Appendix A. Syntax and Semantics of the PHLQA1 Languages.

1. Introduction.

The PHLQA1 languages are typed languages. A system of semantic types plays an important role in their syntax and in their semantics. A semantic type may be an atomic type, or a compound type constructed from atomic types.

The syntax of a language defines which expressions belong to it. The definition of the expressions of a PHLQA1 language consists of two parts:
- a specification of the primitive expressions (the terms) of the language.
- a recursive definition of complex expressions in terms of simpler component expressions.

The syntax also assigns a semantic type to every expression of the language. The rules which construct compound expressions out of simpler component expressions impose conditions on the semantic types of the component expressions.

The semantics of a language specifies how interpretations of the language are defined. An interpretation of a PHLQA1 language is defined in two steps:
1. To every atomic type, a set of entities (called a domain) is assigned. (The type system has semantic rules which define the domain of any type in terms of the domains of the atomic types).
2. To every term, a denotation is assigned; this denotation must be an element of the domain of the type of the term. The semantic rules of the language define the denotation of any expression in terms of the denotations of the terms occurring in it.

2. The Type System of the PHLQA1 Languages.

A PHLQA1 language is defined in two steps. First we define a type system: a class of semantic types and a function (called components) which operates on types. The type system is then used in the definition of the class of expressions of the language.

In the present subsection we give a syntactic definition of the types of the PHLQA1 languages, and an informal discussion of their semantic aspects. The types are defined by the following rules:
1. Every atomic type is a type. There are two kinds of atomic types: formal ones and descriptive ones. The formal atomic types are truthvalue, integer, real and string. They occur in the type system of every PHLQA1
language. The descriptive atomic types are specified separately for every particular PHILQA1 language.

2. If $\alpha$ is a type, then $S(\alpha)$, $B(\alpha)$, $L(\alpha)$, $F(\alpha)$ are types.
3. If $\alpha_1$, ..., $\alpha_n$ are types then $\langle \alpha_1, \ldots, \alpha_n \rangle$ is a type.
4. If $\alpha$ and $\beta$ are types, then $(\alpha \rightarrow \beta)$ and $(\alpha \rightarrow \beta)$ are types.
5. If $\alpha$ is a type, AMT ($\alpha$) is a type.
6. If $i$ is an integer or a string and $\alpha$ is a type, then $ID_i(\alpha)$ is a type.
7. If $\alpha_1$, ..., $\alpha_n$ are types, then $\cup (\alpha_1, \ldots, \alpha_n)$ is a type.

In the definition of the expressions of the language we use a function, called COMPONENTS, which assigns to any type a set of types (its "component types") as follows:

$$\text{COMPONENTS} (\alpha) = \begin{cases} \text{def} & \text{if } \alpha \text{ has the form } \cup (\alpha_1, \ldots, \alpha_n) \\ \text{then } \text{COMPONENTS} (\alpha_1) \cup \ldots \cup \text{COMPONENTS} (\alpha_n) & \text{else if } \alpha \text{ has the form } \langle \alpha_1, \ldots, \alpha_n \rangle \\ \{ \langle \varphi_1, \ldots, \varphi_n \rangle \mid \forall i : \varphi_i \in \text{COMPONENTS} (\alpha_i) \} & \text{else } \{ \alpha \} \end{cases}$$

The role of the semantic types in the semantics of the language will be precisely described in section 4. But in order to give an intuitive idea about the use of the types, we already give an informal description at this point.

An interpretation of the language assigns to every type a set of entities as its domain. The domains of the atomic types are disjoint sets of individual entities. Formal atomic types have the same domain in every interpretation of the language, whereas descriptive atomic types may have different domains assigned to them in different interpretations.

Types of the form $S(\alpha)$, $B(\alpha)$, $F(\alpha)$ or $L(\alpha)$ have domains consisting of compound entities (sets, bags, files and lists, respectively) which consist of elements of the domain of $\alpha$. (Bags are unordered collections where elements can have multiple occurrences. Files are ordered sets. Lists are ordered and allow multiple occurrences). Types of the form $\langle \alpha_1, \ldots, \alpha_n \rangle$ have domains consisting of $n$-tuples whose first element is from the domain of $\alpha_1$, ..., and whose $n^{th}$ element is from the domain of $\alpha_n$. Types of the form $(\alpha \rightarrow \beta)$ have domains consisting of total functions from the domain of $\alpha$ into the domain of $\beta$; types of the form $(\alpha \rightarrow \beta)$ have domains consisting of partial functions from the domain of $\alpha$ into the domain of $\beta$.

Types of the form AMT ($\alpha$) have domains consisting of "amounts", i.e. pairs consisting of a number and an element of the domain of $\alpha$ which is used as a "unit". A type of the form $ID_i(\alpha)$ has a domain consisting of specially constructed objects, which have a one-to-one correspondence to the objects in the domain of $\alpha$. The domain of $\cup (\alpha_1, \ldots, \alpha_n)$ is the union of the domains of $\alpha_1$, ..., $\alpha_n$. 


The type system is used in the syntax which defines the expressions of a PHLIQA1 language (see section 3). Such a syntax also assigns a semantic type to each of the expressions of the language. Syntax and semantics are defined in such a way that for every interpretation of the language the denotation of any expression is an element of the domain of the type of expression.

3. The Definition of the Expressions of a PHLIQA1 Language.

The definition of the expressions of a PHLIQA1 language consists of two parts:
- a specification of the primitive expressions (the terms) of the language.
- a recursive definition of complex expressions in terms of simpler component expressions.

There are two kinds of terms: constants and variables.

There are two kinds of constants: formal constants and descriptive constants. The formal constants are the same in all the PHLIQA1 languages. They stand for logical or mathematical notions, and receive the same standard denotation for every interpretation of the language. The formal constants are:
- TRUE and FALSE, both with type truthvalue.
- the decimal representations of the integers, with type integer,
- all alphanumeric strings between quotes, with type string.

The descriptive constants are different for every PHLIQA1 language. For every atomic type $\alpha$, there is a constant $GS_{\alpha}$.

For every type, there are countably many variables with this type. Every expression of a PHLIQA1 language is either a term (a constant or a variable) or a complex expression. Every complex expression has the form $b(se{{l}_{1}}; e_{1}, \ldots, se{{l}_{n}}; e_{n})$, where $e_{1}, \ldots, e_{n}$ are expressions of the language. $b$ is called the branching category of the expression; $se{{l}_{1}}, \ldots, se{{l}_{n}}$ are the selectors belonging to this branching category.

The recursive definition of the complex expressions of a PHLIQA1 language consists of a number of rules. Each of these rules has the following form: If $e_{1}, \ldots, e_{n}$ are expressions of the language, and their types fulfill certain conditions, then $b(se{{l}_{1}}; e_{1}, \ldots, se{{l}_{n}}; e_{n})$ is also an expression of the language, and its type can be derived from the types of $e_{1}, \ldots, e_{n}$ in a specific way.

PHLIQA1 expressions can also be represented as trees, with constants and variables as terminals, branching categories on the non-terminal nodes, and selectors as labels on the arcs. Such a tree representation corresponds closely to the internal computer representation of the expression. Describing the
expressions as trees of this kind \(^1\) also makes it easier to describe the syntax and the semantics of the languages in a completely formal way – because the trees explicitly display their syntactic structure, and their semantic structure coincides with that. (In section 4 we shall see that the recursive definition of the denotation of an expression parallels the syntactic definition of the expressions given below).

We shall now define a fragment of the PHLIQA1 languages, by giving some (but not all) of the syntactic rules. \(^2\) We omit the elements of the definition which are different for the different languages: the specification of the descriptive atomic types and the descriptive terms. To make the presentation not unnecessarily cumbersome, the type requirements for the sub-expressions of a branching are in many cases chosen to be more simple than in the actual PHLIQA1 system. Especially, the possibility of subexpressions denoting ”collections” other than sets and \(n\)-tuples is largely ignored. To shorten the formulations, we use, for instance, ”\(A\) has type \(\alpha\)” for ”\(A\) is an expression of the language with type \(\alpha\)”.

First of all, we have the core operations of the \(\lambda\)-calculus, \(\lambda\)-abstraction and function-application:

1. If \(x\) is a variable of type \(\alpha\) and \(B\) has type \(\beta\), **abstraction** (\(\text{var: } x, \text{ descr: } B\)) has type \((\alpha \rightarrow \beta)\). The expression may be abbreviated as \((\lambda x:B)\).
   (Because PHLIQA1 expressions may be denotation-less, under certain interpretations, \(\lambda\)-abstraction results in a *partial* function).

2. If \(F\) has type \((\alpha \rightarrow \beta)\) or \((\alpha \rightarrow \beta)\) and \(E\) has any type \(\varepsilon\), then **application** (\(\text{fun: } F, \text{ arg: } E\)) has type \(\beta\). The expression may be abbreviated as \(F (E)\).

Secondly, we have quantification and similar operations. They do not use variables in an explicit way: the only branching category which introduces variables is the \(\lambda\)-abstraction.

3. If \(A\) has type \(\varepsilon (\varepsilon)\) or \(B(\varepsilon)\), \(P\) has type \((\alpha \rightarrow \text{truthvalue})\) or \((\alpha \rightarrow \text{truthvalue})\), and \(F\) has type \((\alpha \rightarrow \beta)\) or \((\alpha \rightarrow \beta)\), then:
   - **universal-quantification** (\(\text{forall: } A, \text{ holds: } P\)) and
   - **existential-quantification** (\(\text{forsome: } A, \text{ holds: } P\)) have type \(\text{truthvalue}\),
   - **selection** (\(\text{head: } A, \text{ mod: } P\)) has type \(\varepsilon (\varepsilon)\),
   - **iteration** (\(\text{for: } A, \text{ apply: } F\)) has type \(B (\beta)\).

\(^1\) Landsbergen and Scha (1977) actually do this.

\(^2\) The corresponding semantic rules are given later, in section 4.
Logical operations:

4. If $P$ and $Q$ have type $\text{truthvalue}$, then
   - non (arg: $P$), also written as $\neg P$,
   - conj (1: $P$, 2: $Q$), also written as $P \land Q$,
   - disj (1: $P$, 2: $Q$), also written as $P \lor Q$,
   have type $\text{truthvalue}$.

Some operations which are useful to form expressions which denote the domains of compound types (see Chapter V, section 3):

5. If $A$ has type $S(\varepsilon)$ then power (arg: $A$) has type $S(S(\varepsilon))$, bags (arg: $A$) has type $S(B(\varepsilon))$, lists (arg: $A$) has type $S(L(\varepsilon))$, files (arg: $A$) has type $S(F(\varepsilon))$.

6. If $A_1$, ..., $A_n$ has type $S(\varepsilon_i)$, then:
   - cartesian-product (1: $A_1$, ..., 2: $A_n$), also written as $A_1 \times \ldots \times A_n$, has type $S(\langle \varepsilon_1, \ldots, \varepsilon_n \rangle)$, and union (1: $A_1$, ..., 2: $A_n$), also written as $\cup (A_1, \ldots, A_n)$, has type $S(\varepsilon_1, \ldots, \varepsilon_n)$.

7. If $D$ has type $S(\alpha)$ and $R$ has type $S(\beta)$, then
   - functions, (domain: $D$, range: $R$) has type $S((\alpha \rightarrow \beta))$ and
   - functions$^p$, (domain: $D$, range: $R$) has type $S((\alpha \rightarrow \beta))$.

Next we mention various other operations.

8. If $A$ has type $S(\alpha)$ then Count (arg: $A$) has type $\text{integer}$.

9. If $D$ has type $S(\alpha)$ and $E$ is an expression, then elt-of (E, D) has type truthvalue. It is also written as $E \in D$.

10. If $A$ has type $S(\alpha)$ then unel (arg: $A$) has type $\alpha$.

11. If $T$ has type $\langle \varepsilon_1, \ldots, \varepsilon_n \rangle$ then, for any positive integer $i \leq n$:
    - el$_i$ (arg: $T$), also written as $T[i]$, has type $\varepsilon_i$.

12. If $F_1$ has type $(\alpha_1 \rightarrow \beta_1)$ or $(\alpha_1 \rightarrow \beta_1)$, ..., $F_n$ has type $(\alpha_n \rightarrow \beta_n)$ or $(\alpha_n \rightarrow \beta_n)$, then function-choice (1: $F_1$, ..., 2: $F_n$) has type $(\cup (\alpha_1, \ldots, \alpha_n) \rightarrow \cup (\beta_1, \ldots, \beta_n))$.

13. If $E_1$ and $E_2$ are expressions, equal (1: $E_1$, 2: $E_2$) has type truthvalue. It is also written as $E_1 = E_2$.

14. If $N$ has a type $\alpha$ such that components $(\alpha) \subseteq \{\text{integer, real}\}$ and $E$ has any type $\varepsilon$, then amount$_{\alpha}$ (num: $N$, unit: $E$) has type AMT ($\varepsilon$).

15. If $N$ has a type $\alpha$ such that components $(\alpha) \subseteq \{\text{integer, real}\}$ and $A$ has a type of the form AMT ($\gamma$), then amount$_{\alpha}$ (num: $N$, amount: $A$) has type AMT ($\gamma$).

16. If $B$ has type $\langle \alpha, \beta \rangle$ then function (pairs: $B$) has type $(\alpha \rightarrow \beta)$.

17. If $A$ has type $B$ $(\alpha)$ then bag-to-set (arg: $A$) has type $S(\alpha)$. 
18. If \( E_1, \ldots, E_n \) have types \( \varepsilon_1, \ldots, \varepsilon_n \), respectively, then
\[
\text{tuple}_n (i: E_1, \ldots, n: E_n) \text{ also written as } \langle E_1, \ldots, E_n \rangle \hspace{1pt} \text{has type } \langle \varepsilon_1, \ldots, \varepsilon_n \rangle
\]
19. If for some \( n \) \( NT \) has type \( \langle \varepsilon_1, \ldots, \varepsilon_n \rangle \) then \textbf{bag} (tuple: \( NT \)) has type
\[
\text{B} ( \cup (\varepsilon_1, \ldots, \varepsilon_n)), \hspace{1pt} \text{set} (\text{tuple}: NT) \hspace{1pt} \text{has type } S (\cup \varepsilon_1, \ldots, \varepsilon_n)), \hspace{1pt} \text{list} (\text{tuple}: NT) \hspace{1pt} \text{has type } L (\cup (\varepsilon_1, \ldots, \varepsilon_n)), \hspace{1pt} \text{file} (\text{tuple}: NT) \hspace{1pt} \text{has type}
\[
F (\cup (\varepsilon_1, \ldots, \varepsilon_n)).
\]
20. If for some \( n \) \( TT \) has type \( \langle \langle \alpha_1, \beta_1 \rangle, \ldots, \langle \alpha_n, \beta_n \rangle \rangle \) then \textbf{function} (tuple: \( TT \)) has type \( (\cup (\alpha_1, \ldots, \alpha_n) \rightarrow \cup (\beta_1, \ldots, \beta_n)) \).
21. If \( TV \) has type \textbf{truthvalue} and \( E \) has type \( \varepsilon \) then \textbf{cond} (if: \( TV \), then: \( E \)) has type \( \varepsilon \).
22. If \( E \) has type \( \varepsilon \) and \( i \) is an integer, then \textbf{id}_i (arg: \( E \)) has type \( \text{ID}_i (\varepsilon) \).
23. If for some integer \( i \) and some type \( \gamma \), \( A \) has the type \( \text{ID}_i (\gamma) \), then
\[
\textbf{rid} (\text{arg}: \( A \)) \hspace{1pt} \text{has type } \gamma.
\]
24. If \( P \) has type \( (\alpha \rightarrow \textbf{truthvalue}) \) or \( (\alpha \rightarrow \textbf{truthvalue}) \) and \( E \) has any type \( \varepsilon \), then \textbf{presup} (presup: \( P \), descr: \( E \)) has type \( \varepsilon \).

If an expression is meant to be read by humans, we may use abbreviated notations instead of the "official" ones. Some abbreviations were already introduced above. Some others:
- If the selectors identify the branching category, the branching category may be left out. (For instance (forall: \( S \), holds: \( P \)), instead of:
  \[
  \textbf{universal-quantification} (\text{forall}: S, \text{holds}: P).
  \]
- The selectors 1-\( n \) may be left out.
- If a branching category has only one selector, the selector may be left out.

The rules 1-24 above define the type of any complex expression in terms of the types of its immediate sub-expressions; so eventually the type of any expression is defined in terms of the types of the terminals.

For any specific language, the types of the terminals are also given. We may therefore assume the existence of a function \textbf{type}, applicable to any legitimate expression of a PHLIQA1 language, and delivering the type of that expression.

The occurrence of the variable \( x \) as the \( \lambda \)-variable in an expression of the form \( (\lambda x: A) \) is called the defining occurrence of \( x \). The expression \( A \) is the scope of this occurrence. If an occurrence of a variable is not a defining one and it is not within the scope of a defining occurrence of the same variable, the occurrence is called free. A closed expression is defined as an expression that does not contain free occurrences of variables.

4. The Semantics of the PHLIQA1 Languages.

Because the languages are many-sorted, assigning an interpretation to a language consists of two steps:
1. To every atomic type, a set of entities (a \textit{domain}) is assigned. By virtue of
the semantic rules of the type system (see below), this defines the domain of any type.

2. To every term, a denotation is assigned; this denotation must be an element of the domain of the type of the term. By virtue of the semantic rules of the language, this defines the denotation of any expression.

The semantic rules assume the following distinct primitive objects: true and false, the integers, the reals, the alphanumerical strings, ID and amount. Sets, bags, files, lists, functions and n-tuples are assumed to be distinct kinds of mathematical entities. (E.g.: a function can never be equal to a set of pairs).

The semantic rules of the type system.

A type interpretation $I_{\text{atom}}$ assigns domains to the atomic types. $I_{\text{atom}}$ must fulfill the following conditions:

a. for any atomic type $\alpha$, $I_{\text{atom}}(\alpha)$ is a set of individuals.

b. for any two distinct atomic types $\alpha$ and $\beta$, $I_{\text{atom}}(\alpha)$ and $I_{\text{atom}}(\beta)$ are disjoint.

c. the domain of truthvalue is $\{\text{true}, \text{false}\}$;

the domain of integer is the set of integers;

the domain of real is the set of reals;

the domain of string is the set of alphanumerical strings.

Now we define a function $\text{DOM}$ which assigns a domain (a set of entities) to any type. $\text{DOM}$ is recursively defined, by means of the following rules:

1. For any atomic type $\alpha$, $\text{DOM}(\alpha) = I_{\text{atom}}(\alpha)$.

2. $\text{DOM}(\text{S}(\beta))$ is the set of all subsets of $\text{DOM}(\beta)$.
   $\text{DOM}(\text{B}(\beta))$ is the set of all bags whose elements are from $\text{DOM}(\beta)$.
   $\text{DOM}(\text{F}(\beta))$ is the set of all files whose elements are from $\text{DOM}(\beta)$.
   $\text{DOM}(\text{L}(\beta))$ is the set of all lists whose elements are from $\text{DOM}(\beta)$.

3. $\text{DOM}(<\alpha_1, \ldots, \alpha_n>)$ is the set of all $n$-tuples $<A_1, \ldots, A_n>$ such that $A_1 \in \text{DOM}(\alpha_1), \ldots, A_n \in \text{DOM}(\alpha_n)$.

4. $\text{DOM}((\alpha \rightarrow \beta))$ is the set of all partial functions from the domain of $\alpha$ into the domain of $\beta$.
   $\text{DOM} ((\alpha \rightarrow \beta))$ is the set of all total functions from the domain of $\alpha$ into the domain of $\beta$.

5. $\text{DOM}(\text{AMT}(\alpha))$ is the set $\{<\text{amount}, <x, y>> | (x \in \text{DOM}(\text{integer}) \lor x \in \text{DOM}(\text{real})) \land y \in \text{DOM}(\alpha)\}$

6. $\text{DOM}(\text{ID}_1(\alpha))$ is the set $\{<\text{id}, \overline{d}, y>> | y \in \text{DOM}(\alpha)\}$

7. $\text{DOM} (\cup (\alpha_1, \ldots, \alpha_n))$ is the union of $\text{DOM}(\alpha_1), \ldots, \text{DOM}(\alpha_n)$. 
The semantic rules of the language.

Let a type interpretation $I_{\text{atom}}$ be given, then a term interpretation $I_{\text{term}}$ can be specified which assigns a denotation to each of the constants and variables of the language. $I_{\text{term}}$ must fulfill the following conditions:

a. the denotation of every term is an element of the domain of its type,

b. the formal constants have their usual standard denotations,

c. for any atomic type $\alpha$, the denotation of $G_{\alpha}$ is the domain of $\alpha$.

(Note that an interpretation also assigns denotations to variables – we do not use a separate value-assignment function for the variables. The semantic rules are such that the denotation of a closed expression does not depend on the assignment of denotations to variables.)

We now give a recursive definition of the denotation $D[E]$ of any expression $E$.\(^3\) If $E$ is a term, $D[E] = I_{\text{term}}(E)$. If $E$ is a complex expression $b$ ($sel_1: E_1, \ldots, sel_n: E_n$), its denotation is defined as follows:

- If $E_1, E_2, \ldots, E_n$ does not have a denotation, $E$ does not have a denotation.

- If $E_1, \ldots, E_n$ all have a denotation, $D[E]$ is defined by the following rules:\(^4\):

1. $D[(\lambda x: A)]$ is the partial function that assigns to any element $e$ in the domain of the type of $x$ the denotation (if there is any) that $A$ has for the term-interpretation $I'_{\text{term}}$ which only differs from $I_{\text{term}}$ in that $x$ denotes $e$.

2. $D[\text{application (fun: F, arg: A)}]$ is the result of applying $D[F]$ to $D[A]$. If $D[A]$ is not an element of the domain of $D[F]$, the expression `application (fun: F, arg: A)` has no denotation.

3. $D[\text{universal-quantification (forall: S, holds: P)}]$ is true if the application of $D[P]$ to all elements of $D[S]$ yields true, and false otherwise. $D[\text{existential-quantification (forsome: S, holds: P)}]$ is true if the application of $D[P]$ to some element of $D[S]$ yields true, and false otherwise.

$D[\text{selection (head: S, mod: P)}]$ is the subset of those elements of $D[S]$ for which $D[P]$ yields true.

$D[\text{iteration (for: S, apply: F)}]$ is the bag of all entities which $D[F]$ yields when applied to each element of $D[S]$.

4. $D[\text{---TV}]$ is true if $D[TV]$ is false; otherwise it is false.

$D[\text{conj (I: A_1, 2: A_2)}]$ is true if both $D[A_1]$ and $D[A_2]$ are true; otherwise it is false.

$D[\text{disj (I: A_1, 2: A_2)}]$ is false if neither $D[A_1]$ nor $D[A_2]$ are true; otherwise it is true.

\(^3\) A, B, E, F, N, P, S, TV, NT, TT and their indexed variants, are meta-variables which stand for PHLIQAl-expressions.

\(^4\) These rules run parallel to the syntax rules in section 3.
5. $D[\text{power} \ (\text{arg}: S)]$ is the set of subsets of $D[S]$.

$D[\text{bags} \ (\text{arg}: A)]$ is the set of all bags whose elements are from $D[A]$.

$D[\text{lists} \ (\text{arg}: A)]$ is the set of all lists whose elements are from $D[A]$.

$D[\text{files} \ (\text{arg}: A)]$ is the set of all files whose elements are from $D[A]$.

6. $D[\text{cartesian-product} \ (I: S_I, \ldots, n: S_n)]$ is the set of all $n$-tuples whose first element belongs to $D[S_I], \ldots,$ and whose $n^{\text{th}}$ element belongs to $D[S_n]$;

$D[\text{union} \ (I: S_I, \ldots, n: S)]$ is the union of the sets $D[S_I], \ldots, D[S_n]$.

7. $D[\text{functions}, \ (\text{domain}: A, \ \text{range}: B)]$ is the set of total functions which map $D[A]$ into $D[B]$.

$D[\text{functions}_p \ (\text{domain}: A, \ \text{range}: B)]$ is the set of functions which map a subset of $D[A]$ into $D[B]$.

8. $D[\text{Count} \ (\text{arg}: S)]$ is the cardinality of $D[S]$.

9. $D[E \in A]$ is $\text{true}$ if $D[E]$ is an element of $D[A]$ and $\text{false}$ if $D[E]$ is not an element of $D[A]$.

10. If $D[S]$ is a one-element set, $D[\text{unel} \ (\text{set}: S)]$ is its element; otherwise, 'unel (set: S)' does not have a denotation.

11. $D[\text{el}_i \ (\text{arg}: NT)]$ is the $i$-th element of the $n$-tuple $D[NT]$.

12. $D[\text{function-choice} \ (I: F_I, \ldots, n: F_n)]$ is the function which yields for any argument the result of applying any applicable one from $D[F_I], \ldots, D[F_n]$.

If there are any arguments to which different functions from $D[F_n], \ldots, D[F_n]$ are applicable and for which they yield different values then 'function-choice (I: F_I, \ldots, n: F_n)' has no denotation.

13. $D[A = B]$ is $\text{true}$ if $D[A]$ and $D[B]$ are identical entities; otherwise it is $\text{false}$.

14. $D[\text{amount}_u \ (\text{num}: N, \ \text{unit}: E)]$ is $<$AMOUNT, $<$A, B$>$>, where $A = D[N]$ and $B = D[E]$.

15. If $D[A] = <$AMOUNT, $<$M, U$>$> then $D[\text{amount}_u \ (\text{num}: N, \ \text{amount}: A)]$ is $<$AMOUNT, $<$B, U$>$>, where $B = M * D[N]$.

16. If $D[B]$ is a set of pairs such that

$\forall a, b, c, d: \langle a, b \rangle \in D[B] \ \& \ \& \ c, d \in D[B] \Rightarrow (a = c \ \& \ b = d)$

then $D[\text{function} \ (\text{pairs}: B)]$ is the function whose extension is defined by $D[B]$; otherwise, 'function (pairs: B)' has no denotation.

17. If $D[A]$ is a bag containing no duplicates, then $D[\text{bag-to-set} \ (\text{arg}: A)]$ is the set containing precisely the same elements as $D[A]$; otherwise 'bag-to-set (arg: A)' has no denotation.

18. $D[\text{tuple}_n \ (I: E_I, \ldots, m: E_m)]$ is the $n$-tuple having $D[E_I]$ as its first element, $\ldots, D[E_m]$ as its $n^{\text{th}}$ element.

19. $D[\text{bag} \ (\text{tuple}: NT)]$ is the bag containing all elements of $D[NT]$.

$D[\text{set} \ (\text{tuple}: NT)]$ is the set containing all elements of $D[NT]$.

$D[\text{list} \ (\text{tuple}: NT)]$ is the list containing all elements of $D[NT]$, in the same order.
$D[\text{file}]$ (tuple: $NT$) is the file containing all elements of $D[NT]$, in the order of their first occurrence.

20. $D[\text{function}]$ (tuple: $TT$) is the function whose extension is defined by $D[TT]$. If $D[TT]$ does not define a function-extension 'function (tuple: $TT$)' has no denotation.

21. $D[\text{cond}]$ (if: $P$, then: $E$) is $D[E]$ if $D[P]$ is TRUE; otherwise it does not have a denotation.

22. If $i$ is an integer, then $D[\text{id}_i]$ (arg: $E$) is $\langle \langle \text{ID}, \triangleright, D[E] \rangle \rangle$.

23. If $E$ denotes, for some $i$ and some $A$, $\langle \langle \text{ID}, \triangleright, A \rangle \rangle$ then $D[\text{rid}]$ (arg: $E$) is $A$.

24. $D[(\text{presup}: P, \text{descr}: E)]$ is $D[E]$. (This shows that the presup-descr branching is semantically superfluous. The role of this branching in the language is described in section Bronnenberg et al. (1980), section 5.5.)

It must be noted that the semantic definition of every branching category is in accordance with its type definition:
If $D[A_1] \in \text{DOM (type}[A_1])$ & ... & $D[A_n] \in \text{DOM (type}[A_n])$ then $D[b(\text{sel}_1: A_1, \ldots, \text{sel}_n: A_n)] \in \text{DOM (type}[b(\text{sel}_1: A_1, \ldots, \text{sel}_n: A_n)])$.

This is a requirement which any branching category must fulfill.

5. Semantic Anomaly.

The phenomenon of 'semantic anomaly' is perhaps best known from the semantics of natural language. In the linguistic literature, one finds the observation that certain sentences, though syntactically well-formed, are nevertheless 'weird', 'absurd' of 'deviant': sentences of the kind "The typewriter drinks the square root of the President of France".

In a formal language with a many-sorted type system, similar phenomena may be observed, in an even more clearcut way: an expression may be semantically anomalous in being tautologous, in being self-contradictory, in denoting the empty set under all interpretations the language (though using descriptive constants in the expression), in having no denotation under any interpretation of the language. Detecting that an expression has properties like those just mentioned, may be an important facility in a question-answering system: it may be worthwhile to be able to detect semantically anomalous representations of questions, since they may represent less plausible readings of the input question.

Procedures which detect the semantic anomaly of expressions must be able to use the appropriate information about the semantic types of the sub-expressions of an expression, since these indicate the range of their possible denotations. The type of any expression is defined recursively in terms of the types of its sub-expressions. (See section 3). To 'match' the types of a
function and its argument, or to compare the domain of a predicate with the range of the variable in a quantification, a procedure which checks the semantic well-formedness of sentences uses the "type-inclusion" relation. This is a relation between types, written as $\subset_T$, which has an important semantic property:
$\alpha \subset_T \beta$ implies $^5$ that for every interpretation the domain of $\alpha$ is a subset of the domain of $\beta$.

The type-inclusion relation is recursively defined, in terms of relation $\subset_c$ on "component types".

$\alpha \subset_T \beta \overset{\text{def}}{=} \forall \phi \in \text{COMPONENTS}(\alpha): \exists \psi \in \text{COMPONENTS}(\beta): \phi \subset_T \psi$

(For the definition of COMPONENTS, see section 2.)

$\phi \subset_T \psi \overset{\text{def}}{=} \begin{cases} \text{if for some } \Omega \in \{L, B, S, F, AMT, ID_2, ID_1, \ldots\} & \\
\phi \text{ has the form } \Omega(\phi') & \psi \text{ has the form } \Omega(\psi') \\
\text{then } \phi' \subset_T \psi' & \\
\text{else if } \exists n: \phi \text{ has the form } \langle\phi_1, \ldots, \phi_n\rangle & \psi \text{ has the form } \langle\psi_1, \ldots, \psi_n\rangle \\
\text{then } \forall i: \phi_i \subset_T \psi_i & \\
\text{else if } \psi \text{ has the form } (\psi_a \rightarrow \psi_v) & \\
\text{then } \phi \text{ has the form } (\phi_a \rightarrow \phi_v) \text{ or } (\phi_a \rightarrow \phi_v) \\
\text{and } \phi_a \subset_T \psi_a & \phi_a \subset_T \psi_a & \psi_a \subset_T \phi_a & \\
\text{else if } \psi \text{ has the form } (\psi_a \rightarrow \psi_v) & \\
\text{then } \phi \text{ has the form } (\phi_a \rightarrow \phi_v) \\
\text{and } \phi_a \subset_T \psi_a & \phi_a \subset_T \psi_a & \psi_a \subset_T \phi_v & \\
\text{else } \phi \text{ and } \psi \text{ are atomic types } & \phi \equiv \psi. \\
\end{cases}$

For instance, for different atomic types $\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1$ and $\gamma_2$:

$\alpha_1 \subset_T \alpha_1$ holds, 
$\cup (\alpha_1, \beta_1) \subset_T \alpha_1$ does not hold,
$\alpha_1 \subset_T \cup (\alpha_1, \beta_1)$ holds,
$\cup (\alpha_1, \beta_1) \cup (\beta_1, \cup (\alpha_1))$ holds,

$^5$ In Landsbergen & Scha (1977) it is claimed that $\alpha \subset_T \beta$ iff for every interpretation the domain of $\alpha$ is a subset of the domain of $\beta$. But the implication from right to left is not valid – neither for the definition of $\subset_T$ which was used in the 1977 paper, nor for the current one.
S (α₁) ⊑ S (∪ (α₁, β₁)) holds,
(α₁ → β₁) ⊑ (α₁ → ∪ (β₁, γ₁)) holds,
(α₁ → β₁) ⊑ (α₁ → β₁) holds,
∪ (α₁, α₂), ∪ (β₁, β₂) ⊑ 
∪ (∪ (α₁, α₂), ∪ (β₁, γ₁), <α₁, β₁>, <α₂, β₂>, <β₂, γ₂>, <γ₂, β₁>) holds.

Equality of types, written as "\(\sim\)", is defined as mutual inclusion:

\[ \alpha \sim \beta \overset{\text{def}}{=} \alpha \subseteq \beta \quad \& \quad \beta \subseteq \alpha. \]

For instance, for different atomic types \(\alpha\) and \(\beta\):

\[ \alpha \sim \beta \quad \text{does not hold}, \]
\[ \alpha \sim \alpha \quad \text{holds}, \]
\[ S (\alpha) \sim \cup (S (\alpha)) \quad \text{holds}, \]
\[ \cup (S (\alpha), S (\beta)) \sim S (\cup (\alpha, \beta)) \quad \text{does not hold}. \]

Using the relation \(\subseteq\), we can now give the following syntactic definitions of function-application, quantification, iteration and selection, which impose stricter demands on the types of their sub-expressions than the definitions given before.

If \(F\) has type \((\alpha \rightarrow \beta)\) or \((\alpha \rightarrow \beta)\), \(E\) has type \(\varepsilon\) and \(\varepsilon \subseteq \alpha\), then **application** \((\text{fun}: F, \text{arg}: E)\) has type \(\beta\).

If \(A\) has type \(S (\varepsilon)\) or \(B (\varepsilon)\), \(P\) has type \((\alpha \rightarrow \text{truthvalue})\) or \((\alpha \rightarrow \text{truthvalue})\), \(F\) has type \((\alpha \rightarrow \beta)\) or \((\alpha \rightarrow \beta)\), and \(\varepsilon \subseteq \alpha\), then:

**universal-quantification** \((\text{forall}: A, \text{holds}: P)\)

and

**existential-quantification** \((\text{forsome}: A, \text{holds}: P)\)

have type \(\text{truthvalue}\),

**selection** \((\text{head}: A, \text{mod}: P)\) has type \(S (\varepsilon)\),

**iteration** \((\text{for}: A, \text{apply}: F)\) has type \(B (\beta)\).

6. Additions and Abbreviations.

The previous sections of this Appendix have defined a fragment of the PHLIQA1 language which is a slight extension of the language defined in Bronnenberg et al. (1980). However, in Chapters II and V the examples lay often outside the boundaries of this fragment; they use some semantic operations which I did not include in the Bronnenberg et al. language.
because they are not particularly interesting. For the sake of readability, I
have also used some abbreviations which deviate considerably from the
PHLIQA1 notation. I shall now list these additions and abbreviations.

Additional operations.

1. If A has type B(integer) or S(integer), Sum (arg: A) has type integer. It
denotes the sum of the elements of D[A].
2. If N has type integer, Ints (arg: N) has type S(integer). It denotes the set of
integers i such that 0<i\leq D[N].
3. If M and N have type integer than greater-than (1:M, 2:N), also written as
M>N, and smaller-than (1:M, 2:N), also written as M<N, have type
truthvalue, and have their usual meaning.
4. If A and B have type S(α) and S(β), then intersection (1:A, 2:B) has type
S(∪{γ1, ..., γn}), where {γ1, ..., γn} = COMPONENTS(α) ∩ COMPONENTS(β);
it has the obvious meaning, and may also be written as A \cap B.
5. If A has a type of the form B(S(α)) or S(S(α)), then union (arg: A), also
written as ∪ (A), has type S(α), with the obvious meaning.

Additional Abbreviations.

{A1, ..., An} stands for set (⟨A1, ..., An⟩).

(λx1, ..., xn: E) with x1 of type α1, ..., xn of type αn, stands for (λu: E') where u
has type ⟨α1, ..., αn⟩ and E' = E [x1 := u[1], ..., xn := u[n]].

If A has type S(α) or B(α) and B has type truthvalue, then:
{x \in A \mid B} stands for selection (head: A, mod: (λx: B)),
[\lambda x \in A \mid B] stands for unel (arg: selection (head: A, mod: (λx: B))),
(∀x \in A: B) stands for universal-quantification (forall: A, holds: (λx: B)),
(∃x \in A: B) stands for existential-quantification (forsome: A, holds: (λx: B)),
(∃x \in A: B) stands for Count (arg: selection (head: A, mod: (λx: B))) = n,
P(A) stands for power (arg: A),
P_n(A) stands for selection (head: power (arg: A), mod: (λx: Count(x) = n)),
where x has type α.
If F has type (⟨α⟩ → β) or (⟨α⟩ → β) and C has type α, then F(⟨C⟩) may be
written as F(C).

In syntactic positions where expressions with a type of the form S(α) are
required, I have also allowed expressions with a type of the form B(α) in
some examples. If A is such an expression one should, in such cases, replace
D[A] by D[bag-to-set (A)] in the semantics.
Remark.

In the PHLIQA1 system, the PHLIQA1 languages are always used in an extensional way – a descriptive atomic type *possible-world* is nowhere assumed.

In Chapter II, where the PHLIQA1 treatment of questions and answers is being compared with some intrinsically intensional treatments of questions and answers, I have however taken the liberty to combine the notation defined above with the use of intension- and extension-operators in the style of Montague’s (1973) IL. I hope the syntax and semantics of this "hybrid language" are sufficiently self-evident, so that I am justified in abstaining from an explicit definition.