and **ceiling** respectively return the largest and smallest integer not larger and smaller than **x**. **Truncate** returns the integer of largest absolute value not larger than **(flabs x)** and of same sign as **x**. **Round** returns the closest (even if tie) integer to **x**. **Int->float** returns the real x such that **(floor** x**) = (ceiling** x**) = x**.

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 floating point values and subroutines is unspecified: truncation may occur if the number of significant digits is too large.

3.9. Char type

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**.

```
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>=?              :  (-> 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)
char-alphabetic?     :  (-> pure ((c char)) bool)
char-numeric?        :  (-> pure ((c char)) bool)
char-whitespace?     :  (-> pure ((c char)) bool)
char-lower-case?     :  (-> pure ((c char)) bool)
char-upper-case?     :  (-> pure ((c 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)
```

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>?, 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 **type**

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-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)
string>=?       : (-> read
                      ((s string) (t string))
                      bool)
string-ci=?     : (-> read
                      ((s string) (t string))
                      bool)
string-ci<?     : (-> read
                      ((s string) (t string))
                      bool)
string-ci>?     : (-> read
                      ((s string) (t string))
                      bool)
string-ci<=?    : (-> read
                      ((s string) (t string))
                      bool)
string-ci>=?    : (-> read
                      ((s string) (t string))
                      bool)
substring       : (-> (maxeff init read)
                      ((s string) (from int) (to int))
                      string)
string-append   : (-> (maxeff init read)
                      ((head string) (tail string))
                      string)
string-copy     : (-> (maxeff init read)
                      ((s string))
                      string)
```

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>=? 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 **type**

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 ()).

```
sym->string    :    (-> init ((s sym)) string)
string->sym    :    (-> read ((s string)) sym)
sym=?          :    (-> pure
                        ((s sym) (t sym))
                        bool)
```

`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).

```
make-permutation :  (-> pure
                        ((pi (-> pure
                                 ((from int))
                                 int))
                         (length int))
                        permutation)
cshift           :  (-> pure
                        ((length int) (offset int))
                        permutation)
identity         :  (-> pure
                        ((length int))
                        permutation)
```

`Make-permutation` returns the permutation that maps every integer `from` in the interval [0,`length`[to (`pi 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,`length`[by `offset` positions on the right if `offset` is positive and by (`neg 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).

```
ref    :    (poly ((t type))
                (-> init ((val0  t)) (refof t)))
^      :    (poly ((t type))
                (-> read ((ref (refof t))) t))
:=     :    (poly ((t type))
                (-> write
                    ((ref (refof t))  (val1  t))
                    unit))
```

`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`.

```
unique   :   (poly ((t type))
                 (-> init ((x t)) (uniqueof t)))
value    :   (poly ((t type))
                 (-> pure ((u (uniqueof t))) t))
eq?      :   (poly ((t type))
                 (-> pure
                     ((u1 (uniqueof t))
                      (u2 (uniqueof t)))
                     bool))
```

`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`.

```
null      :   (poly ((t type))
                  (-> pure () (listof t)))
null?     :   (poly ((t type))
                  (-> pure ((list (listof t))) bool))
cons      :   (poly ((t type))
                  (-> init
                      ((car t) (cdr (listof t)))
                      (listof t)))
car       :   (poly ((t type))
                  (-> read ((list (listof t))) t))
cdr       :   (poly ((t type))
                  (-> read
                      ((list (listof t)))
                      (listof t)))
set-car!  :   (poly ((t type))
                  (-> write
                      ((list (listof t)) (new t))
                      unit))
set-cdr!  :   (poly ((t type))
                  (-> write
                      ((list (listof t)) (new (listof t)))
                      unit))
length    :   (poly ((t type))
                  (-> read ((list (listof t))) int))
append    :   (poly ((t type))
                  (-> (maxeff read init)
                      ((front (listof t))
                       (rear (listof t)))
                      (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
                        ((list (listof t)) (index int))
                        t))
map           :  (poly ((t1 type) (t2 type) (e effect))
                    (-> (maxeff e init read)
                        ((f (-> e ((x t1)) t2))
                         (list (listof t1)))
                        (listof t2)))
for-each      :  (poly ((t1 type) (t2 type) (e effect))
                    (-> (maxeff e read)
                        ((f (-> e ((x t1)) t2))
                         (list (listof t1)))
                        unit))
reduce        :  (poly ((t1 type) (t2 type) (e effect))
                    (-> (maxeff e read)
                        ((f (-> e ((x t1) (red t2)) t2))
                         (list (listof t1))
                         (seed t2))
                        t2))
list->string  :  (-> (maxeff read init)
                     ((chars (listof char)))
                     string)
string->list  :  (-> (maxeff read init)
                     ((chars string))
                     (listof char))
```

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.

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.

3.16. Vectorof (->> type)

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
                         ((vector (vectorof t)))
                         int))
vector-ref     :  (poly ((t type))
                     (-> read
                         ((vector (vectorof t))
                          (index int))
                         t))
vector-set!    :  (poly ((t type))
                     (-> write
                         ((vector (vectorof t))
                          (index int)
                          (new t))
                         unit))
vector-fill!   :  (poly ((t type))
                     (-> write
                         ((old (vectorof t))
                          (new t))
                         unit))
vector->list   :  (poly ((t type))
                     (-> (maxeff init read)
                         ((vector (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)
                      ((f (-> e ((v t1)) t2))
                       (vector (vectorof t1)))
                      (vectorof t2)))
vector-map2 :  (poly ((t1 type) (t2 type) (u type) (e effect))
                  (-> (maxeff e init read)
                      ((f (-> e ((v1 t1) (v2 t2)) u))
                       (vector1 (vectorof t1))
                       (vector2 (vectorof t2)))
                      (vectorof u)))
vector-reduce  :  (poly ((t type) (u type) (e effect))
                     (-> (maxeff e read)
                         ((f (-> e ((x t) (red u)) u))
                          (vector (vectorof t))
                          (seed u))
                         u))
```

```
scan          :   (poly ((t type) (e effect))
                    (-> (maxeff e init read)
                        ((f (-> e ((x t) (y t)) t))
                         (vector (vectorof t)))
                        (vectorof t)))
segmented-scan :  (poly ((t type) (e effect))
                    (-> (maxeff e init read)
                        ((f (-> e ((x t) (y t)) t))
                         (segments (vectorof bool))
                         (vector (vectorof t)))
                        (vectorof t)))
permute       :   (poly ((t type))
                    (-> (maxeff init read)
                        ((mapping permutation)
                         (vector (vectorof t)))
                        (vectorof t)))
compress      :   (poly ((t type))
                    (-> (maxeff init read)
                        ((selection (vectorof bool))
                         (vector (vectorof t)))
                        (vectorof t)))
expand        :   (poly ((t type))
                    (-> (maxeff init read)
                        ((selection (vectorof bool))
                         (vector (vectorof t))
                         (default (vectorof t)))
                        (vectorof t)))
eoshift       :   (poly ((t type))
                    (-> (maxeff init read)
                        ((offset int)
                         (vector (vectorof t))
                         (default (vectorof t)))
                        (vectorof t)))
```

Make-vector allocates and returns a vector of `length` elements, each having the given `value`. `Vector-length` returns the number of elements in `vector`. `Vector-ref` returns the `index`-th element of `vector`. `Vector-set!` replaces the `index`-th value of `vector` with `new` and returns `#u`. `Vector-fill!` replaces each element of `old` with `new` and returns `#u`. `Vector->list` returns a list constructed from the elements of `vector`. `List->vector` allocates and returns a vector constructed from the elements of `list`. `Vector-map` allocates and returns a vector that is obtained by applying `f` to each element `v` of `vector`. `Vector-map2` allocates and returns a vector that is obtained by applying `f` to each element `v1` of `vector1` and `v2` of `vector2`. `Vector-reduce` returns the result of the right-associative running (in `red`) applications of `f` with each element of `vector`. `Scan` allocates and returns a vector in which the element of offset i-1 is the reduction by `f` of the first i elements of `vector`. `Segmented-scan` allocates and returns a vector that contains the reductions by `f` of subvectors of `vector` corresponding to each contiguous sequence of `#f` of `segments`. `Permute` allocates and returns a vector obtained by permuting `vector` according to `mapping`; specifically, if `mapping` maps x to y, then `(vector-ref (permute mapping vector)` y`)` is `(vector-ref vector` x`)`. `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=?   :   (-> read ((s1 sexp) (s2 sexp)) bool)
```

`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 — type

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)
stream-write-char  :  (-> write
                          ((output stream) (value char))
                          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 ((st stream)) unit)
error              :  (poly ((t type))
                          (-> write
                              ((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 `#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`. 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 `-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

[GJLS87] Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A. *The FX-87 Reference Manual.* MIT/LCS/TR-407, 1987.

[HG88] Hammel, R. T. and Gifford, D. K. *FX-87 Performance Measurements: Dataflow Implementation*, MIT/LCS/TR-421, 1988.

[JG91] Jouvelot, P. and Gifford, D. K. Algebraic Reconstruction of Types and Effects. In *Proceedings of the ACM Conference on Principles Of Programming Languages.* ACM, New York, 1991.

[LG88] Lucassen, J. M. and Gifford, D. K. Polymorphic Effect Systems. In *Proceedings of the ACM Conference on Principles Of Programming Languages.* ACM, New York, 1988.

[MTH90] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[OG89] O'Toole, J. and Gifford, D. K. Polymorphic Type Reconstruction. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, New York, 1989.

[R86] Rees, J. A. and Clinger, W. Eds. *The Revised^3 Report on the Algorithmic Language Scheme.* MIT/AI Memo 848a, 1986.

[SG90] Sheldon, M. A. and Gifford, D. K. Static Dependent Types for First Class Modules. In *Proceedings of the ACM Lisp and Functional Programming Conference.* ACM, New York, 1990.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES