
by

G.A. Bussing
K.M. van Hee
M. Voorhoeve

86/02

September 1986
1. Introduction

There exist many programming languages. Each language has its own style of program design. The way one thinks during the design process of a program is affected by the choice of the programming language. There are at least three types of languages. A program designed in a procedural language consists of a sequence of steps which will be evaluated in a certain order. Such steps are usually called statements (see [10]). In a relational language, sometimes called a logic programming language, one defines relations. These relations form a set of axioms. Evaluation of a new relation is trying to prove that this relation holds with respect to the set of axioms, by binding all variables that occur in that relation to values (see [5]). A functional language is based on definitions (see [2,3,6,7,8,9]). A definition is the assignment of a name to an expression. The defined names can be used within other expressions as abbreviations. Evaluation of a functional program is a process of reduction and evaluation of expressions. A functional programmer is in the first place not concerned with efficiency of his program, but mainly with correctness of his definitions. We apply here the principle of separation of concerns.

The main goal of our research is the development of a functional language, called ELISA, to support a designer in making a correct specification of an information system (IS). An IS can be considered as an abstract machine which fulfills tasks for its environment. Such a machine consists of three parts. The state of the machine is represented by a variable. This variable is called the database variable or the state variable. The state variable is bound to a value and ranges over a state space that is determined by its type. The machine has the capability of the execution of transactions. Execution of a transaction may result in a transformation of the database state. The machine receives impulses from its environment which serve as triggers for the execution of a transaction. Each impulse consists of a command name and input data for the transaction. The command name specifies the transaction. The input data for a transaction from the environment and the output data from a transaction destined to the environment are represented by the input variable and the output variable. These variables may have complex structures. A system to bind and view these variables is called the dialog.
system. The dialog system realizes the communication between the machine and its environment. We don't consider the dialog system. Each transaction can be specified by a function which domain is the Cartesian product of the state space of the input variable and the database state space. The range of such a function is a subset of the Cartesian product of the state space of the output variable and the database state space. The machine is activated when it receives a command name. By lazy evaluation it binds the input variable. This implies that the input variable isn't specified completely. Then the machine evaluates the function that is specified by the received command name with the current values of the input variable and the state variable as arguments. After the computations the machine binds the state variable and the output variable to new values which are the results of these computations. Then the machine becomes again passive. Thus the process of the machine is the infinite cycle of being passive, reading, computing, writing and being passive again. The design of an IS is the design of the dialog system, the declaration of the variables and the specification of the transactions. ELISA is developed to support the last two design goals. The variables and transactions are implemented as an ELISA program.

ELISA is a strongly typed functional programming language (see [1,4,9]). We introduce types to:

1: prevent a class of syntax errors which would cause an interrupt during the runtime of an ELISA program.

These faults are mainly function applications of a wrong type. e.g. the expression 19+{86} makes no sense.

2: ease implementation like the reservation of memory space.

3: specify the structure of the variables of an information system and define herewith their state space.

Lambda expressions are used to define functions. After choosing basic objects and their types, one builds with these objects new objects like lambda expressions. The specification of a transaction is described using these constructed objects.

A sequence of definitions written in ELISA is called an ELISA script or an ELISA program. ELISA is a flexible language because it has facilities to change definitions and add new definitions to an existing ELISA script in an easy way. Many data models, that are developed for databases, e.g. the relational data model or the functional data model, can be described within ELISA. If we build an interpreter for ELISA scripts, we are able to generate prototypes of our IS and test them in its
environment. Basic objects can be used in many ways. They may be implemented by a procedural program. A programmer may replace objects defined in an ELISA script by basic objects with a procedural program and test these basic objects using the ELISA interpreter. Now he is able to transform stepwise an ELISA program, i.e. a specification, into a probably more efficient procedural program. So we may use ELISA as a tool for the development of procedural programs. ELISA also embodies the method of stepwise refinement. A programmer may postpone the definition of an object by defining it temporary as a typed basic object, thus without a concrete definition in ELISA. Then he may use the name of this basic object in the definition of other objects. Later on he replaces this basic object by a definition in ELISA or gives for this basic object a procedural program.

For special application fields we can produce ElISA scripts with basic and defined objects that are useful for the design of a specific IS for that field. These defined objects are called components.

In the following chapters we will give a formal definition of ELISA. We do realize that this work needs improvement. Especially in the type system, not all the problems we encountered have been solved to our complete satisfaction. Suggestions for improvement are always welcome.
2. The definition of the language

2.1. A short overview

In this section we will give an introduction to ELISA. An ELISA script consists of a sequence of definitions. The order of the definitions is of no significance. Thus one may say that an ELISA script consists of a set of definitions. A definition is the assignment of a name to a language object. This name may occur in the definition of other language objects. A definition may come after its first use. We distinguish four kinds of definitions: object definitions, type definitions, command definitions and declarations of variables.

New objects can be constructed from basic objects. Reals and strings are basic objects. A basic object (BO) is an object with a name and a type without an ELISA definition. The name of a real or a string is the usual one, e.g. 1 or 'elisa'. With the exception of reals and strings all used BOs have to be declared in the corresponding ELISA script. New objects can be defined explicitly by enumeration or implicitly by an expression. Lambda expressions are objects in which parameters may occur. Parameters range over a domain that can be described in the contextlist of lambda expressions. We distinguish in ELISA typed and untyped lambda expressions. Expressions are objects that arise from function application on lambda expressions. Recursively defined lambda expressions are permitted in ELISA. This means that the name of a lambda expression can be used in the definition of the object belonging to that lambda expression. It is the responsibility of the programmer, that the number of steps of the evaluation of an expression using recursively defined lambda expressions is finite.

Every object is related to a type. This type relation restricts the set of permitted, i.e. correct, objects. Types are defined hierarchically by type constructors like List, Set, Row, Fun, Efun or Tuple.

\[ \text{e.g. Row(\text{Int}, \text{List(Real)}, \text{Fun(\text{Nat}, \text{Set(Bool)}))}). \]

We say the type of an object is the name of the type constructor at the highest level of hierarchy. For example lambda expressions are typed as functions or enumerable functions. Analogously to the definition of objects types can be given a name used as an abbreviation. The semantics of types is
determined by the type axioms and the B0s that are defined on them.
The specifications of transactions is described in an ELISA script by commands. The current values of the state variable and the input variable are used as arguments for the evaluation of a command. After computations the state variable and the output variable are bound to new values. Variables are declared in the ELISA script. Each variable has a name and a type. All objects belonging to the type of a variable form the state space of that variable. A command definition consists of a sequence of argument variables, a sequence of result variables and a name of a lambda expression that specifies the effect of the transaction.

In this chapter we give the syntax of ELISA and the axioms of the type system. An ELISA script is correct if it satisfies the axioms of the type system.
2.2 Basic notions

2.2.1 The metalanguage

The metalanguage is the language we use to define the syntax of ELISA and the corresponding type system. We use the usual mathematical notations of logic and set theory including the union operator (U), the universal quantor (\(A\)), the existential quantor (\(E\)), the implication (\(\rightarrow\)), the "if and only if" operator (\(\leftrightarrow\)) and the and, or and not operators (\(\land\), \(\lor\), \(\neg\)).

We will denote the set of the natural numbers by \(N\) and the set of the integers by \(Z\). We also take the convention that \(0\in N\).

Further we define for each \(n,m\in Z: n..m := \{i\in Z | n\leq i \leq m\}\).

The set of mappings from set \(A\) to set \(B\) is denoted by \([A\rightarrow B]\). The domain and the range of a function \(f\) are denoted by \(\text{dom}(f)\) and \(\text{rng}(f)\). We will define functions often recursively.

Note: \((f\in [A\rightarrow B]) \Rightarrow \text{dom}(f)=A\) and \(\text{rng}(f)\subseteq B\).

Sequences:
An alphabet is a non empty set \(A\). We define the set of all finite \(A\)-sequences by:\n\(A^* := (\bigcup_{k\in N} : [1..k\rightarrow A])\)

The empty sequence \((k=0)\) is denoted by \(\varepsilon\).

Note: \(A\subseteq B \Rightarrow A^* \subseteq B^*\)

We will define some operators on sequences:
Length:
If \(f\) is an \(A\)-sequence and \(\text{dom}(f)=1..k\), \(k\in N\) then we define the length of \(f\) by:\n\(\text{num}(f) := k\)

First:
If \(f\) is an \(A\)-sequence and \(n\) a natural so that \(n\in\mathbb{N}\) then we define the \(A\)-sequence \(\text{first}(f,n) (\in [1..\text{num}(f)\rightarrow A])\) by:\n\(\text{first}(f,n)(i) := f(i) \quad \text{if } i\in 1..n\)

Note: \(A\subseteq \mathbb{N}\): \(\text{first}(f,0)=\varepsilon\) and \(\text{first}(f,\text{num}(f))=f\)

Rest:
If \(f\) is an \(A\)-sequence and \(n\in\mathbb{N}\) then we define the \(A\)-sequence \(\text{rest}(f,n) (\in [1..(\text{num}(f)-n)\rightarrow A])\) by:\n\(\text{rest}(f,n)(i) := f(i+n) \quad \text{if } i\in 1..(\text{num}(f)-n)\)
Concatenation:
If $f$ and $g$ are A-sequences then we define the A-sequence $f \& g$ 
$(\in [1..(\text{num}(f)+\text{num}(g))\rightarrow A])$ by:

$$(f \& g)(i) := f(i) \quad \text{if } i \in 1..\text{num}(f)$$

$$= g(i-\text{num}(f)) \quad \text{if } i \in (1+\text{num}(f))..(\text{num}(f)+\text{num}(g))$$

L:
if $f$ is a A-sequence of length 1 then we define:

$L(f) := f(1)$

(Note 1: We will call $f \& g$ the concatenation of $f$ and $g$.
2: $\forall f \in A^* : \text{eps} \& f = f \& \text{eps} = f$.
3: $\forall f,g,h \in A : (f \& g) \& h = f \& (g \& h)$.
4: Whenever there exists no fear of ambiguity we will denote a A-sequence of length 1 by its image of 1 (i.e. $L(f)$).
And furthermore we will write $fg$ instead of $f \& g$, $f,g \in A^*$.

2.2.2 The first definitions

To specify ELISA we will consider the alphabet of ascii symbols. The set of ascii symbols is denoted by ascii. A special subset of ascii is:

$H := \{a,b,...,z,A,B,...,Z,1,2,3,4,5,6,7,8,9,0\}$.

Now we define the set $S$ by:

$S := \text{ascii}^*$.

ascii-sequences (elements of $S$) are called strings.

Firstly we introduce a group of subsets of $S$:

$Nr := \{x \in \{1,2,3,4,5,6,7,8,9\}^\times \{\text{eps}\}|L(\text{first}(x,1)) \neq 0\},$

$Zr := \{s \& x|x \in \{\text{eps},-\} \text{ and } x \in Nr\},$

$Rr := \{x \& d \& y$

$| x \in Zr \text{ and } d \in \{\text{eps},\}\text{ and } (p=.) \Rightarrow y \in Nr \text{ and } (p=\text{eps} \Rightarrow y=\text{eps})$

$\}$,

$Br := \{\text{true},\text{false}\},$

$Words := \{xH^\times \{\text{eps}\}\}$

$\mid L(\text{first}(x,1)) \in \{a,b,...,z,A,B,...,Z\}$

$\}$

$\{\text{B0,Int,Bool,Real,String,Fun,Efun,Set,Row,List,Tuple,}$

$\text{true},\text{false},\text{type},\text{ivar},\text{ovar},\text{svar}$

$\}$,

$\tilde{W} := \{ \& x \& \tilde{x}|x \in \text{Words}\}$,
"W" := {" & x & " | x ∈ Words}.

Note 1: NrcZr, ZrcRr.

2: Words (elements of Words) are strings consisting only of symbols of H. We may split strings (∈ S) into words by using symbols that don't belong to H.
E.g. The string [elisa][elisa]f(n) consists of four words and the word elisa doesn't occur as a complete word in the string [elisascript].

Operators on S:

Counter:

Ax∈ S : Axcascii :

The counter determines the number of symbols of x preceding the first occurrence of a symbol that doesn't belong to X, i.e. num(x) when x has no symbols of X.

\[
\begin{align*}
count(x,X) &:= 0 & \text{if } L(first(x,l)) \notin X \text{ or } x = \text{eps} \\
&= \text{count}(\text{rest}(x,l),X)+1 & \text{if } L(first(x,l)) \notin X \text{ and } x \neq \text{eps}
\end{align*}
\]

Example: \(\text{count}(\text{elisa/script},H)=5\)

Substitution:

Axcascii : Axc* : Ay∈ S : Axc∈ S :

The substitutor replaces every occurrence of a complete X-sequence x in string z by string y.

\[
\begin{align*}
\text{sub}(X,x,y,z) &:= \text{eps} \text{ if } z = \text{eps} \\
&= \text{first}(z,l) \& \text{sub}(X,x,y,\text{rest}(z,l)) & \text{if } z \neq \text{eps} \text{ and } L(first(z,l)) \notin X \\
&\quad \text{if } y \& \text{sub}(X,x,y,\text{rest}(z,\text{count}(z,X))) & \text{if } z \neq \text{eps} \text{ and } L(first(z,l)) \notin X \\
&\quad \text{if } x = \text{first}(z,\text{count}(z,X)) & \text{and } x \neq \text{first}(z,\text{count}(z,X)) \\
&\quad \text{first}(z,\text{count}(z,X)) \& \\
&\quad \text{sub}(X,x,y,\text{rest}(z,\text{count}(z,X))) & \text{if } z \neq \text{eps} \text{ and } L(first(z,l)) \notin X \\
&\quad \text{and } x \neq \text{first}(z,\text{count}(z,X))
\end{align*}
\]

Example: \(\text{sub}(H,\text{program},\text{script},[\text{elisaprogram}][\text{elisa-program}]) = [\text{elisaprogram}][\text{elisa-script}]\)

Note: Often we will write sub(x,y,z) instead of sub(H,x,y,z), x ∈ H*, y,z ∈ S
**Member checker**

\[ \text{AXascii} : A \times \text{EX}^* : A \times \text{ES} : \]

The member_checker verifies the complete occurrence of the A-sequence x in string y.

\[
\text{member}(X,x,y) := \text{false} \quad \text{if} \quad y = \text{eps}
\]

\[
\text{member}(X,x,\text{rest}(y,1))
\]

\[
\text{true} \quad \text{if} \quad y \neq \text{eps} \quad \text{and} \quad L(\text{first}(y,1)) \in \text{EX}
\]

\[
\text{member}(X,x,\text{rest}(y,\text{count}(y,X)))
\]

\[
\text{true} \quad \text{if} \quad y \neq \text{eps} \quad \text{and} \quad L(\text{first}(y,1)) \in \text{EX}
\]

\[
\text{and} \quad x \neq \text{first}(y,\text{count}(y,X))
\]

Example: \( \text{member}(H,\text{script},[\text{elisascript}][\text{program}]) = \text{false} \)

**Note:** Often we will write \( \text{member}(x,y) \) instead of \( \text{member}(H,x,y) \), \( x \in \text{H}^* \), \( y \in \text{ES} \)

**BNF notations:**

BNF (Backus Naur Form) notations are used to describe the syntax of an ELISA script. We take the convention to underline non terminals. We introduce a new class of production rules. A rule of the form \( x \in X \), where \( x \in \text{S} \) and \( X \subseteq \text{S} \) must be interpreted as:

\( x \) can be replaced by any element belonging to \( X \).

We can consider a set of BNF rules as a set of predicates specifying subsets of \( \text{S} \). Every production rule defines a set of strings. We will identify this set by the bold non terminal at the beginning of that production rule on the left of the ::= or \( \in \) symbol.

Example: \( \text{cp} ::= \text{eps} | ( \text{cp} ) \text{cp} \),

\( \text{cp} \) is the set of all correctly nested parentheses.

**Note:** We use bold identifiers of sets and some bold special symbols like the universal quantor to distinguish them from strings.

**Some special subsets of S:**

names, bo_names, parameters, variables, new_types and command_names are arbitrary, mutually disjunct, subsets of Words.
2.3 Types and objects

Types and objects are the basic tools for the design of an ELISA script. In this section we will give their syntax. Herefore we introduce the sets set of types and the set of objects. types and objects are subsets of $S$. Types and objects are related by the predicate type, defined on the Cartesian product of objects and types. Thus the type relation is determined by the set:

$\{<x,y>|x \in \text{objects and } y \in \text{types and type}(x,y)\}$.

The set of objects belonging to a type is defined by:

$\text{As} \in \text{types} : \text{SO}(s) := \{t \in \text{objects}|\text{type}(t,s)\}$

With the help of axioms on the predicate type, we will define in section 2.6 the partial correctness of an ELISA script. These axioms restrict the set of correct objects and types. Herewith they define indirectly a part of the semantics of ELISA scripts.

2.3.1 Types
We distinguish:
1: basic types like Nat, Int, Real, String and Bool.
2: type constructors like Set, List, Row, Tuple, Efun and Fun.
3: new types which are names used as synonyms for other types.
4: type variables which range over all types.
5: enumerated types which elements are typed with enumerated sets defining that type.
6: type projections which define types indirectly.

Example: Efun(*,Row(Sentence,\{|"a","b"\})) is a type, where * is a type variable, Sentence is a synonym for List(String) and \{|"a","b"\} is an enumerated type.

Note: We call elements of "W" indices.

The set types is determined by:

```
types ::= basic types|type variables|enumerated types|new types| Fun( types , types )| Efun( types , types )| Set( types )| Row( type list )|Row()| Tuple( tuple list )|Tuple()| List( types )| types.index|
```
basic types ::= Nat | Int | Real | String | Bool

type list ::= types | types, type list

tuple list ::= index : types | index : types, tuple list

type variables ::= * | type variables

new types ∈ new_types

enumerated types ::= { | index list |}

index list ::= index | index, index list

index ∈ "W"

Note 1: \(s_1\) is called the domain type and \(s_2\) is called the range type of \(Fun(s_1, s_2)\) or \(Efun(s_1, s_2)\), \(s_1, s_2 \in \text{types}\).

2: The option "types.index" is called type projection.

Type projection
Not all type projections are permitted. The type on the left of the dot in a type projection has to be a tuple or the equivalent of a tuple. The index on the right of the dot has to be an index that occurs in the tuple indicated on the left hand side of the dot. Therefore we start with an axiom that expresses this property and limits the set of types. This axiom violates partly our agreement in section 2.2 concerning the BNF notations:

Axiom 1

\( As \in \text{types} : A i \in "W" : As_1 \in \text{types} : \)

\( s.i \) occurs in \( s_1 \)

\( \Rightarrow \)

\( (En \in 0 \cdot E s_1, \ldots, s_n \in \text{types} : E i_1, \ldots, i_n \in "W" : ) \)

\((As \in \text{objects} : \text{type}(t, s) \Leftrightarrow \text{type}(t, \text{Tuple}(i_1:s_1, \ldots, i_n:s_n))) (*) \)

and

\( i \in \{i_1, \ldots, i_n\} \)

\)

types is the biggest subset of the set of strings produced by the BNF rules for types such that axiom 1 holds.

Note 1: \(x\) occurs in \(y\), \(x, y \in S\), is formalized by:

\( \text{En} \cdot (\text{num}(y) - \text{num}(x)) : \)

\( x = \text{first}(\text{rest}(y, n), \text{num}(x)) \)

and

\( n > 0 \)

\( \Rightarrow \)
L(first(rest(y,n-1),1))\in\{a,...,z,A,...,Z,0,...,9\} \\
and \\
(a < (num(y)-num(x))) \\
\Rightarrow \\
L(first(rest(y,n+num(x)),1))\in\{a,...,z,A,...,Z,0,...,9\}.

Compare this with the definition of the member_checker.

3. The predicate subtype determines a partial order relation on the set of types.

Example: subtype(Nat,Int) and eqtype(Sentence,List(String)) where Sentence\in\textbf{new
types}

Projection can be considered as a type operator on tuples:

\textbf{Axiom2}

\forall n \in \mathbb{N}\setminus\{0\} : A_1,\ldots,i_n \in \mathbb{W} : A_1,\ldots,s_n,\text{\textit{e}types} : 
\text{eqtype}(s,Tuple(i_1:s_1,\ldots,i_n:s_n)) \\
\Rightarrow \\
(A_1,\ldots : \text{eqtype}(s,i_j,s_j))

Example: eqtype(Tuple("a":Int),"a",Int) and eqtype(Tuple("a":Int,"b":Real),Tuple("b":Real,"a":Int))

Note: An index followed by its value in a tuple is called a field. The order of the fields in a tuple is irrelevant. Try to prove this.
Type variables
The semantics of type variables are defined using an axiom.
Elements of types are types without the occurrence of type variables:
\[
\text{types}^\ast := \{ t \in \text{types} \mid \text{Av\textunderscore variables} : \text{not}(\text{member}(\{\ast\}, v, t)) \}
\]

A type variable ranges over the set types^\ast:
Axiom3
\[
\text{At\textunderscore objects} : \text{As} \in \text{types} :
\text{type}(t, s) \iff (\text{Av\textunderscore variables} : \text{As} \in \text{types}^\ast :
\text{type}(t, \text{sub}(s_1, v, s)) )
\]

Then it holds that a type variable ranges over the set types:
Theorem1
\[
\text{At\textunderscore objects} : \text{As} \in \text{types} :
\text{type}(t, s) \iff (\text{Av\textunderscore variables} : \text{As} \in \text{types} :
\text{type}(t, \text{sub}(s_1, v, s)) )
\]

Example: When \text{type}(\text{top}, \text{Fun}(\text{List}(\ast), \ast)), \text{top\textunderscore names},
then it holds that As \in \text{types} : \text{type}(\text{top}, \text{Fun}(\text{List}(s), s))
e.g. type(top, Fun(List(Int), Int))

Figure: The set of types
2.3.2 Objects:
In ELISA several kinds of objects occur: predetermined basic objects, enumerated objects, lambda expressions, expressions, indices and projections.

predetermined basic objects (PBOs):
The PBOs are elements of \( \mathbb{W}, \mathbb{N}, \mathbb{Z}, \mathbb{R} \) and \( \mathbb{B} \). Strings are written in ELISA between quotes (i.e. "") to distinguish them from other objects (especially from names).

Axiom 4
\[
\begin{align*}
\text{At} \in \mathbb{N} & : \text{type}(t, \mathbb{N}) \quad \text{and} \\
\text{At} \in \mathbb{Z} & : \text{type}(t, \mathbb{Z}) \quad \text{and} \\
\text{At} \in \mathbb{R} & : \text{type}(t, \mathbb{R}) \quad \text{and} \\
\text{At} \in \mathbb{W} & : \text{type}(t, \mathbb{W}) \quad \text{and} \\
\text{At} \in \mathbb{B} & : \text{type}(t, \mathbb{B}) \\
\end{align*}
\]

Sets, lists, rows and tuples (enumerated objects):
Enumeration is a constructor which transforms a list of objects into a set, list, or row and a list of fields, i.e. indexed objects, into a tuple. The empty set is denoted by \( \{\} \), the empty list by \([[]]\) and the empty row and tuple by \(<>\).

The syntax rules for enumeration are:
\[
\begin{align*}
eset & ::= \{\} \mid \{ \text{object list} \} \\
elist & ::= [[]] \mid [[ \text{object list} ]] \\
erow & ::= <> \mid < \text{object list} > \\
etuple & ::= <> \mid <> \mid < \text{indexed object list} > \\
\text{object list} & ::= \text{objects} \mid \text{objects},\text{object list} \\
\text{indexed object list} & ::= \text{index} : \text{objects} \mid \text{index} : \text{objects},\text{indexed object list}
\end{align*}
\]

The semantic differences between these first four are determined by the choice of B0s, that are defined over them, and the type axioms:

Axiom 5
\[
\begin{align*}
\text{An} \in \mathbb{N} : \text{At}_{1}, \ldots, \text{At}_{n} & : \text{objects} : \text{types} : \\
\left( \text{At} \in \mathbb{N} : \text{type}(t_{j},s) \right) & \iff \text{type}(\{t_{1}, \ldots, t_{n}\}, \text{Set}(s))
\end{align*}
\]

Example: type(\{1,2,1,2\},\text{Set}(\text{Int}))
Note: \( \text{Ats} : \text{types} : \text{type}(\{\}, \text{Set}(s)) \)
Axiom 6
An6N : At_1,\ldots, t_n e objects : A e types :
(A \in \{1,\ldots, n : \text{type}(t_j, s) \}) \iff \text{type}(|t_1, \ldots, t_n|, \text{List}(s))

Axiom 7
An6N : At_1,\ldots, t_n e objects : A e 1,\ldots, s_n e types :
(A \in \{1,\ldots, n : \text{type}(t_j, s_j) \}) \iff \text{type}(<t_1, \ldots, t_n>, \text{Row}(s_1, \ldots, s_n))

Example: \text{type}(<\{1\}, "elisa">, \text{Row}(\text{Set}(\text{Nat}), \text{String})) \text{ and }
\text{type}(<\{1\}, "elisa">, \text{Row}(\text{Set}(\text{Int}), \text{String})) \text{ and }
\text{type}(<>), \text{row}())

Axiom 8
An6N : A_1,\ldots, i_n e \mathbb{N} : At_1,\ldots, t_n e objects : A e 1,\ldots, s_n e types :
(A \in \{1,\ldots, n : \text{type}(t_j, s_j) \})
\iff \text{type}(<i_1 : t_1, \ldots, i_n : t_n>, \text{Tuple}(i_1 : s_1, \ldots, i_n : s_n))

Example: \text{type}(<"a" : \{1\}, "b" : "elisa">, \text{Tuple}("a" : \text{Set}(\text{Nat}), "b" : \text{String})) 
\text{ and } \text{type}(<\text{}>, \text{Tuple}())

Expressions:
We call objects that are typed as a Fun functions. Expressions
are functions applied to a row of arguments that is typed with
the domain type of that function. We use prefix notation. An
expression is an implicitly defined object. Evaluation of this
expression, based on execution of B0s and reduction, results in
this object.

Syntax rule for expressions:
expression ::= objects( object list )

An expression is correct when its arguments are correctly typed.
Expressions with one argument:
Axiom 9
A f, t e objects : A e 2 e types :
((Es_1 e types : \text{type}(t, s_1) \text{ and type}(f, \text{Fun}(s_1, s_2)))
\iff \text{type}(f(t), s_2)\)}
Example: When \( \text{type}(f, \text{Fun}(\text{Int}, \text{Fun}(\text{Int}, \text{Int}))) \), \( f \in \text{names} \), then \( \text{type}(f(1)(1), \text{Int}) \).

Expressions with more than one argument:

Axiom I

At \( n \in \mathbb{N} \setminus \{0\} \): \( \forall t_1, \ldots, t_n \in \text{objects} : \forall s \in \text{types} : \)

\[
\text{type}(f(t_1, \ldots, t_n), s) \iff \text{type}(f(<t_1, \ldots, t_n>), s)
\]

Example: When \( \text{type}(f, \text{Fun}(\text{Row}(\text{Int}, \text{Int}), \text{Int})) \), \( f \in \text{names} \), then \( \text{type}(f(1,1), \text{Int}) \).

Lambda expressions:

Lambda expressions specify functions which transform, i.e. map, objects into other objects. To describe such a transformation we use parameters. Parameters are objects without a defined value or a type:

Axiom II

At \( s \in \text{types} : \forall x \in \text{parameters} : \text{not}(\text{type}(x, s)) \)

Lambda expressions are objects in which parameters occur. The replacement of each occurrence of the same parameter in an object by one object is called Beta reduction. Another form of reduction is the replacement of names from objects by their definition (delta reduction). Evaluation of a function application is based on such reductions. We denote a function application by an expression. The evaluation operator will be formalized in another chapter. The definition of the evaluation operator specifies partly the semantics of ELISA scripts.

To assign an order to the parameters that occur in an object, all parameters that occur in that object have to be declared in the so called context(list) of that object. An object with a context is called a lambda expression. Objects with the occurrence of parameters that aren’t declared in the context(list) aren’t permitted. Thus unbound parameters are forbidden. This is a result of axiom I1.

The syntax of lambda expressions:

\[
\begin{align*}
\text{lambda expressions} & ::= [ P \text{ dom } ] \text{ objects} \\
P & ::= \text{parameters} \mid < P \text{ list } > \\
P \text{ list} & ::= P \mid P, P \text{ list} \\
\text{dom} & ::= \epsilon s \mid : \text{ objects } : \text{ types} \\
\text{parameters} & ∈ \text{ parameters}
\end{align*}
\]
Example: \([x]\text{add}(x,7)\) is called an untyped lambda expression, \([x:\text{Int}]\text{add}(x,7)\) is called a typed lambda expression, \([x], [x:\text{Int}]\) are the contexts of these lambda expressions, 
\([x][y]\) is called the context list of \([x][y](x + y)\), \(x, y\) \text{parameters} and \(\text{type}(\text{add}, \text{Fun}(\text{Row(Int, Int)}, \text{Int}))\).

The type of a typed lambda expression is determined by:

\textbf{Axiom 12}

\[\text{Ax6parameters : At\text{Objects} : As}_1, s_2 \text{types} : (\text{type}([x : s_1]t, \text{Fun}(s_1, s_2)) \iff \text{type}([x]t, \text{Fun}(s_1, s_2))\)}

Example: \(\text{type}([x : \text{Int}]x, \text{Fun}(\text{Int, Int}))\) and \(\text{type}([x : \text{Int}]\text{add}(x,7), \text{Fun}(\text{Int, Int}))\), try to prove this.

The domain type of a lambda expression restricts the set of objects over which a parameter ranges (see axiom 9 and 10).
In an untyped lambda expression the domain type isn't declared in the context. It has to be deduced with the help of:

\textbf{Axiom 13}

\[\text{Ax6P : At\text{Objects} : As}_1, s_2 \text{types} : \quad \text{(type}([x : s_1]t, \text{Fun}(s_1, s_2)) \iff \text{type}([x]t, \text{Fun}(s_1, s_2))\)}

Example: \(\text{type}([x]x, \text{Fun}(\text{Int, Int})), x\text{parameters}, \) and
\(\text{type}([<x,y>]{x,y}, \text{Fun}(\text{Row(*,*}, \text{Set(*))))}, x, y\text{parameters} \)

We may write an object of the type set in stead of a type in the context of a lambda expression. The "domain" is then restricted to a finite set. Such lambda expressions are called enumerable functions and are typed as an Efun. For enumerable functions we will define a special class of B0s. For example the dom operator which determines for an enumerable function its finite domain.

Example: \([x:\{1,2,3,4\}](x + 7)\) is a lambda expression with a finite domain.
The type of such lambda expressions can be deduced by:

Axiom 14

$\text{AxSP : } \forall o,t \in \text{objects} : \forall s_1, s_2 \in \text{types} :$

$$((\text{type}(o, \text{Set}(s_1)) \land \text{type}([x]:s_1, t, \text{Fun}(s_1, s_2))$$

and

$$((s_1 \in \text{types}) \land \text{type}([x]:s_1, t, \text{Fun}(s_1, s_2))$$

implies

$$\text{not}(\text{subtype}(s, s_1))$$

and

$$\text{type}([x]:o, t, \text{Fun}(s_1, s_2))$$

Example: $\text{type}([x]:\{1, 2, 3, 4\}[y]:\text{Int}||[1, x, y]|$, $\text{Fun}(\text{Nat}, \text{Fun}(\text{Int}, \text{List}(\text{Int})))$

and

$$\text{not}(\text{type}([x]:\{1, 2, 3, 4\}[y]:\text{Int}||[1, x, y]|$, $\text{Fun}(\text{Int}, \text{Fun}(\text{Int}, \text{List}(\text{Int})))$$

$\forall x, y \in \text{parameters}$

if $\text{subtype}(\text{Nat}, \text{Int})$ holds.

To extend the process of pattern matching we introduce possibly nested sequences of parameters in a context. Such sequences are written between $\langle \rangle$. These sequences introduce implicitly the projection operator on rows. With these sequences we introduced also the uncurried lambda expressions.

Example: $[\langle x, y \rangle : \text{Row(Int, Nat)}]\text{add}(x, y)$ is an uncurried lambda expression. Its domain type is $\text{Row(Int, Nat)}$.

$[x]:\text{Int}||[y]:\text{Nat}\text{add}(x, y)$ is a curried lambda expression. Its domain type is $\text{Int}$. Verify this!

$[\langle \langle x, y \rangle, z \rangle : \text{Row(\text{Row(Int, Int)}, Int)}]\text{add}(x, z)$ is correctly nested.
A row of parameters typed in the context as a row type:

\textbf{Axiom\textnumero 15}

\[ \forall n \in \mathbb{N}\setminus \{0\} : \mathbf{Ax_1, \ldots, x_n \in \mathcal{P}} \mathbf{ : At \in \text{objects}} : \mathbf{A_0, \ldots, A_n \in \text{types}} : \]
\[ ((\mathbf{A_0, \ldots, A_n \in \text{types}} : \]
\[ \text{eqtype}(\mathbf{A_0, \ldots, A_n \in \text{types}}) \]
\[ \text{and} \]
\[ \text{type}([x_1:A_1, \ldots, x_n:A_n]t, \text{Fun}(A_1, \text{Fun}(\ldots \text{Fun}(A_n,s)\ldots))) \]
\[ ) \]
\[ ) \]
\[ (=) \]
\[ \text{type}([[x_1, \ldots, x_n]:A_0]t, \text{Fun}(A_0,s)) \]
\[ ) \]

Example: Type([[\langle x, y \rangle: \text{Row}(\text{Int}, \text{Int})]1 \langle x, y \rangle, \]
\[ \text{Fun}(\text{Row}(\text{Int}, \text{Int}), \text{List}(\text{Int})) \]
\[ ) \]

Note: Analogously to rows it is also easy to introduce pattern matching for lists.

\textbf{Enumerable functions:}

Enumerable functions are functions with a finite set as domain. Thus some lambda expressions can be typed as an enumerable function (EF). Lambda expressions which parameters are restricted in the contextlist by an object or a "finite" type are EFs. All properties that are defined for functions, are also defined for EFs:

\textbf{Axiom\textnumero 16}

\[ \mathbf{A_1, A_2 \in \text{types}} : \text{subtype}((\text{Fun}(A_1, A_2), \text{Fun}(A_1, A_2)) \]

We introduce now the predicate fin. This predicate is recursively defined on the union of \textit{types} and \textit{objects}. With the help of fin we determine EFs.

\textit{fin}:

\[ \mathbf{As \in} (\text{types} \setminus (\text{enumerated}\_\text{types} \cup \text{new}\_\text{types})) : \text{not} \text{fin}(s) \]
\[ \text{and} \]
\[ \mathbf{A \in \text{enumerated}\_\text{types}} : \text{fin}(s) \]
\[ \text{and} \]
\[ \mathbf{A \in \text{new}\_\text{types}} : \text{fin}(n) \leftrightarrow (\mathbf{Es \in \text{types}} : \text{fin}(s) \text{ and eqtype}(n,s)) \]
\[ \text{and} \]
\[ \mathbf{A \in \text{objects}} : \text{fin}(t) \]
The type axiom for EFs is:

\texttt{axiom17}

\begin{align*}
\text{Ax6P : Ad6dom : At6objects : As} & , \text{As} \text{6types :} \\
\text{type([x:d]t, Efun(s_1, s_2))} & \iff \text{(fin(d) and type([x:d]t, Fun(s_1, s_2)))}
\end{align*}

Example: \text{type([x:{1,2,3,4}]x, Efun(Nat, Nat))}

\textbf{Indices:}

An index is an object that is typed only by the enumerated types in which this index occurs:

\texttt{axiom18}

\begin{align*}
\text{An6N}\{\emptyset\} & : A_1, \ldots, A_n \in \text{"W" :} \\
A & \in \{i_1, \ldots, i_n\} \\
\text{type}(i, \{i_1, \ldots, i_n\})
\end{align*}

Example: \text{type("true", \{"true", "false"\})}

Note: Notice the difference between \{"a\} and \{|"a"|\}!

\textbf{Projection:}

Projection is an operator on objects that are typed as a tuple. We use infix notation. Projection is a tuple followed by a dot and an index. Evaluation of a projection on an index results in the value of the field that corresponds in the tuple to this index.

The syntax of projections is determined by:

\begin{align*}
\text{projection ::= objects.index}
\end{align*}

\texttt{axiom19}

\begin{align*}
\text{At6objects : As\in\text{types : Ai\in\text{"W" :}}}
\end{align*}

\begin{align*}
(E & \in \text{types :} \\
\text{type}(t, s_1) \text{ and } s_1. i \in \text{types and eqtype}(s, s_1.i) \\
) & \iff \\
\text{type}(t, i, s)
\end{align*}

Note1: For the meaning of \"s_1.i\text{\in}\text{types}\" see axiom 1.

2: Notice that with the help of projection one may consider a tuple as multi function, i.e. a function which range type depends on its argument.
Example: `type(["a":"elisa"],"a",String)` on behalf of axiom 2

Now we are able to define the set of objects:

```
objects ::= PBO|eset|elset|grow|etuple|lambda expressions|
expression|projection|index|parameters|names|
bo_names|( objects )
```

```
names e names
bo_names e bo_names
parameters e parameters
PBO e "W" \cup "Nr" \cup "Zr" \cup "Rr" \cup "Br"
```

Note: Whenever there is a fear for ambiguity we use extra parenthesis in objects. E.g. `[x]f(x)(3),
type(f, Fun(Nat, Fun(Nat, Nat)))`, can be `[x](f(x))(3) or
`([x]f(x))(3)`.  

22
2.4 Properties of the relations subtype and eqtype

In this section we give some theorems on the relations subtype and eqtype which are important for an efficient use and a correct understanding of ELISA. We also introduce new axioms on these relations.

**Axiom 20**

subtype(Nat,Int) and subtype(Int,Real)

Note: We cannot prove this for all objects of the type Nat and Int but it holds that:

At€Nr : t€Zr and At€Zr : t€Rr

For bo names and expressions we cannot prove it.

Not all types are related to objects. Therefore we introduce the set `object_types` which is a subset of types.

definition : object_types := \{s€types|Et€objects : type(t,s)\}

**note 1:** The set of basic types is a subset of `object_types`.

2: `enumerated_types c object_types`, because an enumerated type is never empty. See axiom 18.

3: Let s€object_types and t€objects such that type(t,s), then type({t},Set(s)) and type([|t|],List(s)) on behalf of axiom 5 and 6, thus Set(s) and List(s) are elements of `object_types`.

4: Let n>=0, s_1,...,s_n€object_types and t_1,...,t_n€objects such that Aj€I..n : type(t_j,s_j), then type(<t_1,...,t_n>,Row(s_1,...,s_n)) on behalf of axiom 7, thus Row(s_1,...,s_n) belongs to `object_types`.

5: Let n>=0, s_1,...,s_n€object_types and t_1,...,t_n€objects such that Aj€I..n : type(t_j,s_j). Let i_1,...,i_n€"W", then type(<i_1:t_1,...,i_n:t_n>,Tuple(i_1:s_1,...,i_n:s_n)) on behalf of axiom 8, thus Tuple(i_1:s_1,...,i_n:s_n) belongs to `object_types`.

6: Let s_1,s_2€object_types and t_2€objects such that type(t_2,s_2) then type([x:s_1]t_2,Func(s_1,s_2)) on behalf of axiom 12, thus Fun(s_1,s_2) is an element of `object_types`.

7: Let s€object_types and t€"W" such that s€`types`.

Let t€objects such that type(t,s), then type(t,i,s,i) holds on behalf of axiom 19, thus s.i belongs to `object_types`. 
Herewith we formulate some theorems and new type axioms:

Theorem 2
\[ \text{As}_1, s_2 \text{object types:} \]
\[ \text{subtype}\left(\text{Set}(s_1), \text{Set}(s_2)\right) \Rightarrow \text{subtype}(s_1, s_2) \]

proof:

Let \( t \in \text{objects} \) such that \( \text{type}(t, s_1) \),
then \( \text{type}(\{t\}, \text{Set}(s_1)) \), \( \text{ax}5 \)
so \( \text{type}(\{t\}, \text{Set}(s_2)) \), \( \text{def. subtype} \)
hence \( \text{type}(t, s_2) \) \( \text{ax}5 \)
endproof.

Theorem 3
\[ \text{As}_1, s_2 \text{object types:} \]
\[ \text{subtype}\left(\text{List}(s_1), \text{List}(s_2)\right) \Rightarrow \text{subtype}(s_1, s_2) \]

proof: Analogously to the proof of theorem 2. endproof.

Theorem 4
\[ \text{An} \in \mathbb{N}\{0\} : \text{As}_1, \ldots, s_n \text{object types:} \]
\[ \text{Ai} \in 1..n : \text{subtype}\left(\text{Row}(s_1, \ldots, s_n), \text{Row}(s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n)\right) \Rightarrow \text{subtype}(s_i, s) \]

proof:

Let \( 1 \leq i \leq n \) and \( t \in \text{objects} \) such that \( \text{type}(t, s_i) \).
\[ s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n \text{object types}, \]
thus \( \text{lt}_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n \in \text{objects} : \)
\[ \text{Ai} \in 1..n\setminus\{i\} : \text{type}(t_j, s_j) \].
So \( \text{type}(\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n \rangle, \text{Row}(s_1, \ldots, s_n)) \), \( \text{ax}7 \)
hence \( \text{type}(\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n \rangle, \)
\[ \text{Row}(s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n) \]
), \( \text{def. subtype} \)
so \( \text{type}(t, s) \). \( \text{ax}7 \)
endproof.

Theorem 5
\[ \text{An} \in \mathbb{N}\{0\} : \text{Ai}_1, \ldots, i_n \in \mathbb{N} : \text{As}_1, \ldots, s_n \text{object types:} \]
\[ \text{Ai} \in 1..n : \text{subtype}\left(\text{Tuple}(i_1 : s_1, \ldots, i_n : s_n), \right.
\[ \left. \text{Tuple}(i_1 : s_1, \ldots, i_{j-1} : s_{j-1}, i_j : s_i, i_{j+1} : s_{i+1}, \ldots, i_n : s_n) \right) \Rightarrow \text{subtype}(s_j, s) \]

proof: Analogously to the proof of theorem 4. endproof.
Theorem 6
As₁, s₂, s ∈ object_types :
subtype(Fun(s, s₁), Fun(s, s₂)) ⇒ subtype(s₁, s₂)
proof:
Let t₁ ∈ objects such that type(t₁, s₁).
s ∈ object_types, thus Et ∈ objects : type(t, s).
Let x ∈ parameters such that not(member(x, t₁)).
Consider the lambda expression [x:s]t₁.
Then type([x:s]t₁, Fun(s, s₁)), (ax12)
so type([x:s]t₁, Fun(s, s₂)), (def. subtype)
thus type([x:s]t₁(t), s₂), (ax9)
hence type(t₁, s₂), because applying a function is replacing all
parameters in the object definition of that function by the
objects listed as arguments in the expression that specifies
this function application.
endproof.

The reverse of the implication in theorems 2, 3, 4, 5, and 6 isn’t
provable. Therefore we introduce them as axioms over types’:

Axiom 21
As₁, s₂ ∈ types’ :
subtype(s₁, s₂) ⇒ subtype(Set(s₁), Set(s₂))

Axiom 22
As₁, s₂ ∈ types’ :
subtype(s₁, s₂) ⇒ subtype(List(s₁), List(s₂))

Axiom 23
An ∈ N\{0} : As₁, ..., sₙ, s ∈ types’ : Ai ∈ 1..n :
subtype(sᵢ, s) ⇒
subtype(Row(s₁, ..., sₙ), Row(s₁, ..., s⁻₁, s, s⁺₁, ..., sₙ))

Axiom 24
An ∈ N\{0} : Ai, ..., iₙ ∈ W’ : As₁, ..., sₙ, s ∈ types’ : Aj ∈ 1..n :
subtype(sᵢ, s) ⇒
subtype(Tuple(i₁:s₁, ..., iₙ:sₙ),
Tuple(i₁:s₁, ..., i⁻₁:s⁻₁, i⁻₁:s⁻₁, i⁺₁:s⁺₁, ..., iₙ:sₙ))
Axiom25

As \(s_1, s_2, s \in \text{types} \):

\[
\text{subtype}(s_1, s_2) \Rightarrow \text{subtype}(\text{Fun}(s, s_1), \text{Fun}(s, s_2))
\]

Note: This axiom wouldn't be correct for subtyping the domain type.

E.g. When \(\text{type}(f, \text{Fun}(\text{Nat}, \text{Real})), f \in \text{objects}, \)
for each \(x \in \text{N} f(x)\) is defined as the square root of \(x\),
then it isn't correctly that \(\text{type}(f, \text{Fun}(\text{Int}, \text{Int}))\) because arguments of \(f\) have to be positive.

Theorem7

If we replace \(\text{subtype}\) by \(\text{eqtype}\) in theorems 2, 3, 4, 5 and 6 we regain correct theorems.
proof: See the definition of \(\text{eqtype}\). endproof.

Theorem8

If we replace \(\text{types}'\) by \(\text{types}\) or \(\text{subtype}\) by \(\text{eqtype}\) in axioms 21, 22, 23, 24 and 25 we regain correct theorems.
proof: See axiom 3 and the definition of \(\text{eqtype}\). endproof.
2.5 Commands and variables

Variables are language objects with a name (variables) and a type. This name and type are declared in an ELISA script. We will denote the type of variable \( v \) with \( \text{tp}(v) \). Thus \( \text{tp} \) is a function from the set of variables to the set of types. The type of a variable \( v \) determines the set of objects over which that variable ranges, i.e. \( \text{SO}(\text{tp}(v)) \). Variables are used to indicate the input and output values of commands. We distinguish output variables, input variables and state variables. We will denote the sort of variable \( v \) with \( \text{srt}(v) \). Therefore \( \text{srt} \) is a function from the set of variables to the set \{ivar,ovar,svar\}.

The syntax of variables:

\[
\text{var decl ::= \text{sym ::= \text{var type list}}} \\
\text{sym ::= ovar|ivar|svar} \\
\text{var type list ::= variables : types} \\
\text{variables : types, var type list} \\
\text{variables e variables}
\]

A command is a concurrent assignment, denoted by \( <- \). As argument serves a sequence of input variables and state variables. The effect of a command is specified by a function name. Evaluation of a command is evaluation of the function indicated by the name with as input the current values of the listed variables. After computations new values are assigned to output variables and state variables. These variables are listed in the command definition on the left of \( <- \). Commands are defined in an ELISA script.

The syntax of commands:

\[
\text{command ::= \langle \text{var list} \rangle \langle- \text{names ( var list )} \\
\text{var list ::= variables | variables, var list}
\]
Now we can specify a correctly defined command. Herefore we introduce the predicate correctly_defined_command on the set of commands.

Axiom 26
\[ \text{An} \in \mathbb{N} \setminus \{0\} : \text{Ax}_{1, \ldots, x_n} \in \text{variables} \quad \text{An} \in \mathbb{N} \setminus \{0\} : \\
\text{Ay}_{1, \ldots, y_m} \in \text{variables} : \text{Afnames} : \\
(((\forall \text{Aj} \in \ldots \text{n} : \text{srt}(x_j) \in \text{svar, ovar} \quad \text{and} \quad \#(i \in \ldots \text{n} | x_i = x_j) = 1) \quad (*) \quad \text{and} \\
((\forall \text{Aj} \in \ldots \text{m} : \text{srt}(y_j) \in \text{svar, ivar})) \\
((\forall \text{Es}_1, \ldots, s_n, s_1, \ldots, s_m \in \text{types} : \\
(\forall \text{Aj} \in \ldots \text{n} : \text{tp}(x_j) = s_j) \\
(\forall \text{Aj} \in \ldots \text{m} : \text{tp}(y_j) = s_j) \\
((m = 1 \quad \text{and} \quad n = 1) \Rightarrow \text{type}(f, \text{Fun}(s_1, s_1^*)) \\
((m > 1 \quad \text{and} \quad n = 1) \Rightarrow \text{type}(f, \text{Fun}(\text{Row}(s_1, \ldots, s_m), s_1^*)) \\
((m = 1 \quad \text{and} \quad n > 1) \Rightarrow \text{type}(f, \text{Fun}(s_1, \text{Row}(s_1^*, \ldots, s_n)), \\
\text{Row}(s_1^*, \ldots, s_n)) \\
) \\
) \\
) \\
) \\
) \\
) \\
) \]

Note: (*) This expresses that with the help of a command one can bind each output variable only to one value.
2.6 Definitions, forming an ELISA script

2.6.1 Axioms in ELISA

One can define new axioms in ELISA. These axioms extend the axioms of the type system. They will be used to verify the correctness of our ELISA script with respect to our type axioms.

Syntax of axioms:
axiom ::= type( names, types )

Example: type(sum,Fun(Row(Nat,Nat),Nat)), sum\\in\text{\textit{names}}, can be an axiom that is defined by the programmer.

Note 1: Axioms are part of an ELISA script.
2: It is the responsibility of the programmer of an ELISA script, that the axioms he defines, aren't in mutual contradiction or in contradiction with the rest of the type axioms.
3: Further on in section 2.6.2 we introduce a method to define synonyms for types. An eqtype axiom will hold for such abbreviations.

2.6.2 The correctness specification of an ELISA script

An ELISA script is a sequence of sentences:

\begin{align*}
\text{ELISA script} & ::= \text{sentence} @ | \text{sentence} ; \text{ELISA script}
\end{align*}

Most sentences start with a name, followed by an assignment sign. A name belongs to the set of names, new types or command names. A definition is the assignment of a name to a language object. This name can be used in the rest of the ELISA script as a synonym for this language object. All properties of a language object that is abbreviated by a name are transferred to that name. This is done with the help of the type or the eqtype relation. We distinguish object definitions, type definitions and command definitions.

At the end of an object definition one can add a type. This type has to be a type of that object. A special subclass of object definitions are the basic objects (B0s). These are objects
with a name and a type without a concrete definition in ELISA. Typing of BOs is obligatory. BOs are used to build more complex objects. Examples of BOs are given in chapter 3.

`newtypes` and `command_names` are the sets of names used to abbreviate types and commands.

A sentence in an ELISA script is a definition, a type axiom or the declaration of variables.

Syntax of sentences:

```
sentence ::= axiom|var decl|definition
definition ::= names := objects typing |
            bo names := BO : types |
            new types := types |
            command names := command
typing ::= eps : types
new types c new_types
names c names
bo names c bo_names
command names c command_names
```

Now we will give the axioms to verify the correctness of the individual sentences of a given ELISA script. Herefore we introduce the predicate correct (with respect to our ELISA script), which is defined on the set of sentences. We also introduce set AX. AX is a set of axioms belonging to our ELISA script. The elements of AX are used to verify the correctness of our ELISA script.

Axiom sentences:

**Axiom27**

**Aax∈axiom** :

`correct(ax)`

and

when such a correct sentence occurs in a given ELISA script, then it holds that ax∈AX

Variable declarations:

**Axiom28**

**An∈N : Av₁,...,vn∈variables : As₁,...,sn∈types : Asymb∈{ovar,ivar,svar} :**

`correct(symb::v₁:s₁,...,vn:sₙ)`

and
when such a correct sentence occurs in a given ELISA script, then it holds that
\[(\forall j \in \mathbb{N} \colon \text{tp}(v_j) = s_j \text{ and } \text{srt}(v_j) = \text{sym})\]

**Definitions of language objects**

**Command definitions:**

**Axiom 29**

\[\text{An6names} : \text{Ac6commands} : \]
\[(\text{correct}(n::=c) \iff \text{correctly_defined_command}(c))\]

**BO definitions:**

**Axiom 30**

\[\text{An6bo_names} : \text{As6types} : \]
\[(\text{correct}(n:=\text{BO}:s) \text{ and when such a correct sentence occurs in a given ELISA script, then it holds that } \text{type}(n,s) \in \text{AX})\]

**Newtype definitions:**

**Axiom 31**

\[\text{An6new_types} : \text{As6types} : \]
\[(\text{correct}(n==s) \text{ and when such a correct sentence occurs in a given ELISA script, then it holds that } \text{eqtype}(n,s) \in \text{AX})\]

**Object definitions with the exception of lambda expressions:**

**Axiom 32**

\[\text{An6names} : \text{At6(objects \ \setminus \ \text{lambda_expressions})} : \]
\[((\text{As6types} : (\text{type}(t,s) \iff \text{correct}(n:==t:s))) \text{ and ((Es6types} : \text{type}(t,s)) \iff \text{correct}(n:=t)) \text{ and when such a correct sentence occurs in a given ELISA script, then it holds that } (\text{As6types} : \text{type}(t,s) \iff \text{type}(n,s)) \in \text{AX})\]
Recursively defined lambda expressions are permitted in an ELISA script. This means that the name of a lambda expression can be used in the definition of that lambda expression. But this name may only occur in the heading of an expression. Otherwise we could create functions like:

\[ g := \{x : \text{Int} \} < x, g \} \text{ or } f := \{x : \text{dom}(f) \} f(x), \]
where \text{dom} is a function \((\text{BO})\) that assigns to a function its domain.

(recursive) function definitions:

\text{Axiom33}

\( \text{Af6names} : \text{At6lambda_expressions} \)

\( (\text{A}\_\text{s2etypes} : \)

\( (\text{every occurrence of } f \text{ in } t \text{ is followed by } "(" \) \)
and

\( (\text{Es1s2etypes} : \text{type}(f, \text{Fun}(s1, s2)) \Rightarrow \text{type}(t, \text{Fun}(s1, s2))) \)
\)

\( <=> \)

\( \text{correct}(f := t : s2) \)
\)

and

\( (\text{every occurrence of } f \text{ in } t \text{ is followed by } "(" \) \)
and

\( (\text{Es1s2etypes} : \text{type}(f, \text{Fun}(s1, s2)) \Rightarrow \text{type}(t, \text{Fun}(s1, s2))) \)
\)

\( <=> \)

\( \text{correct}(f := t) \)
\)

and

when such a correct sentence occurs in a given ELISA script then it holds that

\( (\text{Asetypes} : \text{type}(t, s) \Leftrightarrow \text{type}(f, s)) \text{EAX} \)
\)

Now we are able to give the partial correctness definition of a given ELISA script:

An ELISA script is correct when

1: All names of (basic) object definitions, type definitions and command definitions occur only once at the beginning of a sentence in this script.
2: All variables are declared only once in this script and there is declared only one state variable.

3: Every sentence in the script is correct with respect to the type axioms of this script. The type axioms are axioms 1,...,33 from this chapter, the axioms that are defined in the given ELISA script and the elements of AX

Note: It is easy to verify the correctness of a new ELISA script that arises from the addition of a sentence to a correct ELISA script. One has to check only the correctness of the added sentence with respect to the old ELISA script.

The type checker:
The type checker is an operator for the correctness verification of a given ELISA script:

\[ \text{AbELISA}\_\text{script} : \text{type}\_\text{checker}(b) := b \text{ is correct.} \]
2.7 The model of an IS

An information system can be modelled with the help of an ELISA script. We define the state space of variable $v$ that is declared in an ELISA script by the set $SO(tp(v)) \{\text{empty}\}$. Empty denotes empty state.

An IS belonging to a correctly given ELISA script $b$ (thus $\text{type\_checker}(b)$ holds) is the four row $<I,O,S,T>$ where,

1: $I$ is the Cartesian product of the state spaces of the variables that are declared in script $b$ with $srt(v)=ivar$. Their order is irrelevant. We define the input variable as a variable that ranges over $I$.

2: $O$ is the Cartesian product of the state spaces of the variables that are declared in script $b$ with $srt(v)=ovar$. Their order is irrelevant. We define the output variable as a variable that ranges over $O$.

3: We suppose that there is declared a state variable in script $b$. So from the end of section 2.6 we infer that there is exact one state variable. $S$ is the state space of the state variable of script $b$.

4: $T$ is a function valued function such that the domain of $T$ is the set of command names defined in script $b$ and

Ac\text{command\_names} :

$T(c)$ is a function from the Cartesian product of $I$ and $S$ to the Cartesian product of $O$ and $S$.

The effect of $T(c)$ is indirectly described by $f$ where $f$ is the name of the function used to specify in script $b$ the effect of command $c$.

After the receipt of a command name $c$ the ELISA machine becomes active and binds the input variable to a value. Then the machine evaluates $T(c)(i,s)$, where $i$ is the current value of the input variable, $i\in I$ and $s$ is the current value of the state variable, $s\in S$. After computations the left projection of the result of the evaluation of $T(c)(i,s)$ is assigned to the output variable and the right projection of the result of the evaluation of $T(c)(i,s)$ is assigned to the state variable. Then the machine becomes passive again.

Specifying an IS is writing an ELISA script. During the lifetime of an IS this ELISA script doesn't change. A change of this ELISA script is considered as a change of the IS.
3. Standard constructions

3.1 Working with ELISA

In this chapter we will give an example of a correct ELISA script. We introduce some sets and functions that are useful components for the design of a large variety of ELISA scripts. The goal of our research is constructing a functional language that supports the design of an IS. This doesn't imply that ELISA can be used only for this purpose. It covers a greater field of applications. Therefore the ELISA script that is given in this chapter, will be a universal one, which can be used for the construction of many programs. We will call this ELISA script "the first chapter of an ELISA script".

Bos are necessary to construct new objects. If we want to build an implementation of ELISA, then we must build an implementation of all Bos that can occur in ELISA scripts, based on their semantics. This is impossible. We have to restrict ourselves. We make the agreement that Bos can be defined only in the first chapter of an ELISA script. Deviation of this rule is only permitted when the programmer defines a BO with which he delivers an implementation. If we take care that all Bos of the first chapter are implementable, and build an interpreter for this first chapter, including for the coping with commands, then we will have an implementation for a large number of ELISA scripts.

An ELISA script without a certain rate of explanation isn't readable. Explanations will be denoted between % signs. The semantics (Smt) of Bos will be described in the same metalanguage as we introduced in chapter 2. We will use the so operator defined in section 2.5. Hereby we must realize that we use operators on sets of objects created with so which aren't formally defined. We will correct this negligence in a following chapter. Especially the semantics of the equality operator needs special attention. The formal definition of the syntax of ELISA doesn't allow infix notations, although one is common to work with them. Again for readability considerations, we will accept the use of such a syntactical sugaring.

ELISA is meant for the design of ISs. Therefore we will give in the last two sections of this chapter a discription of a method for the design of the functional data model and the
relational data model. We will line out the construction of the state spaces of the databases described in these data models. We will also introduce the corresponding state variables. Consider this as the introduction to the second chapter of an ELISA script.
3.2 The first chapter of an ELISA script

%the conditional operator and the recursor:%

cond := BO : Fun(Row(Bool,*,*),*);
% Smt: Ab€B : As€types : Ax,y€S0(s) :
  cond(b,x,y) := x if b=true
          y if b=false

note: The conditional operator corresponds with the if statement in procedural languages.

% recursor := BO : Fun(Row(*,Efun(**,Fun(*,*»),*);
% Smt: As I ,s2€types :
  Ax€S0(sI) :
  Af€S0(Efun(s2,Fun(sl'sI»)
recursor(x,f) := x if dom(f)={} 
   f(h)(recursor(x,f'))
   where h€dom(f) and f' is defined as 
   dom(f') := dom(f)\{h} 
   and Ay€dom(f') : f'(y)=f(y) 
   if dom(f)#{}

note 1: The recursor quantifies function f over dom(f).
Therefore we will call all lambda expressions, which are based on the recursor, quantifications.
2: Application of the recursor isn't unique determined.
It depends on the order of choosing h€dom(f). Therefore we need the precondition that:
Ax,y€dom(f) : At€S0(sI) : f(x)(f(y)(t))=f(y)(f(x)(t))
3: It isn't necessary that the recursor is a BO. One can construct him recursively with the conditional operator and the pick operator. We will do this in section 3.3.
4: All applications of the recursor can be replaced by a recursive definition.

%logical operators:%

impl := [<b1,b2>:Row(Bool,Bool)]
cond((b1 = false),true,cond((b2 = false),false,true))
: Bool;
%infix: (b1 -> b2)

Note 1: Impl represents the logical implication.
2: For the definition of "=" see "comparative operators".
%
not := \[b:Bool\](b -> false) : Bool;
or := \[(b_1,b_2):Row(Bool,Bool)\](not(b_1) -> b_2) : Bool;
\%infix: (b_1 or b_2) \%
and := \[(b_1,b_2):Row(Bool,Bool)\]not(b_1 -> not(b_2)) : Bool;
\%infix: (b_1 and b_2) \%
eq := \[(b_1,b_2):Row(Bool,Bool)\]
((b_1 -> b_2) and (b_2 -> b_1)) : Bool;
\%infix: (b_1 <-> b_2)

Note: From the point of view of efficiency one may consider
to make basic objects of all logical operators. This
holds for all the defined objects in this section.

all := \[p:Efun(*,Bool)\]
recursor(true,[x:dom(p)]\[b:bool\](p(x) and b)) : Bool;
\%Note: All represents the universal quantor from logics.\%
exist := \[p:Efun(*,Bool)\]not(all([x:dom(p)]not(p(t))) : Bool;
\%Note: Exist represents the existential quantor from logics.\%

\%comparitive operators:%
is := BO \: Fun(Row(*,*),Bool);
\%Smt: As\$t\$es : Ax,y\$\in\$SO(s) :
    is(x,y) := true if x=y
    \false if x\#y
    \%infix: (x = y)
\%
isnot := \[(x,y):Row(*,*)\]not(x = y) : Bool;
\%infix: (x , y) \%
less := BO \: Fun(Row(Real,Real),Bool);
\%Smt: Ax,y\$\in\$R :
    less(x,y) := true if x<y
    \false if x\geq y
    \%infix: (x < y)
\%
moreis := \[(x,y):Row(Real,Real)\]not(x < y) : Bool;
\%infix: (x >= y) \%
more := \[(x,y):Row(Real,Real)\]((x >= y) and (x \# y)) : Bool;
\%infix: (x > y) \%
lessis := \[(x,y):Row(Real,Real)\]((x < y) or (x = y)) : Bool;
\%infix: (x <= y) \%
max := \[(x,y):Row(Real,Real)\]cond((x >= y),x,y) : Real;
min := \[(x,y):Row(Real,Real)\]cond((x < y),x,y) : Real;
%set operators:%

\begin{align*}
\text{ins} & := \text{BO} : \text{Fun}(\text{Row}(\text{Set}(\text{*}),\text{*}),\text{Set}(\text{*})); \\
\text{del} & := \text{BO} : \text{Fun}(\text{Row}(\text{Set}(\text{*}),\text{*}),\text{Set}(\text{*})); \\
\text{pick} & := \text{BO} : \text{Fun}(\text{Set}(\text{*}),\text{*}); \\
\text{setg} & := \{p: \text{Efun}(\text{*},\text{Bool})\} \\
\text{member} & := \{<x,S>: \text{Row}(\text{*},\text{Set}(\text{*}))\} \text{exist}([y:S](x = y)) : \text{Bool}; \\
\text{notmember} & := \{<x,S>: \text{Row}(\text{*},\text{Set}(\text{*}))\} \text{not}(x \in S) : \text{Bool}; \\
\text{subset} & := \{<S_1,S_2>: \text{Row}(\text{Set}(\text{*}),\text{Set}(\text{*}))\} \\
& \text{All}([x:S_1](x \in S_2)) : \text{Bool}; \\
\text{union} & := \{<S_1,S_2>: \text{Row}(\text{Set}(\text{*}),\text{Set}(\text{*}))\} \\
& \text{recursor}(S_1,[x:S_2][S:\text{Set}(\text{*})]\text{ins}(S,x)) : \text{Set}(\text{*}); \\
\text{diff} & := \{<S_1,S_2>: \text{Row}(\text{Set}(\text{*}),\text{Set}(\text{*}))\} \\
& \text{recursor}(S_1,[x:S_2][S:\text{Set}(\text{*})]\text{del}(S,x)) : \text{Set}(\text{*});
\end{align*}

%Note: The \text{set_generator} assigns to a boolean function \( p \)
%the subset of the domain of \( p \) where it has the value true.
intersect := [\langle S_1, S_2 \rangle : \text{Row}(\text{Set}(*), \text{Set}(*))] 
setg([x:S_1](x \in S_2)) : \text{Set}(*);
%Infix: \( S_1 \cap S_2 \)
number := [S:\text{Set}(*)]\text{sum}([x:S]1) : \text{Nat};
%Note 1: Number counts the number of elements of \( S \).
2: For the definition of \text{sum} see "operators on reals".
%
empty := [S:\text{Set}(*)](\text{number}(S) = 0) : \text{Bool};

%functional operators:%

dom := \text{BO} : \text{Fun}(\text{Efun}(*,**), \text{Set}(*));
%Smt: As \( A_1, A_2 \) types : \text{Af6S0}(\text{Fun}(s_1, s_2)) :
  \text{dom}(f) := \text{the domain of } f.
%Note: The domain is computable because \( f \) is an \( \text{EF} \).
%
rng := [f:\text{Efun}(*,**)]
    \text{recursor}({}, [x:\text{dom}(f)][S:\text{Set}(**)]\text{ins}(S, f(x))) : \text{Set}(**);
%Note: Rng computes the range of function \( f \). %
maxf := [f:\text{Efun}(*,\text{Real})]
    \text{recursor}(\text{minvalue}, [x:\text{dom}(f)][y:\text{Real}]\text{max}(f(x), y)) : \text{Real};
%Note 1: Maxf computes the maximum of function \( f \).
2: For the definition of \text{minvalue} see "operators on reals".
%
minf := [f:\text{Efun}(*,\text{Real})]-(\text{maxf}([x:\text{dom}(f)]-(f(x)))) : \text{Real};
%Note: Minf computes the minimum of function \( f \). %
fununion := [\langle f_1, f_2 \rangle : \text{Row}(\text{Efun}(*,**), \text{Efun}(*,**))]
    [x:(\text{dom}(f_1) \cap \text{dom}(f_2))]
    \text{cond}((x \in \text{dom}(f_1), f_1(x), f_2(x)) : \text{Efun}(*,**);
%Note 1: Notice the importance of the order of \( f_1 \) and \( f_2 \) when
their domains aren't disjunct.
2: Fununion is an example of a curried function.
%
extend := [f:\text{Efun}(*,**)]
    \text{recursor}({}, [x:\text{dom}(f)][S:\text{Set}(\text{Row}(*,**))] 
    \text{ins}(S, \langle x, f(x) \rangle)
)
    : \text{Set}(\text{Row}(*,**));
%Note: extend assigns to a function its corresponding set of pairs.
%
intend := BO : Fun(Set(Row(*,**)),Efun(*,**));
%Smt: As1,s2€types : A$ESO(Set(Row(s1,s2))) :
    Intend assigns to a set of pairs the corresponding
    function, i.e. intend(S) := f where S=extend(f).
    Note: Not all sets of pairs imply a function. Therefore we
    need the precondition:
    As1,s2ES :
      ((left(s1) ≠ left(s2)) or (right(s1)=right(s2))).
For the definition of left and right see "row
operators".
%
%list operators:%

top := BO : Fun(List(*),*)
%Smt: As€types : A$\in\{0\} : Ax1,...,x_nESO(s) :
    top([[x1,...,x_n]]) := x1
    Note: Be aware that for top(ls) the precondition ls$\neq$[] must
    hold, ls$\in$ESO(List(s)).
%
pop := BO : Fun(List(*),List(*));
%Smt: As€types : A$\in\{0\} : Ax1,...,x_nESO(s) :
    pop([[x1,...,x_n]]) := [[x2,...,x_n]]
    Note: Be aware that for pop(ls) the precondition ls$\neq$[] must
    hold, ls$\in$ESO(List(s)).
%
push := BO : Fun(Row(List(*),*),List(*));
%Smt: As€types : A$\in\{0\} : Ax1,...,x_nESO(s) :
    push([[x1,...,x_n]],x) := [[x,x1,...,x_n]]
%
%row operators:%

left ::= [<x,y>:Row(*,**)]x : * ;
right ::= [<x,y>:Row(*,**)]y : ** ;
prod ::= [<S1,S2>:Row(Set(*),Set(**))]
    recursor({},[x:S1][S':Set(Row(*,**))])
    union(S',
        recursor({},[y:S2][S":Set(Row(*,**))]
          ins(S",<x,y>))
    )
)

41
):Set(Row(*,**));

%Note: prod computes the Cartesian product of S1 and S2.%

%operators on reals:%

swap := BO : Fun(Real,Real);
%Smt: Ax6R : swap(x) := -x
  prefix: -(x)
%
add := BO : Fun(Row(Real,Real),Real);
%Smt: Ax,y6R : add(x,y) := x+y
  infix: (x + y)
%
subtract := [<x,y):Tuple(Real,Real)](x + -(y)) : Real;
%infix: (x - y)
%
mult := BO : Fun(Row(Real,Real),Real);
%Smt: Ax,y6R : mult(x,y) := x* y
  infix: (x * y)
%
sum := [f:Efun(*,Real)]
  recursor(O,
    [x:dom(f)] [y:Real](f(x) + y)) : Real;
prod := [f:Efun(*,Real)]
  recursor(I,
    [x:dom(f)] [y:Real](f(x) * y)) : Real;
power := [<x,n):Tuple(Real,Nat)]prod([k:(1..n)]x) : Real;
%infix: (x ^ n)
Note: For the semantics of "l..n" see "some useful sets".
%
inv := BO : Fun(Real,Real);
%Smt: Ax6R : inv(x) := l/x
  Note: Be aware that the precondition x#0 must hold.
%
quot := [<x,y):Row(Real,Real)](x * inv(y));
%Note: Be aware that the precondition y#0 must hold.
  infix: (x / y)
%
maxvalue := BO : Real;
%Smt: Ax6R : x < maxvalue %
minvalue := -(maxvalue) : Real;
sign := [x:Real]cond((x = 0),0,cond((x > 0),1,-1)) : Int;
sqr := [x:Real](x * x) : Real;
sin := BO : Fun(Real,Real);
\texttt{cos := BO : Fun(Real,Real);}
\texttt{arctan := BO : Fun(Real,Real);}
\texttt{ln := BO : Fun(Real,Real);}
\texttt{exp := BO : Fun(Real,Real);}
\texttt{sqrt := BO : Fun(Real,Real);}
\texttt{trunc := BO : Fun(Real,Real);}
\texttt{round := BO : Fun(Real,Real);}

\textcolor{red}{%Note: The semantics of these real operators is the usual one.\%}

\texttt{\%operators on integers:\%}
\texttt{type(swap,Fun(Int,Int));}
\texttt{type(add,Fun(Row(Int,Int),Int));}
\texttt{type(subtract,Fun(Row(Int,Int),Int));}
\texttt{type(mult,Fun(Row(Int,Int),Int));}
\texttt{type(abs,Fun(Int,Nat));}
\texttt{type(maxvalue,Int);}
\texttt{type(minvalue,Int);}
\texttt{type(sum,Fun(Efun(*,Int),Int));}
\texttt{type(prod,Fun(Efun(*,Int),Int));}
\texttt{type(power,Fun(Tuple(Int,Nat),Int));}
\texttt{div := \{<n,m>:<Int,Int>\}}
\texttt{((sign(n) \times sign(m))}
\texttt{\times}
\texttt{cond(((n < 0) or (m < 0)),}
\texttt{div(abs(n),abs(m)),}
\texttt{cond((n < m),0,(div(n - m),m) + 1))}
\texttt{)}
\texttt{: Int;}

\textcolor{red}{%Note: Be aware that the precondition m\neq0 must hold.\%}
\texttt{mod := \{<n,m>:Tuple(Int,Int)\}(n - (m \times \text{div}(n,m))) : Int;}

\texttt{\%operators on naturals:\%}
\texttt{type(add,Fun(Row(Nat,Nat),Nat));}
\texttt{type(mult,Fun(Row(Nat,Nat),Nat));}
\texttt{type(maxvalue,Nat);}
\texttt{type(sum,Fun(Efun(*,Nat),Nat));}
\texttt{type(prod,Fun(Efun(*,Nat),Nat));}
\texttt{type(power,Fun(Tuple(Nat,Nat),Nat));}
\texttt{type(div,Fun(Row(Nat,Nat),Nat));}
type(mod, Fun(Row(Nat, Nat), Nat));

% operators on strings: %

eps := "" : String;
first := BO : Fun(Row(String, Nat), String);
rest := BO : Fun(Row(String, Nat), String);
concat := BO : Fun(Row(String, String), String);

% Note 1: The semantics of these BOs is defined in section 2.2.
2: Compare these BOs with the BOs for list operators.
%

num := [x: String] cond((x = eps), 0, (num(rest(x, 1)) + 1)) : Nat;

% some useful finite sets: %

I := BO : Fun(Row(Int, Int), Set(Int));

% Smt: I(n, m) := n..m
  infix: (n..m)
%

type(I, Fun(Row(Nat, Nat), Set(Nat)));

% The universal object: %

error := BO : *;

% Smt: Every function that is evaluated with error as an argument
  has error as result, with the exception of the equality
  operator, i.e. is(error, error) := true.

Note: As types : type(error, s).
%
3.3 A few examples

**The recursor:**
In section 3.2 we mentioned that the recursor isn't necessarily a basic object. We can build him with the help of the conditional operator, the dom operator and the pick operator. If we do so, evaluation of the recursor will become less efficient. It isn't surprising that the ELISA definition of the recursor harmonizes strongly with its semantical definition.

\[ \text{recursor} := \left\{ \langle x, f \rangle : \text{Row}(*, \text{Ef}(**, \text{Fun}(*, *)) \right\} \]

\begin{align*}
\text{cond}(\text{empty}(\text{dom}(f)), \\
x, \\
f(\text{pick}(\text{dom}(f))) \\
(\text{recursor}^\prime(x, \{y : (\text{dom}(f) \setminus \text{pick}(\text{dom}(f)))\}f(y))) \\
) \\
: *;
\end{align*}

**Linear search:**
One may represent a finite sequence in ELISA by a function on \(1..n, n \in \mathbb{N}\). Suppose one would like to determine the set of zeros of such a sequence \(s\), i.e. \(\{m \in 1..n \mid s(m) = 0\}\). The most convenient solution would be

\[ \text{zeros} := \left\{ \langle s, n \rangle : \text{Row}(\text{Fun}(\mathbb{N}, \mathbb{R}), \mathbb{N}) \right\} \text{setg}([m : (1..n)](s(m) = 0)); \]

We may also use the recursor:

\[ \text{zeros2} := \left\{ \langle s, n \rangle : \text{Row}(\text{Fun}(\mathbb{N}, \mathbb{R}), \mathbb{N}) \right\} \]

\begin{align*}
\text{recursor}([], [], [m : (1..n)] [S : \text{Set}(\mathbb{N})] \\
\text{cond}(s(m) = 0, \text{ins}(S, m), S) \\
); \\
\end{align*}

We noticed in a note in section 3.2 that we can replace every use of the recursor by a recursive definition:

\[ \text{help3} := \left\{ \langle s, m, n \rangle : \text{Row}(\text{Fun}(\mathbb{N}, \mathbb{R}), \mathbb{N}, \mathbb{N}) \right\} \]

\begin{align*}
\text{cond}((m > n), \\
\{\}, \\
\text{cond}(s(m) = 0), \\
\text{ins}(\text{help3}(s, (m + 1), n), m), \\
\text{help3}(s, (m + 1), n) \\
) \\
); \\
\end{align*}

Then the definition of \(\text{zeros3}\) will be:
zeros3 := [<s,n>:Row(Fun(Nat,Real),Nat)]help3(s,1,n);

If we know that the sequence is ordered, i.e.
\[ \text{Ai6l..n : Aj6l..n : } s(i) \leq s(j) \]
then we can use binary search to search more efficiently.
Nevertheless we must be careful that we don't forget that ELISA is a functional programming language. We mustn't use it as a procedural language. Efficiency is a minor problem.

The invertor:
The pop operator, top operator and push operator cause that lists can be used as stacks. The invertor transforms a list into the reverse ordered list.

help := [<ls1,ls2> : Row(List(*),List(*))]
    cond((ls2 = [[l]]),
        ls1,
        help(push(ls1,top(ls2)),pop(ls1))
    );
invertor := [ls:List(*)]help([[l]],ls1);

Combinatorial theory:
Many mathematical specifications cannot be transformed directly into ELISA. Often the specification isn't computable because the search space is infinite. To prevent such failures we introduced enumerable functions.

We define
\[ \text{An6N : } \text{Ak60..n :} \]
\[ \text{comb(n,k) := the number of subsets of l..n with k elements.} \]

In ELISA we would like to write:
\[
\text{comb} := [n:Nat][k:(0..n)]
    \text{number(setg([S:Set(Nat)]}
        ((\text{number}(S) = k) \text{ and } (S \text{ c (1..n))))
    )
\];

This isn't correct because
\[ [S:Set(Nat)]((\text{number}(S) = k) \text{ and } (S \text{ c (1..n)))} \]
isn't an enumerable predicate and therefore the set_generator isn't evaluable. Mostly one can solve such problems with recursion.

From combinatorics we gain the property that:

\[ \text{An} \in \mathbb{N} : \text{comb}(n, 0) = 1 \text{ and comb}(n, n) = 1 \]

and

\[ \text{An} \in \mathbb{N} \setminus \{0\} : \text{Ak} \in \cdots (n-1) : \text{comb}(n, k) = \text{comb}(n-1, k) + \text{comb}(n-1, k-1) \]

Herewith we can create a correct definition for comb:

\[
\text{comb} := [\text{n} : \mathbb{N}][\text{k} : (0..\text{n})]
\begin{align*}
\text{cond} & (\text{(k = 0) or (k = n)}), \\
& 1, \\
& (\text{comb(n - 1)(k) + comb(n - 1)(k - 1)})
\end{align*}
\]

Note: Notice that the precondition \(0 \leq k \leq n\) has to be verified.
3.4 The functional data model

We may use the functional data model for constructing a database space. According to this model the database space can be generated by a five tuple, called the skeleton:

$$<CI, FI, D, R, V>$$

where:
1: CI is the set of category indices, CI ⊆ S, CI is finite.
2: FI is the set of function indices, FI ⊆ S, FI is finite,
3: D is a function from FI to CI, D is called the domain function.
4: R is a function from FI to CI, R is called the range function.
5: V is a function from CI to types.

The state space of category c ∈ CI is the set SO(V(c)). The state space of function f ∈ FI is the set SO(Fun(V(D(f)), V(R(f))))).

The design of a database space based on the functional data model, is the definition of the skeleton. This skeleton must be specified in an ELISA script.

Firstly we introduce:
$$\{c_i\}_{1 \leq i \leq n}, n = \#(CI),$$
is an enumeration of set CI.
$$\{f_i\}_{1 \leq i \leq m}, m = \#(FI),$$
is an enumeration of set FI.

To define the skeleton in ELISA we must add to the ELISA script:

CI := \{"c_1", \ldots, "c_n"\};
FI := \{"f_1", \ldots, "f_m"\};
D := |<"f_1":???", \ldots, "f_m":???">|;
R := |<"f_1":???", \ldots, "f_m":???">|;
V := Tuple("c_1":???, \ldots, "c_n":???);

Note 1: Instead of ??? one must write the domain category index, the range category index or the type of the category that corresponds to the index preceding the colon.
2: CI and FI are enumerated types, V is a tuple type and D and R are objects.

New types are used to denote the type (i.e. structure) of the state space of the categories, the functional state space and the
state space of the database.

\[ C \equiv \text{Tuple}("c_1":\text{Set}(V."c_1"), \ldots,"c_n":\text{Set}(V."c_n")); \]

\[ F \equiv \text{Tuple}("f_1":\text{Efun}(V.D."f_1",V.R."f_1"), \]
\[ \ldots, \]
\[ "f_m":\text{Efun}(V.D."f_m",V.R."f_m") \]
\[ ); \]

\[ S \equiv \text{Tuple}("c":C,"f":F); \]

Note: Notice that we use indirectly defined indices in projections, i.e. objects which stand right of the dot in a projection and define indirectly an index instead of a concrete index, e.g. \( D."f_1" \) in \( V.D."f_1" \). Formally we didn't introduce this. Only a so called dynamic type checker could now verify the correctness of types. A dynamic type checker must be able to evaluate expressions and projections. We will limit ourselves to index transformations with the help of tuples like \( D \) and \( R \) and the use of parameters ranging over indices (see the definition of constraint on the next page). With this limitation we still are able to build a type checker that can be used before runtime. All our uses of indirectly defined indices can be replaced by a formally correct use of projections.

The corresponding state variable is declared by:

\[ \text{svar:: sv:S;} \]

When in the context list of a lambda expression a parameter \( s \) occurs that ranges over the state space of the database, i.e. \( s \) is typed with \( S \), we would want to use the individual categories and functions of the database state denoted by \( s \). This is realized with the help of projection:

For each \( i \in \{1 \ldots n\} \), the sentence:

\[ c_i := [s:S]s."c"."c_i"; \]

and for each \( j \in \{1 \ldots m\} \), the sentence:

\[ f_j := [s:S]s."f"."f_j"; \]

can be added to our ELISA script.

In general we don't allow all elements of \( S_0(S) \) as database
states. We would like to introduce constraints which restrict this state space. We require that the domain of the function with index \( f \), type \( \text{type}(f, \text{FI}) \), is a subset of the current state of its domain category, i.e. \( \text{D}.f \), and the range of this function is a subset of the current state of its range category, i.e. \( \text{R}.f \). Therefore we introduce the boolean function constraint, that is defined on \( \text{SO}(S) \), which has to be true for \( s' \in \text{SO}(S) \), if \( s' \) is assigned as a new value to the state variable.

\[
\text{constraint} := [s:S] \forall [x:FI] ((\text{dom}(s.f.x) \subset s.c.D.x) \land (\text{rng}(s.f.x) \subset s.c.R.x)) : \text{Bool};
\]

There can be more restrictions on the database state. A precondition and a postcondition must hold before an update of the database. Suppose that \( p_1 \) is a boolean function on \( \text{SO}(S) \), that represents the precondition and \( p_2 \) is a boolean function on \( \text{SO}(S) \), that represents the postcondition. So \( \text{type}(p_1, \text{Fun}(S, \text{Bool})) \) and \( \text{type}(p_2, \text{Fun}(S, \text{Bool})) \). Let \( iv \) denote an input variable such that \( \text{tp}(iv) = I \), \( I \in \text{types} \). Assume that the update is specified by the function \( up \). Thus \( \text{type}(up, \text{Fun}(\text{Row}(I,S), S)) \) must hold. Then we introduce in ELISA a function \( up' \):

\[
up' := [<i,s>:\text{Row}(I,S)] \text{cond}((p_1(s) \land p_2(up(i,s))), up(i,s), s) : S;
\]

The command with name update is specified by:

\[
\text{update} := <sv> <- up'(iv, sv);
\]
3.5 The relational data model

The construction of a database space according to the relational data model is analogous to the construction of the database space of the functional data model. The database space is determined by a three tuple that is called the skeleton of the relational data model:

\[ \langle TN, G, V \rangle \]

where
1: TN is the set of table names, TNcS, TN is finite.
2: G is a function on TN,
   \( \text{At} \in \text{TN} : G(tn) \) is the set of attribute names that belongs to table tn, \( G(tn) \subseteq S \).
3: V is a function valued function on TN such that
   \( \text{At} \in \text{TN} : V(tn) \) is a function from \( G(tn) \) to types.
   \( \text{At} \in \text{TN} : \text{At} \in G(tn) : V(tn)(t) \) is called the object characterization of attribute t of table tn.

The state space of each attribute t from table tn is defined by the set \( S_0(V(tn)(t)) \).
The number of elements of TN is denoted by n, thus n is the number of tables of the database.

\((t_1)_{1 \leq i \leq n} \) is an enumeration of TN.

and
\( A \in \text{TN} : (a_{1,j})_{1 \leq j \leq m(i)} \), \( m(i) = |G(t_i)| \) is an enumeration of set \( G(t_i) \), thus \( a_{1,j} \in G(t_i) \) for \( j \in \{1, \ldots, m(i)\} \).

To define the skeleton in ELISA we must add to the ELISA script:

\[
\begin{align*}
TN & = \{\text{"t}_1\text{"}, \ldots, \text{"t}_n\text{"}\}; \\
G & = \text{Tuple}(\text{"t}_1\text{"} : \{\text{"a}_1,1\text{"}, \ldots, \text{"a}_1,m(1)\text{"}\)}, \\
& \quad \ldots, \\
& \quad \text{"t}_n\text{"} : \{\text{"a}_n,1\text{"}, \ldots, \text{"a}_n,m(n)\text{"}\}) ; \\
V & = \text{Tuple}(\text{"t}_1\text{"} : \text{Tuple}(\text{"a}_1,1\text{"} : ???, \ldots, \text{"a}_1,m(1)\text{"} : ???), \\
& \quad \ldots, \\
& \quad \text{"t}_n\text{"} : \text{Tuple}(\text{"a}_n,1\text{"} : ???, \ldots, \text{"a}_n,m(n)\text{"} : ???)) ;
\end{align*}
\]

Note: Instead of ??? one must write the type of the corresponding attribute.
A new type is used to denote the type (i.e. structure) of the state space.

\[
S \equiv \text{Tuple}("t_1" : \text{Set}(\text{Tuple}("a_1,1" : V."t_1","a_1,1"),
    
    \ldots,
    
    "a_1,m(1)" : V."t_1","a_1,m(1)"
    
),

\)

\ldots,

"t_n" : \text{Set}(\text{Tuple}("a_n,1" : V."t_n","a_n,1"),
    
    \ldots,
    
    "a_n,m(n)" : V."t_n","a_n,m(n)"
    
)

);  

The corresponding state variable is declared by:

\[
s\text{var}:: sv:S;
\]

In the relational algebra it is usual to access the individual tuples of a table and the value of an attribute in a tuple. This is done with the help of projection.

For each \(i \in \{1\ldots n\}\), the sentence:

\[
t_i := [s:S]s."t_i";
\]

and for each \(i \in \{1\ldots n\}\) and \(j \in \{1\ldots m(i)\}\), the sentence:

\[
a_{i,j} := [s:S]t_i"a_{i,j}"
\]

can be added to our ELISA script.

**Note:** The select operator of the relational algebra is easily to construct using the set_generator.

In general, like in the functional data model, we don't allow all elements of \(\text{SO}(S)\) as database states. One may use constraints to restrict the state space. We discern tuple constraints, table constraints and database constraints. Two kinds of constraints are commonly used. A key is a table constraint and a subset requirement is a database constraint.
Keys:
Let $s \in S_0(S)$ and $i \in 1..n$. Thus $s_{t_i}$ is the projection of $s$ on $t_i$. So $s_{t_i}$ is the current state of table $t_i$.

Let $m \in 1..m(i)$. Let $A = \{|a_1|,...,|a_m|\}$ be an enumeration of attribute names such that $A \in 1..m : \text{type}(|a_j|,G_{t_i})$.

Then we call $A$ a key if and only if
for each two tuples from table $s_{t_i}$ it holds that:
the projections of these two tuples on $A$ are equal if and only if
these two tuples are equal.

In ELISA we introduce:

\[
\text{key} := \{\langle s,t,A^\prime\rangle: \text{Row}(S,TN,\text{Set}(G.t))\}
\]

\[
\text{all}(\langle x_1,x_2\rangle: \text{prod}(s.t,s.t))\]
\[
\text{all}(\langle y:A^\prime\rangle(x_1.a = x_2.a))
\]
\[
\text{cond}(\text{empty(setg([y:s_{t_i}]\text{all}([a:A](y.a = x.a)))))
\]

Thus in this particular case $\text{key}(s_{t_i},\{|a_1|,...,|a_m|\})$ must hold. Before each update of the database one must verify that this constraint will be satisfied after the update. If this is done the database state space will be a subset of $\{s \in S_0(S) | \text{key}(s_{t_i},\{|a_1|,...,|a_m|\})\}$.

If we want to select a tuple from table $t_i$ for a given value of the key, i.e. a given tuple with indices from $\{|a_1|,...,|a_m|\}$, we may do this with the help of:

\[
\text{select1} := \{\langle s,x\rangle: \text{Row}(S,Tuple(|a_1|,V_{t_i}.|a_1|,
\]
\[
\ldots,
\]
\[
|a_m|,V_{t_i}.|a_m|
\]
\[
\}
\]

\[
\text{cond}(\text{empty(setg([y:s_{t_i}]\text{all}([a:A](y.a = x.a)))))}
\]

error,
\[
\text{pick(setg([y:s_{t_i}]\text{all}([a:A](y.a = x.a))))}
\]

Note: Notice that we can’t make in ELISA a general construction for "selectl" because we can’t make in ELISA a general construction for tuple types. A tuple type has to be enumerated.

Subset requirements:
Let $s \in S_0(S)$ and $i,j \in \ldots$. Thus $s.t_i$ is the projection of $s$ on $"t_i"$ and $s.t_j$ is the projection of $s$ on $"t_j"$. So $s.t_i$ is the current state of table "t_i" and $s.t_j$ is the current state of table "t_j".
Let $m \in \ldots \min(m(i), m(j))$.
Let $A = \{"a_1", \ldots, "a_m"\}$ be an enumeration of attribute names so that $Ak : type("a_k", G."t_i")$.
Let $B = \{"b_1", \ldots, "b_m"\}$ be an enumeration of attribute names so that $Ak : type("b_k", G."t_j")$.
We introduce an attribute transformation $db$, which is specified by a tuple: $db := \langle "a_1" : "b_1", \ldots, "a_m" : "b_m" \rangle$.

Then we call $db$ a subset requirement if and only if
for each tuple from table $s.t_i$ there exists a tuple in table $s.t_j$ such that for each attribute $t$ from $A$ the projection of the first mentioned tuple on $t$ equals the projection of the second tuple on $db.t$
and $B$ is a key for $"t_j"$, i.e. key($s,t_j$,\{"b_1", \ldots, "b_m"\}) holds.

In ELISA we introduce:
\[ subset\_requirement := \{s:S\} \]
\[
\quad (all([x:s.t_i])
\quad \quad exist([y:s.t_j])
\quad \quad \quad all([s:A](x.a = y.db.a))
\quad )
\]
and
\[
\quad key(s,"t_j",\{"b_1", \ldots, "b_m"\})
\];

Thus in this particular case $subset\_requirement(s)$ must hold. Before each update of the database one must verify that this constraint is satisfied. If this is done the database state space will be a subset of $\{s \in S_0(S) | subset\_requirement(s)\}$.
If we want to select for a given tuple from table "t_i" the corresponding tuple from table "t_j" we may do this with the help of:

\[
\text{select2} := [s:S][x:s."t_i"] \\
\text{pick(setg([y:s."t_j"]all([a:A](y.db.a = x.a))))};
\]

Note: Notice that we can't make in ELISA a general construction for "select2" because we can't make in ELISA a general construction for tuple types. A tuple type has to be enumerated.
4. An example: the jobshop

We will consider as an example some parts of an IS for a jobshop. A jobshop fits on a large class of firms. It fulfills jobs for clients. We consider concrete as well as abstract jobs. A hospital for example "produces" treatments of patients. Each job consists of a number of tasks. A task can be executed only if its parent task is finished. Thus tasks of a job form a forest without cycles. Each task requires for its execution a machine of a certain type. The jobshop can have more machines of the same type. Execution of a task on a machine is called an operation. Tasks are planned with the help of the priority of its job. An IS for a jobshop has to fulfill many functions. It must registrate a finished operation or assign a new operation. A more complex function is the planning of a new job or a new machine. It has to fulfill retrieval and maintenance functions.

We will specify in this chapter a part of such an IS with help of the functional data model. Herefore we will use the construction that is described in section 3.4. Firstly we have to give a definition of the skeleton. When this is done we will describe some examples:

1: Update of the database.
2: Retrieval of information from the database.
3: Generation of updates by the system based on rather complicated computations.
4.1 The construction

The structure diagram of the jobshop is:

```
<table>
<thead>
<tr>
<th>priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
</tr>
<tr>
<td>job</td>
</tr>
<tr>
<td>f2</td>
</tr>
<tr>
<td>task</td>
</tr>
<tr>
<td>f3</td>
</tr>
<tr>
<td>operation</td>
</tr>
<tr>
<td>f6</td>
</tr>
<tr>
<td>machine</td>
</tr>
<tr>
<td>f9</td>
</tr>
<tr>
<td>slowness</td>
</tr>
<tr>
<td>f10</td>
</tr>
</tbody>
</table>
```

Explanation of the functions:

- $f_1$: The priority of the job that is used by the assignment of an operation.
- $f_2$: The job that the task belongs to.
- $f_3$: The parent of a task.
- $f_4$: The machine type that the task requires.
- $f_5$: The time that the task requires on a standard machine.
- $f_6$: The task that is involved by the operation.
- $f_7$: The start time of the operation.
- $f_8$: The end time of the operation.
- $f_9$: The machine that is involved by the operation.
- $f_{10}$: The speed of the machine compared to the standard machine.
- $f_{11}$: The type of the machine.

Note: There are many constraints involved in this IS. We won't consider them.

The skeleton:

```plaintext
CI == {"job","priority","task","duration","machine_type",
       "operation","time","machine","slowness"};
FI == {"1","2","3","4","5","6","7","8","9","10","11"};
```
>|
>|
V == Tuple("job" : Nat, "priority" : Nat, "task" : Nat,
"duration" : Nat, "machine_type" : Nat,
"operation" : Row(Nat,Nat), "time" : Nat,
"machine" : Nat, "slowness" : Nat
);

C, F, S, sv, job,..., slowness, f_j, j € 1..11 and constraint are defined as in section 3.4.

We will construct three commands:
1: An update, i.e. the addition of a complete database state to the current state.
2: Retrieval of the job with the longest minimal rest processing time.
3: Generation of operations for unplanned tasks.
4.2 The update

Our update will consist of the addition of a complete database to a consisting database. This means that from the environment a complete database state may be added. If the environment wants to update only a part of the database, the dialog system could take care that these values are transformed into the correct datastructure, i.e. a complete database of the type $S$.

Firstly we define a function $U$:

\[
U := [{s_1, s_2}: \text{Row}(S, S)]
\]

\[
|<c> : <\text{"job"}, \text{union(job}(s_1), \text{job}(s_2)), \text{"priority"}, \text{union(priority}(s_1), \text{priority}(s_2)), \text{"task"}, \text{union(task}(s_1), \text{task}(s_2)), \text{"duration"}, \text{union(duration}(s_1), \text{duration}(s_2)), \text{"machine_type"}, \text{union(machine_type}(s_1), \text{machine_type}(s_2)), \text{"operation"}, \text{union(operation}(s_1), \text{operation}(s_2)), \text{"time"}, \text{union(time}(s_1), \text{time}(s_2)), \text{"machine"}, \text{union(machine}(s_1), \text{machine}(s_2)), \text{"speed"}, \text{union(speed}(s_1), \text{speed}(s_2)) >|, \text{"f"} : <+\text{"1"}, \text{fununion}(f_1(s_2), f_1(s_1)), \text{"2"}, \text{fununion}(f_2(s_2), f_2(s_1)), \text{"3"}, \text{fununion}(f_3(s_2), f_3(s_1)), \text{"4"}, \text{fununion}(f_4(s_2), f_4(s_1)), \text{"5"}, \text{fununion}(f_5(s_2), f_5(s_1)), \text{"6"}, \text{fununion}(f_6(s_2), f_6(s_1)), \text{"7"}, \text{fununion}(f_7(s_2), f_7(s_1)), \text{"8"}, \text{fununion}(f_8(s_2), f_8(s_1)), \text{"9"}, \text{fununion}(f_9(s_2), f_9(s_1)), \text{"10"}, \text{fununion}(f_{10}(s_2), f_{10}(s_1)), \text{"11"}, \text{fununion}(f_{11}(s_2), f_{11}(s_1)) >| \]

Note: The order of $s_1$ and $s_2$ is important.
In general we want to make some checks before an update of the database state. We would like to verify for example firstly that the predicate constraint will hold after the update. We could also for example add the checks that

1: Each task in the updated database must belong to a job.
2: Each job number in the current database must be different from the job numbers in the database that we want to add to the current database.
3: The left projection of an operation in the updated database must equal the result of function application of \( f_6 \) on this operation.
The right projection of an operation in the updated database must equal the result of function application of \( f_9 \) on this operation.

Herefore we introduce three predicates:

\[
pl := \{ s : S \} \left[ \forall t : \text{task}(s) \exists j : \text{job}(s)(f_2(s)(t) = j) \right] ;
\]

\[
p2 := \{ s_1, s_2 : \text{Row}(S, S) \} \left[ \forall x : \text{job}(s_1)(x \notin \text{job}(s_2)) \right] ;
\]

\[
p3 := \{ s : S \} \left[ \forall t_1, t_2 : \text{operation}(s) \left( (f_6(s)(t_1, t_2) = t_1) \text{ and } (f_9(s)(t_1, t_2) = t_2) \right) \right] ;
\]

Analogously to the function \( up' \) on page 45 we define \( up'' \) as

\[
up'' := \{ s_1, s_2 : \text{Row}(S, S) \} \left[ \text{cond}((\text{constraint}(U(s_1, s_2)) \text{ and } pl(U(s_1, s_2))) \text{ and } (p2(s_1, s_2) \text{ and } p3(U(s_1, s_2)))) \right] ;
\]

\[
U(s_1, s_2),
\]

\[
s_1
\]

The declaration of the input variable becomes:

\[
\text{ivar:: iv : S ;}
\]

The corresponding command becomes:

\[
\text{update ::= <sv> <- up''(sv, iv)}
\]

60
4.3 The retrieval problem

We continue with a retrieval problem. We want to retrieve the minimal rest processing time of a job, beginning with a task of that job. Herefore we introduce the minimal task time of a task. This is defined as the duration of a task multiplied by the lowest slowness of a machine of a suitable type.

\[
mtt := \left\{ \langle s, t \rangle : \text{Row}(S, S."c"."task") \right\} \\
( f_5(s)(t) \\
* \\
\text{minf}([x:\text{setg}([m:\text{machine}(s)]) \\
( f_{11}(s)(m) = f_4(s)(t)) \\
] \\
f_{10}(s)(x) \\
) \\
);
\]

The minimal rest processing time of task \( t \) is the sum of the minimal task time of \( t \) and the maximum of all minimal rest processing times of all the sons of task \( t \).

The sons of task \( t \) are defined as:

\[
sons := \left\{ \langle s, t \rangle : \text{Row}(S, S."c"."task") \right\} \\
\text{setg}([t':\text{task}(s)](f_3(s)(t') = t));
\]

Then we can define the minimal rest processing time as

\[
\text{mrpt} := \left\{ \langle s, t \rangle : \text{Row}(S, S."c"."task") \right\} \\
( mtt(s,t) \\
+ \\
\text{cond}(\text{empty}(sons(s,t)), \\
0, \\
\text{maxf}([x:sons(s,t)]mrpt(s,x)) \\
) \\
);
\]

Suppose we want to retrieve a job with the task that has the longest minimal rest processing time. We use herefore a function that is based on the recursor:

61
lmrpt := \([s:S]\)
    (f_2(s)
        (recursor(pick(task(s)),
            [t_1:task(s)][t_2:S."c"."task"]
                cond(\(\text{mrpt}(s,t_1) \geq \text{mrpt}(s,t_2)\),t_1,t_2)
            )
        )
    )
);
4.4 The scheduler

In this section we will consider the generation of operations for unplanned tasks within some time interval such that:

1: A task won't be executed before its parent task is finished.
2: If a machine becomes idle and there are waiting jobs with unplanned tasks that can be executed on a machine of that type, an operation for that machine is chosen.
3: From all tasks that are candidate for an operation on a free machine, a task belonging to the job with the highest priority is chosen.
4: A task is assigned only once to an operation.

Firstly we define a function that selects for some point in time the set of all candidate operations, i.e. combinations of unplanned tasks and machines such that a machine is free from this moment until the completion time of the operation. Candidates are selected with the help of the following predicates:

The machine is of the type that the task needs,
\[ s_1 := \{<s,t,m>:\text{Row}(S,S."c"."task",S."c"."machine")\} \]
\[ (f_4(s)(t) = f_{11}(s)(m)) ; \]

The task isn't planned yet,
\[ s_2 := \{<s,t>:\text{Row}(S,S."c"."task")\} \]
\[ \text{all}([o:\text{operation}(s)](f_6(s)(o) \neq t)) ; \]

Its parent operation is ready,
\[ s_3 := \{<s,t,time>:\text{Row}(S,S."c"."task",S."c"."time")\} \]
\[ \text{all}([o:\text{operation}(s)]((f_6(s)(o) \neq f_3(s)(t)) \]
\[ \text{or} \]
\[ (f_8(s)(o) \leq time) \]
\[ ) \]

The task can be performed within the idle period of the machine,
\[ s_4 := \{<s,t,m,time> : \text{Row}(S,S."c"."task",S."c"."machine",S."c"."time") \} \]
all([o:operation(s)])
   ((f_9(s)(o) = m)
    ->
    ((f_8(s)(o) <= time)
     or
     (f_7(s)(o) >= compl(s,t,m,time))
    )
   )
);

Where compl, i.e. the completion time of an operation starting at time, is defined by:
compl := [<s,t,m,time>
             :
             Row(S,S."c"."task",S."c"."machine",S."c"."time")
             ]
       ((f_5(s)(t) * f_10(s)(m)) + time);

The selected candidates are:
candidates := [<s,time>:Row(S,S."c"."time")]
             setg([<t,m>:prod(task(s),machine(s))]
                 ((s_1(s,t,m) and s_2(s,t))
                  and
                  (s_3(s,t,time) and s_4(s,t,m,time))
                 )
             );

There can be more tasks for one machine or more machines for one task among the candidates. We select one task from the candidate list with the help of the priority of jobs. It is a decision of another programmer if he wants to select with a criterion among the candidate operations belonging to jobs with the same priority (See section 4.1.1). We don’t do this now.

choice := [St:Set(S."c"."operation")]
         pick(setg([<t_1,m_1>:St]
             all([<t_2,m_2>:St]
                 (f_2(s)(f_1(s)(t_1))
                  >=
                  f_2(s)(f_1(s)(t_2))
                 )
             )
         );
Such a chosen candidate operation can be added to the database. The involved machine and task must be removed from the candidate list and we can choose another candidate from the list that rests.

```plaintext
selection := [St:Set(S."c"."operation")]
  cond(empty(St),
    {},
    ins(selection(
      setg([x:St]
        ((left(x) # left(choice(St)))
         and
         (right(x) # right(choice(St)))
       )
     )
    ),
    choice(St)
  )
)

Now we can add the selected candidates to the database:
newc := [<s,time>:Row(S,S."c"."time")]
|"job" : job(s),
 "priority" : priority(s),
 "task" : task(s),
 "duration" : duration(s),
 "machine_type" : machine_type(s),
 "operation" : union(operation(s),
    selection(candidates(s,time))
  )
 "time": union(ins(time(s),time),
    rng([<t,m>:selection(candidates(s,time))]
     compl(s,t,m,time)
   )
 ),
 "machine" : machine(s),
 "speed" : speed(s)
 >]
```
newf := [<s,time>:Row(S,S,"c","time")]
|<"f_1" : f_1(s),
  "f_2" : f_2(s),
  "f_3" : f_3(s),
  "f_4" : f_4(s),
  "f_5" : f_5(s),
  "f_6" : fununion([<t,m>:selection(candidates(s,time))])t, f_6(s)
 ),
  "f_7" : fununion([<t,m>:selection(candidates(s,time))]) time, f_7(s)
 ),
  "f_8" : fununion([<t,m>:selection(candidates(s,time))]) compl(s,t,m,time), f_8(s)
 ),
  "f_9" : fununion([<t,m>:selection(candidates(s,time))]) m, f_9(s)
 ),
  "f_{10}" : f_{10}(s),
  "f_{11}" : f_{11}(s)
 ];

If for a time point all possible operations are created then we have to reset our clock. We try to determine the first moment after the time point we were considering when a machine becomes idle. To describe this we define the function idle and event:

idle := [<s,time,end>:Row(S,S,"c","time",S,"c","time")]
setg([t:time(s)]
exist([o:operation(s)]((f_8(s)(o) = t)
and
((t > time) and (t <= end)))
)
);

66
event := [<s,time,end>:Row(S,S."c"."time",S."c"."time")]
   cond(empty(idle(s,time,end)),
       end,
       minf([t:idle(s,time,end)]t)
   );

Finally we can define the scheduler which plans all possible unplanned operations during a time interval.

scheduler := [<s,begin,end>:Row(S,S."c"."time",S."c"."time")]
   cond((begin >= end),
       s,
       scheduler(<"c": newc(s,begin),
                  "f": newf(s,begin)
             >,
          event(<"c": newc(s,begin),
                 "f": newf(s,begin)
             >,
                 begin,
                 end
          ),
      end
   )
);
wants to do this, he needs some knowledge of programming in ELISA. And moreover, a change of an ELISA script is considered as a change of the corresponding IS (see section 2.7). We need a more flexible way to solve this problem. One may consider a criterion as a predicate on a set of pairs of possibilities, that is true if we prefer the first possibility and false if we cannot decide which one we prefer. A predicate is a boolean function. The strength of ELISA is that a function can be used as an argument of a command. Thus a predicate or a criterion can be an argument of the scheduler. If we do this, each planner may use his own selection criterion without changing the IS.

Let $s$ be a type. Suppose that a finite set of possibilities is denoted by $S$ and the type($S$, set($s$)) holds. A criterion $c$ on pairs of possibilities is a boolean function that is true if the first possibility is the best and false if we cannot decide which one is the best. Thus type($c$, Fun(Row($s$, $s$), Bool)) holds. We call a criterion $c$ complete if and only if

- $\forall x, y, z \in S$: $(c(x, y) \text{ and } c(y, z) \Rightarrow c(x, z))$
- $\forall x, y \in S$: $(c(x, y) \text{ or } c(y, x) \text{ or } x = y)$
- $\forall x \in S$: not($c(x, x)$)

A complete criterion determines a complete order on $S$. A complete order is necessary to make a unique selection among the possibilities of set $S$. We suppose now that the criterions a planner uses are complete.

In the scheduler each member of the set of candidates at a time point consists of a task and a machine. Sometimes at a time point a task can be executed on more than one machine or at a time point there can be more tasks which can be executed on the same machine. Thus we need two criterions (complete!) for these two kinds of possibilities. One to select the best task and one to select the best machine. We will denote the first criterion by "left" and the second by "right". Left and right are arguments of the scheduler:

```
ivar:: leftv : Fun(Row(S."c"."task",S."c"."task"),Bool),
         rightv : Fun(Row(S."c"."machine",S."c"."machine"),Bool);
plan ::= <sv> <- scheduler(sv,start,stop,leftv,rightv);
```
We leave it to the reader to add left and right as parameters in the definitions and as arguments of the function applications of the scheduler, `newc` and `newf`.

The selection operator becomes completely different. Firstly we define a predicate `accept`. Accept determines recursively if a candidate operation is acceptable or not acceptable. A candidate is acceptable if all candidate operations for the same task or same machine, which are better with respect to our criterions, aren't acceptable:

\[
\text{accept} := \left[\text{St}, \text{left}, \text{right}, \langle t_1, m_1 \rangle \right]
\]

\[
: \text{Row}(\text{Set}(\text{S."c"."operaton"}),
\text{Fun}(\text{Row}(\text{S."c"."task"}, \text{S."c"."task"}), \text{Bool}),
\text{Fun}(\text{Row}(\text{S."c"."machine"}, \text{S."c"."machine"}), \text{Bool}),
\text{S."c"."operation"}
)\]

\[
\text{all}([\langle t_2, m_2 \rangle: \text{St}]
\]

\[
(((t_1 = t_2) \text{ and } \text{left}(m_2, m_1))
\]

\[
\text{or}
\]

\[
((m_1 = m_2) \text{ and } \text{right}(t_2, t_1))
\]

\[
\rightarrow
\]

\[
\text{not(accept(St, left, right, } \langle t_2, m_2 \rangle)\)
\]

\[
);\]

Note: Try to prove with the help of the completeness of the criterions and the finiteness of St at each time point, that the number of elementary steps to evaluate a function application of `accept`, is finite.

Selection becomes now:

\[
\text{selection} := \left[\text{s}, \text{time}, \text{left}, \text{right} \right]
\]

\[
: \text{Row}(\text{S},
\text{S."c"."time"},
\text{Fun}(\text{Row}(\text{S."c"."task"}, \text{S."c"."task"}), \text{Bool}),
\text{Fun}(\text{Row}(\text{S."c"."machine"}, \text{S."c"."machine"}), \text{Bool}),
)\]

\[
\text{setg}([\text{o:candidate(s, time)}]
\]
accept(candidate(s,time),left,right,o)
);

Note 1: Try to formulate a few criterions for the scheduler.

2: Try to develop a selection operator based on criterions
for which the second and third property of completeness
don't hold. Thus such criterions define only a partial
order on St.
Tip: Transform an incomplete criterion into a
corresponding complete criterion.

3: The criterion to select the best task among two is
independent of the involved machine and vice versa.
Try to construct a selection operator based on complete
criterions which include such an interweaving. In
literature this is known as the stable marriage problem.
(See [10])
The syntax of ELISA

\[\text{types ::= basic types | type variables | enumerated types | new types | Fun( types , types ) } |  \\ 
Efun( types , types ) |  \\ 
Set( types ) |  \\ 
Row( type list ) | Row() |  \\ 
Tuple( tuple list ) | Tuple() |  \\ 
List( types ) |  \\ 
types.index |  \\ 
\]

basic types ::= Nat | Int | Real | String | Bool  

type list ::= types | types , type list  

tuple list ::= index : types | index : types , tuple list  

type variables ::= * | * type variables  

new types ::= new types  

enumerated types ::= { | index list | }  

index list ::= index | index , index list  

index ∈ 'W'  

eset ::= {} | { object list }  

elist ::= [[]] | [[ object list ]]  

erow ::= <> | < object list >  

etuple ::= <> | <| indexed object list |>  

object list ::= objects | objects , object list  

indexed object list ::= index : objects |  

index : objects , indexed object list  

lambda expressions ::= [ P dom ] objects  

P ::= parameters | < P list >  

P list ::= P | P . P list  

dom ::= eps | : objects | : types  

parameters ∈ parameters  

projection ::= objects . index  

expression ::= objects ( object list )  

objects ::= PBO | eset | elist | erow | etuple | lambda expressions |  

expression | projection | index | parameters | names |  

bo names ( objects )  

names ∈ names  

bo names ∈ bo_names
axiom ::= type( names, types )

var decl ::= sym :: var type list
sym ::= ivar|ovar|svar
var type list ::= variables : types |
variables : types, var type list

command ::= < var list > <- names ( var list )
var list ::= variables|variables , var list
variables e variables

definition ::= names := objects typing |
bo names := BO : types | new types == types | command names ::= command
typing ::= eps| : types
command names e command_names

ELISA script ::= sentence @/ sentence ; ELISA script
sentence ::= axiom|var decl|definition
References


4 Cardelli, L., "A semantics of Multiple Inheritance", AT&T Bell Laboratories, Murray Hill, New Jersey 07974.


In this series appeared:

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>85/01</td>
<td>R.H. Mak</td>
<td>The Formal Specification and Derivation of CMOS-circuits</td>
</tr>
<tr>
<td>85/02</td>
<td>W.M.C.J. van Overveld</td>
<td>On arithmetic operations with M-out-of-N-codes</td>
</tr>
<tr>
<td>85/03</td>
<td>W.J.M. Lemmens</td>
<td>Use of a Computer for Evaluation of Flow Films</td>
</tr>
<tr>
<td>85/04</td>
<td>T.Verhoeff, H.M.J.L. Schols</td>
<td>Delay insensitive Directed Trace Structures Satisfy the Foam Rubber Wrapper Postulate</td>
</tr>
<tr>
<td>86/01</td>
<td>R. 'Koymans</td>
<td>Specifying Message Passing and Real-time Systems</td>
</tr>
</tbody>
</table>