Semantics of Kernel/CORE Language

Wojciech Fraczak

Université du Québec en Outaouais
Gatineau, Québec, Canada

Revision: 1.13

Abstract

A semantics of Kernel/CORE Language is described. In addition, we investigate a "concrete semantics" of Kernel Language which can be used as a starting point for design of a high-performance compiler.

1 Introduction

This paper is composed of two parts. In the first part we describe in an abstract but precise way the main concepts of Kernel/CORE syntax and semantics such as *basic types*, *templates*, *environment*, and *traces*. The second part defines what we call "concrete semantics". Concrete semantics defines an encoding of flat values and practical approach to the traces, partitioning the trace alphabet into input and output.

2 Abstract syntax

Most often, the goal of a Kernel/CORE pogram is to define a "*relation*". Such a program consists of basic type definitions and relation definitions. Kernel is "flat", which means that a relation is a set of pairs of non-functional values (we call them *flat*, *printable*, or *first-order* values) of basic types.

2.1 Basic types

All basic types are defined using two constructs: *disjoint union* using square brackets and *product* using curly brackets.

```
T ::=   [ 'l1 T1, ..., 'ln Tn ]  |  { 'l1 T1, ..., 'ln Tn }
```

Unit type corresponds to the empty product, i.e., {}; it carries only one value also denoted by {}. *Empty type* which carries no value corresponds to the empty union, [].

Names of union and product fields are called *labels* and are prefixed by a single quote. Within one union or one product definition labels cannot repeat, however globally they can be reused. E.g.:

```
<type> two_letter_alphabet <is> ['a {}, 'b {}];
<type> three_letter_alphabet <is> ['a {},'b {}, 'c {}];
```

Types can be named and recursive types can be defined. E.g.:

```
<type> List_of_bits <is> [ 'empty {}, '0 List_of_bits, '1 List_of_bits ];
```

Every basic type defines an enumerable set of *flat values*, which are finite, edge labeled, rooted trees. Basic types are represented as vertices of a minimal labeled *and-or* graph, called *type graph*: product types are presented by *and* vertices and union nodes by *or* vertices. The *unit* type is an *and* vertex without children. The type graph can be seen as a deterministic top down tree automaton: the set of values of the type corresponds to the set of trees accepted by the corresponding tree automaton.

2.2 Relations

A relation f (also called function since it generalizes the notion of a function) is a set of ordered pairs of flat values, $f \subseteq D_f \times I_f$, where D_f and I_f are two sets of values of some basic types. Intuitively, D_f denotes the domain of f and I_f the image of f. The pair of basic types (t_D, t_I) corresponding to D_f and I_f is called the *type* of f and denoted by $f : T_D \to T_I$. If for every $x \in D_f$, the set $f(x) \stackrel{\text{def}}{=} \{y \in I_f \mid (x, y) \in f\}$ is empty or a singleton, then f defines a partial mapping from D_f to I_f.

A relation f from unit type, i.e., $f : \{\} \to I_f$, is called a constant. A named relation definition in Kernel looks like follows:

```
(Tx -> Ty) rel_name = rel_body ;
```

The body of a relation is an expression `E` of the following form:

```
E  ::=  rel  |  @label  |  $var  |  'label E  |  E . label
        |  product  |  union
        |  E :: E  |  ( $var P -> E )
product  ::=  { 'label E, 'label E, ... }
union  ::=  [ E, E, ... ]
rel, label, var  ::=  identifier
```

where `@label` is a reference to a member of product constructor labeled by `'label`, `$var` is a parameter name, `'label` is a *label*, and `P` is a *template*. The intuitive meaning of the above constructs are:

`rel` — reference to relation named `rel`;

`@label` — reference to a field of a product constructor;

`$var` — reference to the actual parameter value;

`'label E` — variant value constructor;

`E . label` — projection/selector;

`{ 'label E, ... }` — product value constructor;

`[ E, ... ]` — union operation;

`E :: E` — composition; and

`($x P -> E)` — abstraction.

Intuitively, the product value constructor represents parallel composition in which corresponding field values are calculated by evaluation of sub-expressions. *Product annotations* can be added to the expression (just before closing squarle backet) in order to specify how the interaction of parallel sub-processes with environment should be performed. Presently two strategies exist:

- empty annotation (i.e., default) — in this mode the subprocesses interact with environment synchronously; and
- `<serialize> ChanName` — interaction of the subprocesses with environment is serialized on the pipe `ChanName`; the order of the interaction in that channel is determined by the actual enumeration order of the fields in the product expression.

The union operation is defined as union of relations. In general, union of functions yields a relation which is not a function. We introduce *union annotations* which flattens the relation into a function (assumed all arguments are functions) by ordering elements of the relation:

- empty annotation (default) — only the first sub-expression which is defined is considered,
- `<longest> ChanName` — the sub-expression which is defined and which interacts the *longest* on pipe "`ChanName`" is considered, and
- `<shortest> ChanName` — the sub-expression which is defined and which interacts the *shortest* on pipe "`ChanName`" is considered.
- `<unknown>` (must be the last annotation) explicitely states that the order is unknown (e.g., for verification/specification/abstraction purposes).

We require that every Kernel Language expression is of first order type, and thus parameters have to be of a basic type, expressions defining the body of a functional abstraction have to be a constant, and all arguments for product constructors are constants.

2.2.1 Sub-expressions and free references

For every Kernel subexpresion p we define the set of the free references in the usual recursive way:

- If p is a parameter name or a label of a product constructor, i.e., $p = \$x$ or $p = @l$, then $\texttt{free}(p) = \{p\}$.
- If p is a relation name, then $\texttt{free}(p) = \emptyset$, i.e., empty set.
- If p is a pipeline (composition) or a union operation, i.e., $p = p_1 :: p_2$ or $[p_1, p_2]$ then $\texttt{free}(p) = \texttt{free}(p_1) \cup \texttt{free}(p_2)$.
- If p is a product constructor, i.e., $\{f_1\ p_1, f_2\ p_2\}$, then $\texttt{free}(p) = (\texttt{free}(p_1) \cup \texttt{free}(p_2)) \setminus \{@f_1, @f_2\}$.
- If p is a projection or a variant, i.e., $p = q.l$ or $p =' l\ q$, then $\texttt{free}(p) = \texttt{free}(q)$.
- If p is an abstraction, i.e., $p = (\$x... \to q)$, then $\texttt{free}(p) = \texttt{q} \setminus \{\$x\}$.

A subexpression p is called *closed* if $\texttt{free}(p)$ is empty.

Let V be a countable set of *variable names*, each typed by a basic type. We call *context*, any partial mapping from (a finite part of) V into values of corresponding types. E.g., if $V = \{x, y\}$ such that x is of type T_x and y is of type T_y, then $C : \{x \mapsto v_x, y \mapsto v_y\}$ is a context, with v_x and v_y being some values of types T_x and T_y, respectively. The mapping defined by a context naturally extends to Kernel expressions.

2.2.2 Patterns (templates)

Templates play the role of filters for values of the actual parameters. Every flat type definition defines the set of values (trees) together with a syntax for unary predicates on them, which we call patterns. Intuitively, a pattern is a initial part of the tree (from the root) which selects all values with the same initial part.

```
P  ::=  T  | { 'l1 P1, ..., 'ln Pn } | 'label P
```

where `T` is a type expression, `'l1`, ..., `'ln`, `'label` are labels, and `P1`, ..., `Pn`, `P` are patterns (templates).

The semantics of a pattern for a type is a non-empty subset of values of the type.

3 Semantics

A Kernel program can describe functional and behavioral aspects of computing. Even though we introduced some constructions (such as three kinds of union and two kinds of product constructors) which explicitly deal with behavioral aspect of computing, we have not provided any primitives.

We believe that Kernel can be used in various contexts in which the primitives would be very different. Thus, we assume that Kernel is used in a context for which the set of *synchronization events* is defined apart and it is accessible to a Kernel Language user through a set of predefined relations and basic types.

More precisely, we assume the existence of *Kernel environment*, $\mathcal{E}$ defined elsewhere, which consists of a set of *synchronization events* and a composition operation which will allow us to build complex *traces* (i.e., sequences of synchronization events) starting from atomic traces consisting of a single synchronization event. Such an environment can be formalized by a notion of *monoid*. A monoid is a triple $\mathcal{M} = (M, \cdot, 1)$, where M is a set, $\cdot$ is an associative total mapping $M \times M \mapsto M$, and 1 is an elements from M called unit, which verifies $x \cdot 1 = 1 \cdot x = x$, for all $x \in M$. In a monoid $(M, \cdot, 1)$, we define *prefix relation*, $\leq$, as follows: $x \leq y$ if there exists $z \in M$ such that $xz = y$.

We will suppose that a monoid $\mathcal{E} = (M, \cdot, 1)$ which represents a Kernel environment verifies the two following conditions

- *prefix relation* is an order, i.e., $x = xyz$ implies $x = xy$, for all $x, y, z \in M$; and
- *prefix relation* generates unique infimum, i.e., $x \vee y \stackrel{\texttt{def}}{=} \min\{z \in M \mid x \leq z, y \leq z\}$, if exists, is unique. If $x \vee y$ exists then we write $x \perp y$. Otherwise, i.e., when $x \vee y$ does not exist, we write $x \not\perp y$.

Proposition 1. *The following monoids are* environment monoids*:*

- *any free monoid;*
- *any set monoid (i.e., idempotent commutative free monoid);*
- *if M_1 and M_2 are environment monoids then $M_1 \times M_2$ is an environment monoid.*

Proposition 1 allows us to consider the following syntax for defining monoids which, by construction, will be environment monoids:

$$M \ ::= \ \{e_1, \ldots, e_n\}^* \ \mid \ \mathcal{P}\{e_1, \ldots, e_n\} \ \mid \ M \times M$$

for any $n \geq 0$. Construction $\{e_1, \ldots, e_n\}^*$ defines a free monoid generated by n prime elements $e_1, \ldots, e_n$, and construction $\mathcal{P}\{e_1, \ldots, e_n\}$ defines a set monoid generated by n prime elements $e_1, \ldots, e_n$.

From a practical point of view one can see the environment as a collection of FIFO channels $C_1, C_2, \ldots, C_k$. In such a model, every channel defines a free monoid of "atomic data packets" which can be carried by the channel. The whole environment is the product monoid of all those free monoids. In such a case, the modifiers of union and product constructions can be selectively chosen per the channel and the direction (read or write) basis (called also "dimension").

3.1 Non recursive definitions

With every Kernel subexpression p of type $(T_a \mapsto T_b)$ we associate its semantics, $\|p\|$, i.e., a mapping from a context defined on $\texttt{free}(p)$ to a set of quadruples (m, n, a, b), where m and n are elements of the environment monoid, a is a flat value of type T_a, and b is a flat value of type T_b. Instead of writing $\|p\|(C)$ we will write $\|p\|_C$. If p is closed then $\|p\|_C$ does not depend on C and thus we will write $\|p\|$ for denoting the unique set of quadruples (m, n, a, b) belonging to $\|p\|_C$.

Intuitively, a quadruple $(m, n, a, b) \in \|q\|$ means, that q can map a into b in the environment context mn by effectively consuming m (the n part represents a "look-ahead").

unit: $\|\{\}\|_C \stackrel{\texttt{def}}{=} \{(1, n, \{\}, \{\}) \mid n \in M\}$.

identifier: $\|\$x\|_C \stackrel{\texttt{def}}{=} \{(1, n, \{\}, C(\$x)) \mid n \in M\}$.

variant: $\|f\ q\|_C \stackrel{\texttt{def}}{=} \{(m, n, a, f\ b) \mid (m, n, a, b) \in \|q\|_C\}$.

projection: $\|q.f\|_C \stackrel{\texttt{def}}{=} \{(m, n, a, b.f) \mid (m, n, a, b) \in \|q\|_C\}$.

structure: We distinguish two synchronization modes which yields two product value constructors (defined by "dimension"):

$$\|\texttt{CONCAT}\{f_1\ q_1, f_2\ q_2\}\|_C \stackrel{\texttt{def}}{=} \{(m_1 m_2, m_2^{-1}(n_1 \vee m_2 n_2), \{\}, \{f_1\ b_1, f_2\ b_2\}) \mid (m_1, n_1, \{\}, b_1) \in \|q_1\|_C, (m_2, n_2, \{\}, b_2) \in \|q_2\|_{C.[@f_1 \mapsto b_1]}\}$$

$$\|\texttt{INTER}\{f_1\ q_1, f_2\ q_2\}\|_C \stackrel{\texttt{def}}{=} \{(m_1 \vee m_2, (m_1 \vee m_2)^{-1}(m_1 n_1 \vee m_2 n_2), \{\}, \{f_1\ b_1, f_2\ b_2\}) \mid (m_1, n_1, \{\}, b_1) \in \|q_1\|_C, (m_2, n_2, \{\}, b_2) \in \|q_2\|_{C.[@f_1 \mapsto b_1]}\}$$

union: Disambiguation can be done in three different ways.

$$\|\texttt{FIRST}[q_1, q_2]\|_C \stackrel{\texttt{def}}{=} \|q_1\|_C \cup \{(m, nn', a, b) \mid n' \in M, (m, n, a, b) \in \|q_2\|_C, \forall (m_1, n_1, a, b_1) \in \|q_1\|_C\, m_1 n_1 \not\perp mnn'\}$$

$$\|\texttt{LONGEST}[q_1, q_2]\|_C \stackrel{\texttt{def}}{=} \{(m, n, a, b) \in \|q_1\|_C \mid \forall (m', n', a, b') \in \|q_2\|_C\, (m'n' \perp mn \Rightarrow m \not\leq m')\} \cup \{(m, n, a, b) \in \|q_2\|_C \mid \forall (m', n', a, b') \in \|q_1\|_C\, (m'n' \perp mn \Rightarrow m' < m)\}$$

$$\|\texttt{SHORTEST}[q_1, q_2]\|_C \stackrel{\texttt{def}}{=} \{(m, n, a, b) \in \|q_1\|_C \mid \forall (m', n', a, b') \in \|q_2\|_C\, (m'n' \perp mn \Rightarrow m' \not\leq m)\} \cup \{(m, n, a, b) \in \|q_2\|_C \mid \forall (m', n', a, b') \in \|q_1\|_C\, (m'n' \perp mn \Rightarrow m < m')\}$$

composition: $\|q_1 :: q_2\|_C \stackrel{\texttt{def}}{=} \{(m_1 m_2, m_2^{-1}(n_1 \vee m_2 n_2), a, b) \mid (m_1, n_1, a, c) \in \|q_1\|_C, (m_2, n_2, c, b) \in \|q_2\|_C\}$

abstraction: $\|(\$x\, P \mapsto q)\|_C \stackrel{\texttt{def}}{=} \{(m, n, a, b) \mid (m, n, \{\}, b) \in \|q\|_{C.[\$x \mapsto a]}\}$, where $C.[\$x \mapsto a]$ denotes the following context:

$$C.[\$x \mapsto a](v) \stackrel{\texttt{def}}{=} \begin{cases} a & \text{if } v = \$x \\ C(v) & \text{otherwise} \end{cases}$$

The above rules describe the semantics of all non-recursive Kernel definitions.
We say that $S = \|q\|_C$ for a subexpression $q : T_1 \mapsto T_b$ is:

- *functional*, whenever $(m, n, a, b), (m, n, a, b') \in S$ implies $b = b'$;
- *prefix-deterministic*, whenever $(m, n, a, b), (m', n', a, b') \in S$ and $mn \perp m'n'$ implies $m = m'$ and $b = b'$;
- *finite*, whenever $\{(m, a, b) \mid (m, n, a, b) \in S\}$ is finite;
- *rational*, whenever $\{(m, a, b) \mid (m, n, a, b) \in S\}$ is rational;
- *simple*, ???
- *k-simple*, ???

Proposition 2. *All above operations on S preserve the above properties of its arguments. I.e., if $\|q_i\|_C$ are* functional *(resp., prefix-deterministic, finite, rational), then variant, projection, concat, inter, first-match, longest-match, shortest-match, composition, and abstraction are* functional *(resp., prefix-deterministic, finite, rational).*

Proof. No proof so far. It has to be proved! □

3.1.1 Examples

We consider the following environment:

$$M \;=\; \{\texttt{get}[x] \mid x \in \texttt{BYTE}\}^* \times \{\texttt{put}[x] \mid x \in \texttt{BYTE}\}^* \times \{\texttt{lookup}[x, y] \mid x, y \in \texttt{BYTE}\}^*$$

where `BYTE` is a finite indexing set isomorphic to the set of integers from 0 to 255. Thus, the primes (atomic actions) are: $\texttt{get}[0], \ldots, \texttt{put}[255], \texttt{put}[0], \ldots, \texttt{put}[255], \texttt{get}[0, 0], \ldots, \texttt{put}[0, 255], \mathit{textttput}[1,$
The above environment can be seen as two FIFOs, one for reading and one for writing.

We assume the following Kernel declarations corresponding to the primes:

```
<type> BYTE <is> {'0, '1, '2, ..., '255};
BYTE getByte = <input>;           // get
(BYTE -> {}) putByte = <output>;  // put[]
```

Notice that `getByte` and returns a value. In order to keep a value produced by the prime, we put it as an index, i.e., if a particular instance of `getByte` returns value x, then we denote it by $\texttt{get}_x$.

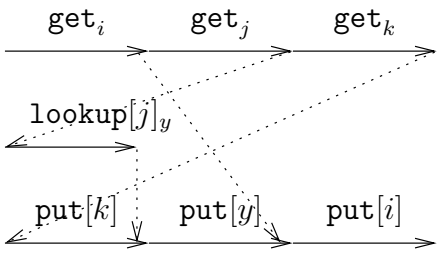
Let us consider the following Kernel function which reads three bytes and outputs third byte first, then it outputs the result of the lookup on the second byte, and finally outputs the first read byte.

```
type TRIPLE is {'a BYTE, 'b BYTE, 'c BYTE };
{} example1 =
   {'a getByte, 'b getByte, 'c getByte <serialize> getByte}
   :: ($x TRIPLE ->
       $x.c :: putByte ::
       $x.b :: lookup :: putByte ::
       $x.a :: putByte);
```

The semantics of function `example1` is calculated bottom up in the following way:

$$
\begin{aligned}
\|getByte\| &\stackrel{\text{def}}{=} (\texttt{get}_x, *, \{\}, x) \\
\|putByte\| &\stackrel{\text{def}}{=} (\texttt{put}[x]_y, *, x, \{\}) \\
\|lookup\| &\stackrel{\text{def}}{=} (\texttt{lookup}[x]_y, *, x, y) \\
\|\texttt{CONCAT}\{a\ \texttt{getByte}, \ldots, c\ \texttt{getByte}\}\| &= (\texttt{get}_i\texttt{get}_j\texttt{get}_k, *, \{\}, \{a\ i, b\ j, c\ k\}) \\
\|\$x.a\|_C &= (1, *, \{\}, C(x).a) \\
\|\$x.c :: \texttt{putByte}\|_C &= (\texttt{put}[C(x).c], *, \{\}, \{\}) \\
\|\$x.c :: \texttt{putByte} :: \$x.b :: \texttt{lookup}\|_C &= (\texttt{put}[C(x).c]\texttt{lookup}[C(x).b]_y, *, \{\}, y) \\
\|\$x.c :: \ldots :: \texttt{putByte}\|_C &= (\texttt{put}[C(x).c]\texttt{put}[y]\texttt{put}[C(x).a]\texttt{lookup}[C(x).b]_y, *, \{\}, \{\}) \\
\|(\$x\ \texttt{TRIPLE ->} \ldots)\| &= (\texttt{put}[x.c]\texttt{put}[y]\texttt{put}[x.a]\texttt{lookup}[x.b]_y, *, x, \{\}) \\
\|\texttt{example1}\| &= (\texttt{get}_i\texttt{get}_j\texttt{get}_k\texttt{put}[k]\texttt{put}[y]\texttt{put}[i]\texttt{lookup}[j]_y, *, \{\}, \{\})
\end{aligned}
$$

The semantics of `example1` can also be depicted by the following figure, with explicit causality.



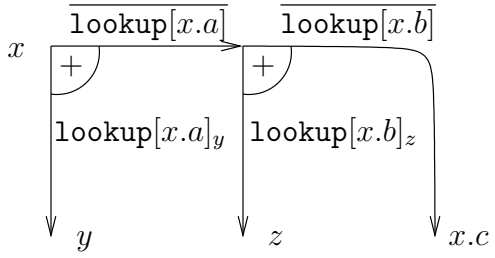

```
(TRIPLE -> BYTE) example2 =
   ($x TRIPLE -> [
       $x.^a :: lookup,
       $x.^b :: lookup,
       $x.^c ]);
```

The semantics of function `example2` is calculated bottom up in the following way (by

$\overline{\texttt{lookup}[a]}$ we denote the failure of action $\texttt{lookup}[a]$):

$$\begin{aligned}
\|\$x.a :: \texttt{lookup}\|_C &= (\texttt{lookup}[C(x).a]_y, *, \{\}, y) \\
\|[\ldots, \$x.b :: \texttt{lookup}]\|_C &= (\texttt{lookup}[C(x).a]_y, *, \{\}, y) \ + \\
&\quad (\texttt{lookup}[C(x).b]_z, \overline{\texttt{lookup}[C(x).a]}, \{\}, z) \\
\|[\ldots, \$x.c]\|_C &= (\texttt{lookup}[C(x).a]_y, *, \{\}, y) \ + \\
&\quad (\texttt{lookup}[C(x).b]_z, \overline{\texttt{lookup}[C(x).a]}, \{\}, z) \ + \\
&\quad (1, \overline{\texttt{lookup}[C(x).a]}\,\overline{\texttt{lookup}[C(x).b]}, \{\}, C(x).c) \\
\|\texttt{example2}\| &= (\texttt{lookup}[x.a]_y, *, x, y) \ + \\
&\quad (\texttt{lookup}[x.b]_z, \overline{\texttt{lookup}[x.a]}, x, z) \ + \\
&\quad (1, \overline{\texttt{lookup}[x.a]}\,\overline{\texttt{lookup}[x.b]}, x, x.c)
\end{aligned}$$

Many graphical represenations (implementations) can be used to depict the semantics of `example2`, which consists of three exclusive alternatives. For example:



3.2 Recursive definitions

Let $\vec{x} = (x_1, \ldots, x_n)$ be a vector of names of closed recursive expressions of form $x_i = q_i(\vec{x})$, for $i \in [1, n]$. The semantics $\vec{S} = (\|x_1\|, \ldots, \|x_n\|)$ is defined as a fix point satisfying the defining equations. In general, a set of equations may have none, one, or many solutions.

A solution for a recursive definition can be calculated in the following iterative way:

- $\vec{S}_0 = (\|x_1\|_0, \ldots, \|x_n\|_0) \stackrel{\texttt{def}}{=} (\emptyset, \ldots, \emptyset)$
- $\vec{S}_i = (\|x_1\|_i, \ldots, \|x_n\|_i) \stackrel{\texttt{def}}{=} \left(q_0(\vec{S}_{i-1}/\vec{x}), \ldots, q_n(\vec{S}_{i-1}/\vec{x})\right)$
- $\vec{S} = (\|x_1\|, \ldots, \|x_n\|)$, where $\|x_i\| \stackrel{\texttt{def}}{=} \{(m, n, a, b) \mid \exists k \forall j \geq k \ (m, n, a, b) \in \|x_i\|_j\}$, for $i \in [1, n]$.

4 Concrete semantics

In the previous section we defined the abstract semantics of Kernel, which makes possible reasoning about Kernel programs. In order to turn a Kernel specification into a program which can be executed, we refine the abstract semantics into *concrete semantics*, where encoding details of flat values and Kernel functions are described.

4.1 Environment events separation

In order to elaborate the concrete semantics, we separate the synchronization events $\mathcal{E}$ into two groups: input events $\mathcal{I}$ and output events $\mathcal{O}$, in such a way that $\mathcal{E} = \mathcal{I} \times \mathcal{O}$. Intuitively, input events consist of such events as reading some bits from the input packet or querying a lookup table, and output events consist of writing to an external (write-only) devices.

4.2 Flat value encoding

Given a type graph, we consider a very natural encoding of values of types which the graph represents, by choosing an encoding alphabet Σ, e.g., $\Sigma = \{0, 1\}$, and labeling all outgoing edges of each *or* node by strings over the encoding alphabet in such a way that no two different edges outcoming of the same node are labeled by words which are in the prefix relation.

DFS of the hyper-path of a value defines its encoding...

4.3 Graph representation of the semantics of a Kernel expression

Let $\mathcal{E} = \mathcal{I} \times \mathcal{O}$ and Σ be the environment monoid and the flat value encoding alphabet, respectively. We define *semantics graph* $G = (V, E, s_0, s_f, F, \lambda)$ over $\mathcal{M} = \Sigma^* \times \mathcal{E} \times \Sigma^*$, as a digraph (V, E), with one starting vertex $s_o \in V$, one ending vertex $s_f \in V$ without outgoing edges, a set of final vertexes $F \subset V$, and a edge labeling $\lambda : E \mapsto \mathcal{M}$.

Intuitively, vertices of the graph represent states and edges represent transitions. An edge t from vertex s_1 to s_2 and labeled by $(w_i, e, w_o) \in \mathcal{M}$ tells us that the modeled system with argument value encoded by w_i, when in state s_1 can move to the state s_2 producing value w_o and interacting with the environment by e.

More precisely, every path π from s_0 to s_f passing by a final vertex defines a quadruple (m, n, a, b), where $\lambda(\pi) = (w_a, mn, w_b)$ and $\lambda(\pi') = (w'_a, m, w'_b)$, where π' is the longest initial part of π ending in a final vertex, w_a is the encoding of a and w_b is the encoding of b.

4.3.1 Composition

Given two finite semantics graphs G_1 and G_2 for expressions $(T_1 \to T_2)E_1$ and $(T_2 \to T_3)E_2$, we can construct a new finite semantics graph $G = G_1 \circ G_2$ which represents the functional composition $(T_1 \to T_3)E_1 :: E_2$.

Without loss of generality we assume that all edge labels of G_2 are of a form (w_a, m, w_b) with w_a being empty word or a letter, i.e., $w_a \in \Sigma \cup \{\varepsilon\}$. Therefore, a vertex A in G_2 and a word $w \in \Sigma^*$ defines a set of vertexes $G_2(A).w$ of G_2 which are reachable from A by a path whose labels w_a generate w. i.e., if $B \in G_2(A).w$ then there exists a path π in G_2 starting in A and such that $\pi) = (w, m, w_b)$ for some $m \in \mathcal{E}$ and $w_b \in \Sigma^*$.

The composition of G_1 and G_2 is obtained by a Cartesian product of vertixes of G_1 and G_2, i.e., vertexes of G as pairs $(A, B) \in (V_1 \times V_2)$.

4.3.2 Projection

4.3.3 Structure

4.3.4 Union

4.3.5 Functional abstraction

—

In general,

We require that such a graph be *prefix-deterministic*, i.e., for every two different edges e_1 and e_2 originating from the same vertex u, we have $\overline{\lambda(e_1)} \not\perp \overline{\lambda(e_2)}$.

Let $\mathcal{M} = M \times \Sigma^* \times \Sigma^*$, where M is an environment monoid and Σ is a finite alphabet used for encoding flat values. Notice, that $M \times \Sigma^*$ and $\mathcal{M}$ are also environment monoids. Intuitively, an element $(m, a, b) \in \mathcal{M}$ represents a step of calculation parametrized by a, yielding b, and interacting with environment via m. By $\overline{(m, a, b)}$ we denote (m, a), i.e., projection onto two first coordinates.

We define *semantics graph* $G = (V, E, i, s, F, \lambda)$ over $\mathcal{M}$, as a digraph (V, E), with one starting vertex $i \in V$, one ending vertex $s \in V$ without outgoing edges, a set of finte vertexes $F \subset V$, and a edge labeling $\lambda : E \mapsto \mathcal{M}$. We require that such a graph be *prefix-deterministic*, i.e., for every two different edges e_1 and e_2 originating from the same vertex u, we have $\overline{\lambda(e_1)} \not\perp \overline{\lambda(e_2)}$.

4.4 FIFO represenation of Kernel expressions

Kernel program (without recursive constructs) can effectively be translated into a network of transducers connected by FIFO channels. In this section we describe a way of constructing the FIFO system.

Transducers (or more generally state based computing) are used to encode *functions* by *behavior*. We describe a systematics method of encoding a Kernel program which contains functional and behavioral description, into a transducer. In our presentation we assume that the set of *events* is finite and well defined. Moreover, for the sake of this paper, the set of events is divided into two disjoint classes:

- Read events: `getZero`, `getOne`, `getEof`
- Write events: `putZero`, `putOne`, `putEof`

Alternatively, read events can be seen as elements of an input alphabet and write events as elements of an output alphabet. For the sake of simplicity we assume only the two FIFOs: I and O.

In order to implement all Kernel constructs we designed the generic component interface for every Kernel sub-expression, see Figure 1. The interface consists in:

- A — input FIFO through which the actual value of the argument is provided;
- R — output FIFO through which the result (R) is communicated to the environment;

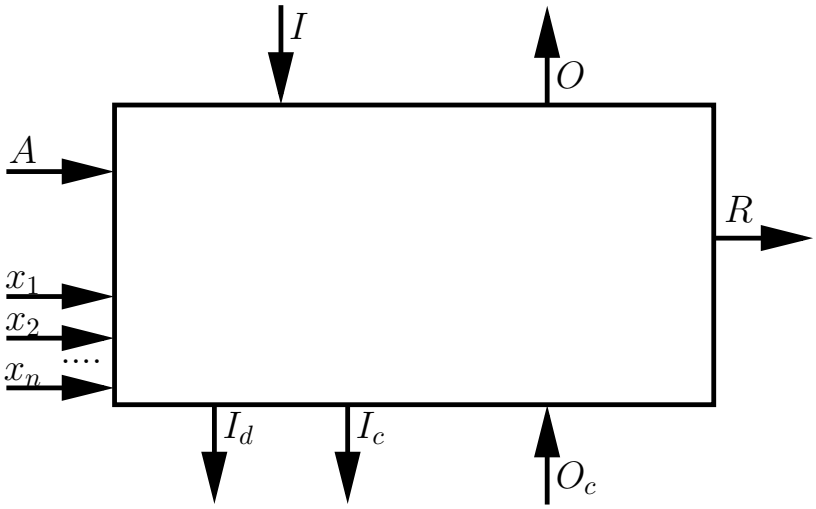


Figure 1: Kernel generic component interface.

- I, O — input and output FIFOs for environment monoid (in this presentation we limit ourselves to the one input and one output stream);
- I_d, I_c — output FIFOS which splits the stream I into what was effectively consumed (done), and the rest (continuation);
- O_c — input FIFO for "output continuation";
- $x_1, x_2, \ldots, x_n$ — free variable FIFOS.

4.4.1 Free variable references

Reference to a field or an argument is just a FIFO system as depicted in Figure 2. The "eof" box in the picture denotes a system which generates a single "end-of-file" signal.

4.4.2 Abstraction

```
($x P -> f)
```

The pattern P is considered as a filter over domain values of the relation. The filter can be implemented by a CSM (i.e., using stack).

A stack can be implemented by a single state machine with one self-looping FIFO channel. The interface of a stack consists of a input channel caring the alphabet of the stack plus special symbol `pop`, and an output channel caring the alphabet of the stack plus special symbol `empty`. In Kernel it would be:

```
<type> stack_alphabet <is> [ ... ];
<type> request <is> ['push stack_alphabet, 'pop ];
<type> answer <is> ['pop stack_alphabet, 'empty ];
```

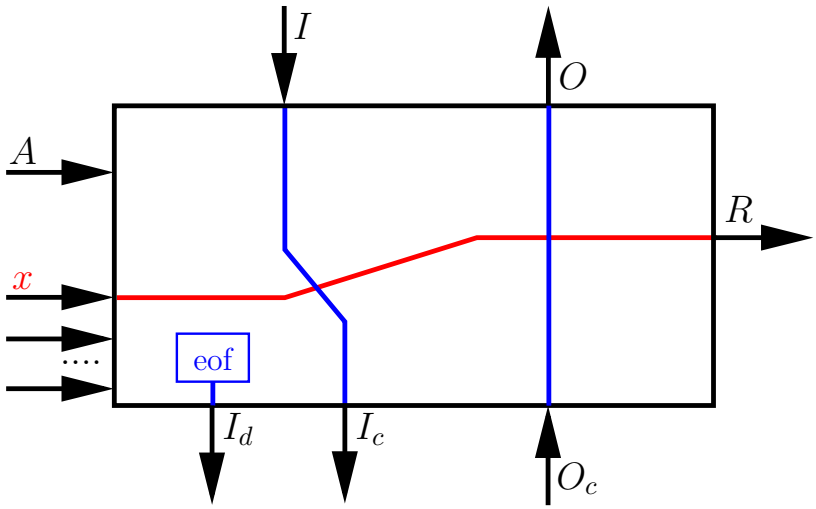


Figure 2: FIFO element for a reference: $\$a$ or $@a$.

Therefore, a filter for every pattern can be implemented. The overall module is depicted in Figure 3.

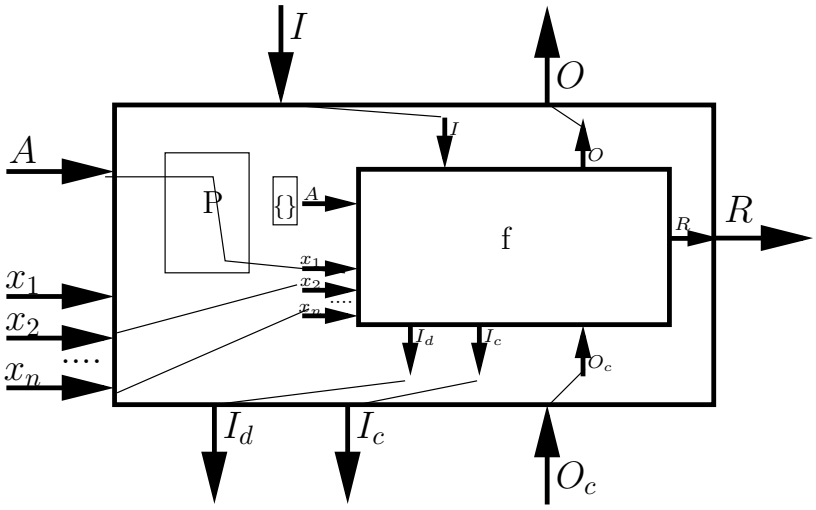


Figure 3: Abstraction

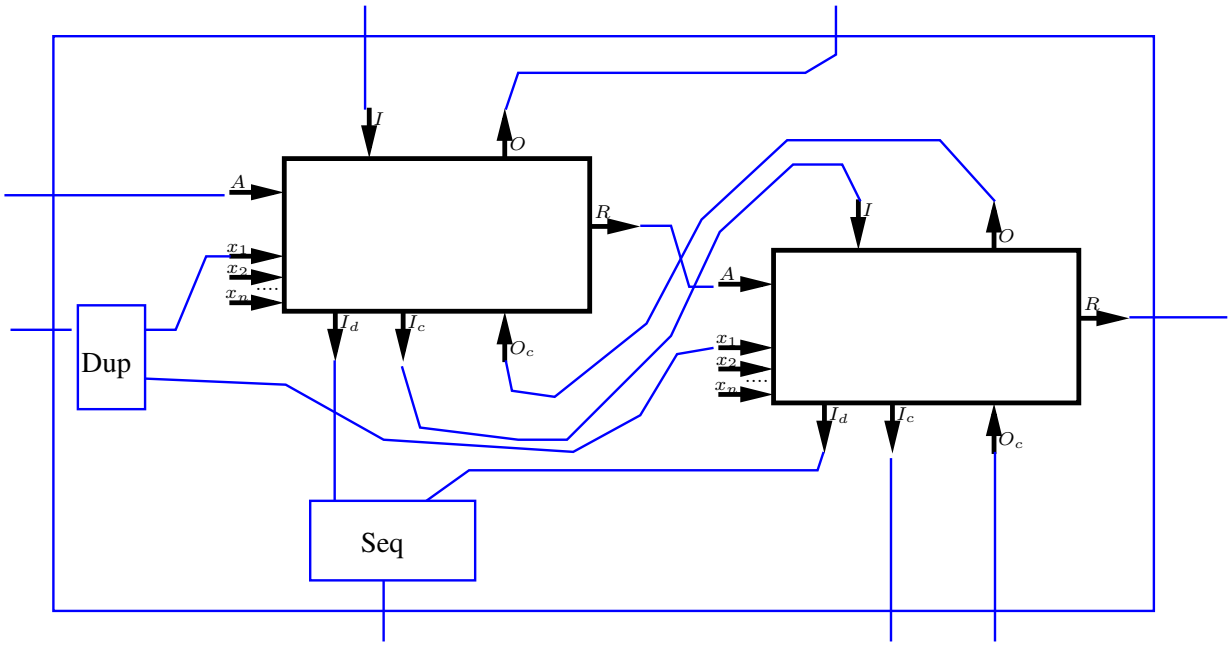


Figure 4: Composition

4.4.3 Composition

4.4.4 Product

4.4.5 Union

4.4.6 Variant constructor

4.4.7 Projection

4.5 Bottom-up transducer construction

The write and read events are represented as built-in unit type objects. The semantics of an event is a two-state-one-transition transducer, whose transition is labaled by the event. Alternatively, a read event is represented as the two-state-one-transition transducer, and a write event is represented as a one-state-no-transition transducer with initial output being the write event. The name of an event will be used to denote the corresponding transduser.

4.6 Value encoding

Type graph is an edge labeled *and/or* graph. Types correspond to vertices: product types are presented by *and* vertices, union nodes by *or* vertices, and the unit type is an *and* vertex without children.

We assume that we are given the type graph induced by the program. A construction of the minimal type graph is relatively easy. Given an alphabet Σ, we propose the natural

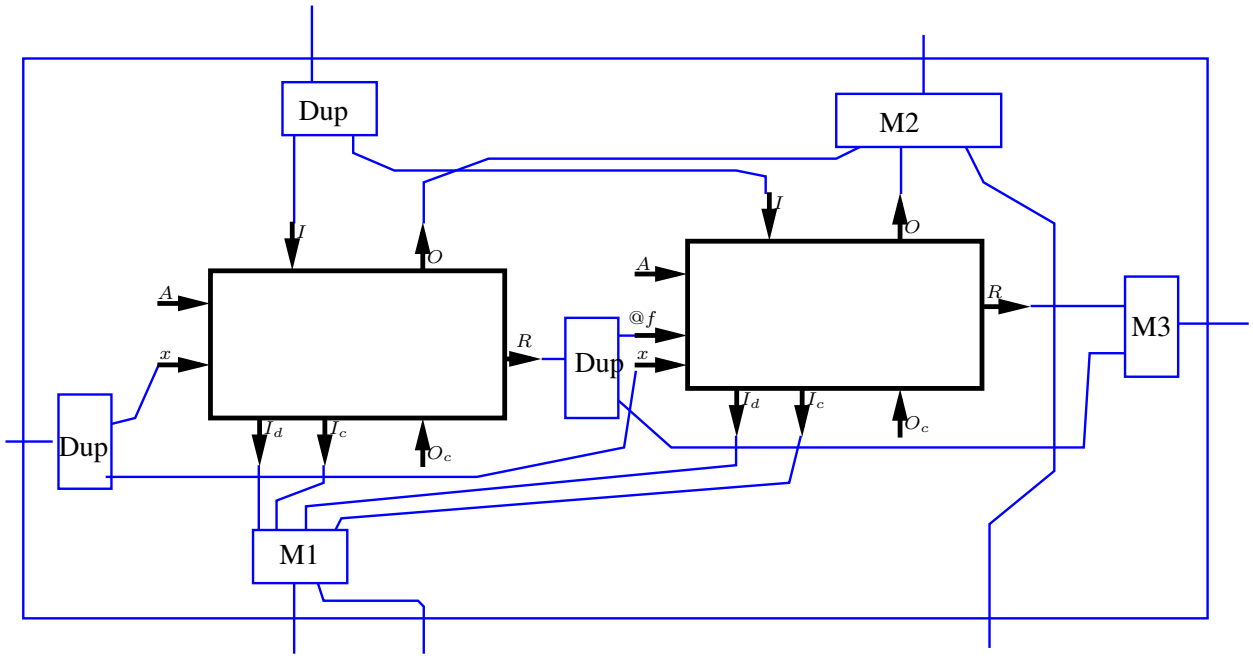


Figure 5: Product by intersection

encoding of values of a given type by words over Σ. For a given type, the set of all its values is a prefix code.

If t is a type, by $\texttt{id}(t)$ and $\texttt{skip}(t)$ we denote the identity transducer (or CSM) on all values of t, and the automaton accepting all values of t, respectively. If f is a field name of type t, then $\texttt{proj}(t, f)$ denotes the transducer taking any value of type t and returning, if defined, the value of the field f. These transducers (CSMs) can be easily constructed from the type graph.

4.7 Denotational semantics of subexpressions by annotations

With every subexpression p we associate a pair $\|p\| = (\pi, T)$, called annotation of p, where π is a list of free references (with types) in p, and T is a transducer (or CSM). If p is a closed subexpression, then π is empty word, and T is a transducer implementing p. If p contains some free occurences of references, then π represents thier order, possibly with repetitions, which they should be supplied to the transducer T in order to be instantiated. Notice that an open subexpression can have many different annotations, however a closed one (if the transducer is normalized) will only have one.

The main difficulty in constracting a transducer semantics of a Kernel subexpression is the treatement of free references. In order to overcome the problem we give ourselves *replicator* transducers. We will write $\texttt{repl}(\pi_1, \pi_2)$, where $\pi_1 = t_1 t_2 \dots t_n$ is a list of types, and $\pi_2 = i_1 i_2 \dots i_k$ is the list of indexes, $i_j \in [1, n]$ for $j \in [1, k]$, to denote the transducer accepting any input of type π_1 and transforming it into a (sub-) list of the same values potentially with

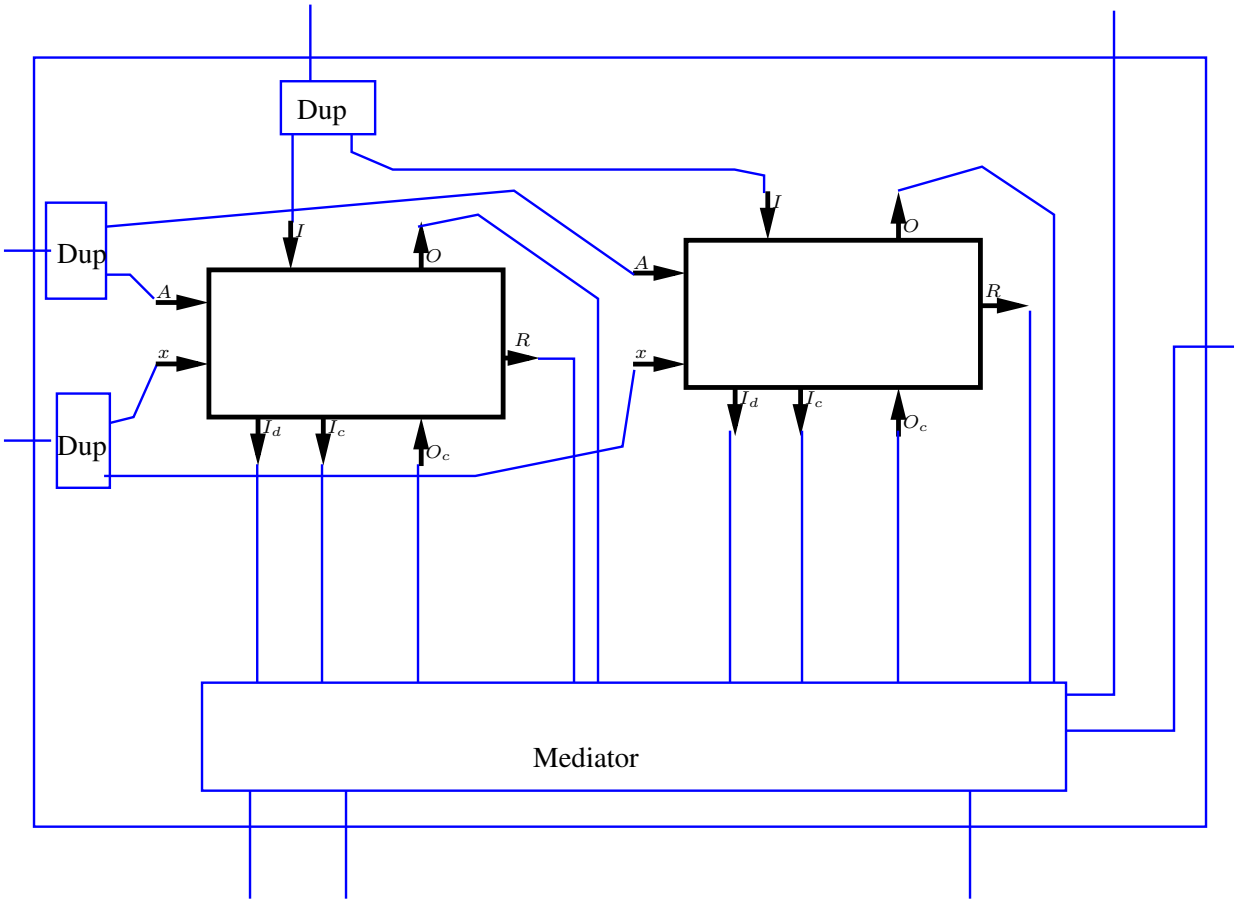


Figure 6: Union

duplications and reorderings, which is defined by π_2. For example, $\texttt{repl}((t),(1))$ corresponds to $\texttt{id}(t)$, and $\texttt{repl}((t),(1,1))$ corresponds to the transducer accepting any value v of type t and producing vv.

identifier: $\|\$x\| \stackrel{\texttt{def}}{=} (\$x, \texttt{id}(t))$, where t is the type of $\$x$.

variant: $\|f\ q\| \stackrel{\texttt{def}}{=} (\pi, \texttt{code}(f : t) \cdot T)$, where $\|q\| = (\pi, T)$, t is the image type of $f\ q$, and $\texttt{code}(f : t)$ is the constant transducer producing the encoding for field f of type t.

structure: $\|\{f_1\ q_1, f_2\ q_2\}\| \stackrel{\texttt{def}}{=} (\pi_1\pi_2, T_1 \cdot T_2)$, where $\|q_1\| = (\pi_1, T_1)$ and $\|q_2\| = (\pi_2, T_2)$.

The evaluation order of elements of a structure is given by the order of occurrences (from left to right) of sub-expressions. Thus, in general, we should write:

$$\|\{f_{i_1}\ q_{i_1}, \ldots, f_{i_n}\ q_{i_n}\}\| \stackrel{\texttt{def}}{=} (\pi_{i_1} \cdots \pi_{i_n}, (T_{i_1} \cdots T_{i_n}) :: \texttt{repl}((t_{i_1} \ldots t_{i_n}), (1, \ldots, n))),$$

where $i_1, \ldots, i_n$ is a permutation of the encoding order $1, 2, \ldots, n$ of the product type $\{t_1, \ldots, t_n\}$.

projection: $\|q.f\| \stackrel{\texttt{def}}{=} (\pi, T :: \texttt{proj}(t, f))$, where t is the image type of q and $\|q\| = (\pi, T)$.

union: $\|[q_1, q_2]\| \stackrel{\texttt{def}}{=} (\pi_1\pi_2, T_1 \cup T_2$, where $\|q_1\| = (\pi_1, T_1)$, $\|q_2\| = (\pi_2, T_2)$, and $T_1 \cup T_2$ denotes the priority merge of classifiers.

composition: $\|q_1 :: q_2\| \stackrel{\texttt{def}}{=} (\pi_2\pi_1, (\texttt{id}(\pi_2) \cdot T_1) :: T_2)$, where $\|q_1\| = (\pi_1, T_1)$ and $\|q_2\| = (\pi_2, T_2)$.

abstraction: $\|(\$x\, P \mapsto q)\| \stackrel{\texttt{def}}{=} (\pi', \texttt{repl}(\pi'\, x, \omega) :: (\texttt{id}(\pi) \cdot \texttt{pattern}(P)) :: T)$, where $\|q\| = (\pi, T)$, pattern P is of type t, and π' and ω are such that x is not in π' and $\texttt{repl}(\pi'\, x, \omega)$ corresponds to π, e.g., if $\pi = yxxyz$ then $\pi' = yz$ and $\omega = (1, 3, 3, 1, 2)$ are possible values.

5 Transducer operations with respect to side effects

5.1 Product

We have two cases for synchronization:

- concatenation $T_1; T_2$ — every synchronization event of T_1 precedes every event of T_2
- intersection $T_1 \cdot T_2$ — an event in T_1 can occur only if the same (unified?) event occurs in T_2, or T_2 already finished (and vice-versa).

5.2 Union

5.3 Pipeline

As for concatenation product, in $T_1 :: T_2$ every synchronization event of T_1 precedes every event of T_2.