COMPOSITIONAL INTERPRETATION OF COMPUTER COMMAND LANGUAGES

GÁBOR RÁDAI[*] AND LÁSZLÓ KÁLMÁN[*†]

[*]DEPARTMENT OF SYMBOLIC LOGIC, BUDAPEST UNIVERSITY (ELTE)
[*]RESEARCH INSTITUTE FOR LINGUISTICS, HAS, ROOM 119
[†]THEORETICAL LINGUISTICS PROGRAMME, BUDAPEST UNIVERSITY (ELTE)
E-MAIL: radai@nytud.hu, kalman@nytud.hu

WORKING PAPERS IN THE THEORY OF GRAMMAR, VOL. 2, NO. 2
SUPPORTED BY THE HUNGARIAN NATIONAL RESEARCH FUND (OTKA)

THEORETICAL LINGUISTICS PROGRAMME, BUDAPEST UNIVERSITY (ELTE)
RESEARCH INSTITUTE FOR LINGUISTICS, HUNGARIAN ACADEMY OF SCIENCES

BUDAPEST I., P.O. BOX 19. H-1250 HUNGARY
TELEPHONE: (36-1) 175 8285; FAX: (36-1) 212 2050

0. Introduction

The aim of this paper is to examine the traditional concept of *compositionality*. We will be dealing with a language, namely, the language of commands used in the Unix operating system, the interpretation of which is intuitively far from compositional, although it fits the traditional definition of compositionality. We will outline the reason of this discrepancy, then we will show how to modify the language so that it receives an intuitively compositional interpretation. We show that this will get us closer to a more reasonable definition of the principle of compositionality and its significance for the semantics of natural languages.

The paper is organized as follows. In section **1** we present the Principle of Compositionality and argue that it is to be strengthened, because it is too loose in its original formulation. In particular, we introduce the Principle of Independence, and propose to include it into the Principle of Compositionality. The rest of the paper discusses a language, namely, the language of commands used in the Unix operating system, the interpretation of which is far from compositional in the intuitive sense of the word. However, the traditional Principle of Compositionality does not preclude such an interpretation. First, in section **2**, we explain the concept of *shells* (command interpreters), and show how the Unix command language is non-compositional. Then we present an alternative command language which has a more natural interpretation, based on our version of the concept of compositionality. Section **3** informally presents the way in which such a 'compositional Unix shell' should work. Then we develop a language to talk about the semantic domains relevant to our interpretation, i.e., various components of a simplified concept of *machine states* (section **4**). Then we explain the concept of *denotational semantics* (section **5**), a non-procedural view of the interpretation of computer programs, which underlies the particular structure that we attribute to our semantic domains (section **6**). The actual syntax and semantics of the language in which we can talk about those objects is given in section **7**, and the description of the semantics of *command lines* (commands followed by parameters) will be explained in section 8. The way in which we produce those meanings from those of the command names and the parameters in a compositional way is explained in section **9**. Finally, we offer some conclusions (section **10**).

1. Compositionality

Let us first define the concept which will be in the centre of our attention throughout this paper. The interpretation of a language can be said *compositional* if and only if it obeys the Principle of Compositionality, which runs as follows:

1.1. The Principle of Compositionality

The meaning of a complex expression is a function of the meanings of its constituents and their mode of combination.

This definition leaves it open whether 'the meanings of the constituents' may depend on each other or on the function that we use to calculate the meaning of the complex expression. However, it seems that the Principle of Compositionality would be rather vacuous if we were to allow for such dependencies. That is, we understand that the intended content of the Principle of Compositionality implies a *Principle of Independence*:

1.2. The Principle of Independence

The meanings of the constituents of a complex expression are assigned independently of each other and the function that yields the meaning of the complex expression.

The reason why we propose to add this principle is that, as we will see shortly, languages that obey the Principle of Compositionality may still be rather 'non-compositional' if they fail to satisfy the Principle of Independence. In such languages, the meaning of an expression may vary depending on what it is a constituent of. As a result, very similar constructions (e.g., containing the same expression in the same syntactic role) may be interpreted in heterogeneous (or even unrelated) ways. We submit that this contradicts the intuition behind the concept of compositionality.

Note that the interpretation of compositionality proposed here implies that the meaning contributions of the constituents of an expression are constant, i.e., they do not vary from one construction to the other. This means a certain *context-independence* as well, which many would deny. We conceive of this as a price to pay for a reasonable concept of compositionality. In our approach, the context of utterance (and the utterance-internal context of any sub-expression) can only play a role inasmuch as both the meanings and the functions that combine them are *underspecified*. That is, by virtue of their underspecification, contextual factors (including the internal context, i.e., the presence of the others) may enrich these meanings. This kind of mechanism does not contradict the Principle of Independence, because it is not *the meanings assigned* that depend on each other, but what they become later on.

It is easy to see that the Principle of Independence is not vacuous at all. The interaction of meanings is by definition *contentful*, i.e., the Principle of Independence prevents meaning assignments from depending on *formal* properties of the context (e.g., the shape of a co-occurring constituent). Only genuine homonyms (homophonous expressions with independent meanings) challenge this principle; those have to be considered different expressions which accidentally are of the same shape. So whether an ambiguity is due to an accidental surface coincidence or a systematic semantic phenomenon must be determined independently.

2. Unix shells

A *shell* is a program that establishes contact between the operating system of a computer and its user. Its task is to forward the user's *commands* to the operating system (after a check of correctness). (A command is also called a *command line*; we will refer to it as a *cml*.) Many shells offer additional features to the user (such as abbreviatory mechanisms and ways of referring to commands issued earlier), as well as built-in commands. The shells used with the Unix operating system (especially the C-shell) offer many such features. The commands that do not exploit the extra possibilities offered by the shell may contain a *command name* (cm_n) and various types of *parameters* that follow it. The command name is simply the name of a computer program; the program processes the parameters, so their interpretation is its 'internal affair'. (Built-in shell commands do not correspond to programs, the parameters of such commands are processed by the shell itself.) The language also has certain *operators* (*opr*), which can be prefixed to any comand line. They correspond to programs that run the remaining command line, and perform some uniform computation in the meantime.[1]

The informal syntactic and semantic description of command lines is available in the form of *manual pages* provided with the operating system. A manual page contains the summary of the syntax associated with a command name followed by the description of what the command lines do. Let us take a look at the syntactic description of the command called `grep`:

2.1. Example

`grep [-bchilnsvy] [-f` *expfile*`] [[-e]` *expression*`] [`*files*`]`

First comes the specification of the command name, followed by the list of *flags* (*fl*). In the case of `grep`, these are one-character strings that can be concatenated in any order and their concatenation must be preceded by a minus sign. In general, we can think of a flag as any string containing no blank space and preceded by a minus sign. (Flags are in principle optional; in manual pages, [·] means optionality.) Then come two *options*, each consisting of an *option letter* and its *argument*. (An option letter is like a flag, but it has an argument.) The option letter in the second option is itself optional. Finally, the last item is an *optional argument* (*opt*), i.e., a parameter that has a fixed position in the command line which is not preceded by an option letter. In fact, the above syntactic summary is the abbreviation of two different syntactic possibilities:

2.1′. Example

a. `grep [-bchilnsvy] [-f` *expfile*`] [-e` *expression*`] [`*files*`]`
b. `grep [-bchilnsvy] [-f` *expfile*`] [`*expression*`] [`*files*`]`

[1] For example, the operator `time` will return the time the process given as its argument has taken to run.

In **2.1′**a, we have nine flags, two options and an optional argument; in **2.1′**b, there are nine flags, one option and two optional arguments.[2]

In general, the syntax of the relevant fragment of the language of Unix command lines ($L^{(cml)}$) in BNF is as follows:

2.2. Definition

1. $cml \stackrel{def}{=} opr\ cml \mid cm_0 \mid cml\ fl \mid cml\ opt$;
2. $cm_n \stackrel{def}{=} c^0_n \mid \ldots \mid cm_{n+1}\ expr \mid cm_{n-1}\ opl$;
3. $opt \stackrel{def}{=} expr$;
4. $expr \stackrel{def}{=} n \mid c^0 \mid \ldots \mid var^0 \mid \ldots$.

c^n_n stands for n-argument command name constants, n stands for natural numbers, and c^n stands for a name constant denoting elements of the universe — files, directories, etc., as we will see. As one can see from the definition, we assume that flags and options come at the end of command lines rather than between the command name and its arguments. This modification does not make any difference except for the fact that the description of the semantics of the relevant constructions will be far simpler. In what follows, we will not discuss the semantics of most of the constructs specific for the shell language; we will concentrate on the semantics of commands.

The language presented above is an idealisation of the currently available languages, as the construction rules in the given form are context free, whereas in the actual command language as specified in the manual pages construction rules are separately given for every command as can be seen from the syntax of the command **grep** above. It is obvious that, for example, the syntactic rule that combines command names with flags is context sensitive in the sense that the program will report a syntax error if a flag is not explicitly listed in the program description. On the one hand, it would be desirable to have a context free language as $L^{(cml)}$ and, on the other hand, it is more in line with our intuition that if a modifier comes from a closed syntactic class, but is not applicable in a certain context, then this is a semantic, rather than a syntactic phenomenon. It should be explained in terms of semantic incompatibility or vacuous semantic operations rather than in syntactic terms. In what follows, we will assume the above language and let our semantic apparatus be such that it accounts for the problems connected with the relevant constructions.

There are also more important problems, related to the compositionality of the interpretation of commands. Besides the fact that command names come

[2] The above description is not quite correct, since exactly one of the *expfile* and *expression* arguments is in fact obligatory.

with some predefined sets of possible parameters (flags and option letters), the interpretation of these also depends on the command name at hand. For example, the flag `-l` means roughly 'long, verbose listing' in connection with the command name `ls`,[3] whereas as an argument to `wc` it means something like 'count lines only'.[4] Similarly, while the option letter `-f` (standing for 'file') introduces the name of an auxiliary file (containing expressions or commands) with `grep` and similar commands (`make`, `awk`, `sed` etc.), it is a flag that stands for 'force' with the command `rm` (remove), and has a totally different effect.[5]

A second problem is the issue of multiple flags. In general, the order of flags does not make any difference and multiple occurrences of the same flag in one command cause the same change in behaviour as single occurrences, as one would expect. Yet we have to face the problem of *dependent flags*, i.e., the problem that certain flags can only appear in the presence of some other flag. For example, the flag `-u` depends on the presence of `-t` in this sense with the command name `ls`.[6] Though even the informal semantics makes this perfectly understandable, currently this is treated as a syntactic constraint, which again clearly does not agree with one's intuition.

As a matter of course, the idiosyncratic behaviour of flags can be explained away by assuming that flags are functors over command names as arguments.

---

[3] `ls -l` lists the files specified by its argument in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file. If the file is a symbolic link, the filename is printed followed by '`->`' and the pathname of the referenced file. If the file is a special file, the size field will contain the major and minor device numbers, rather than a size. A total count of blocks in the directory, including indirect blocks, is printed at the top of long format listings.

[4] `wc` counts lines, words and characters in the named files, or in the standard input if no names appear. It also keeps a total count for all named files. A word is a maximal string of characters delimited by spaces, tabs, or newlines. The flags `-l`, `-w` and `-c` may be used in any combination to specify that a subset of lines, words, and characters are to be reported.

[5] `rm` removes each given file. By default, it does not remove directories. If the `-f` ('force') flag is used, it ignores nonexistent files and does not prompt the user if the file is unwritable.

[6] `ls -t` sorts the files listed by last modification time (latest first) rather than by name.

`-u` uses time of last access instead of time of last modification for sorting; can only be used with the `-t` flag.

Since there is only a finite number of commands, the meaning of a flag could be a partial function defined pointwise, i.e., one whose action is determined by first looking at its argument.[7] A similar issue is raised by the ways in which the presence vs. absence of options and optional arguments is significant. For example, if the command `set` is followed by two arguments (a *name* and a *value*), it causes the variable *name* to be set to *value*, whereas if it stands without an argument, the corresponding action is to display the currently set variables with their values. This can again be dealt with using several mathematical tricks such as polymorphic functions or empty strings as arguments, defining the function again pointwise.

Obviously, under the current wording of the Principle of Compositionality, a compositional interpretation of Unix commands can be given that uses only functional application,[8] although we have the very strong feeling that, under a more appropriate view of compositionality, this should not be possible. In particular, the heterogeneous interpretation of flags (and other option letters) as well as the heterogeneous behaviour of absent optional arguments are incompatible with our Principle of Independence. In what follows, we will specify a semantics that we feel comes closer to the original idea behind compositionality and that will remedy some of the problems mentioned above. We will see that this type of interpretation will satisfy the Principle of Independence.

3. Compositional Unix: An Informal Outline

Anomalies like the homonymy of the `-f` flag mentioned earlier should not occur in a Unix shell with compositional semantics (and they occur to a very limited extent in natural languages). In a compositional Unix shell, there must be a flag `--force` to be used with `rm` (and similar commands)[9], and a different flag `--file` to be used with `grep` (and similar commands). (Needless to say, what name we choose for these flags is immaterial.) The meanings of `--force` and `--file` must

[7] This method would give us a function that is as good as any other mathematically. Even if we assume that the number of commands is infinite and that the function is totally defined, we just have to define the result of the application of a flag to some command for which it is undefined as the action of issuing some error message — again an action that makes exactly as much sense as any other from the mathematical point of view.

[8] For example, the meaning of a flagged command is the action it performs. Compositionality in the above sense is not even destroyed by the fact that the flag as a function does not necessarily preserve anything of the original action performed by its argument.

[9] As it is conventional, we will use `--` instead of `-` to indicate that something is a multiletter flag rather than the concatenation of independent flags.

be assigned uniformly and independently of the context. For example, `--force` could be interpreted as 'overwrite the file argument if you own the file, even if you do not have write permission for it'. (Eventually, it can also cover 'do not check if the file argument exists at all', although it would be cleaner to separate these two meanings, so that the latter is to be expressed by, say, `--ignore`.) Similarly, the interpretation of the option letter `--file` would be interpreted as 'the name of an auxiliary file (containing commands etc.) follows'.

Assuming that the programs corresponding to `rm`, `grep` etc. operate as they usually do in Unix (i.e., that we are not to rewrite them), the shell will interpret these program names independently of their original interpretation (or relying on the original interpretation if needed). To achieve this, we will assume that the shell maintains a *lexicon* which contains a *program specification* for each possible command name. Program specifications contain *variables* corresponding to the possible effects of parameters. For example, the value of the variable WRITECHECK determines whether write permission is to be checked before overwriting a file; the variable EXISTCHECK determines whether the non-existence of a file will trigger a special action; and the value of AUXFILE stores the name of the auxiliary (command) file. If necessary, program specifications assign *default values* to such variables, which can be overridden by parameters.

The procedure described above corresponds to a certain *underspecification* of the actual effect of running the programs. The program specifications will ensure that the external context (the so-called *environment*, a set of variable bindings) and the (obligatory and optional) parameters together specify the exact action to take when invoking a program.

4. Machine States

To give a semantics for the language of Unix commands, we assume that the relevant basic domain is that of *machine states* (*MS*). For the sake of simplicity, we will represent a machine state with the disjoint union of a *typed directed acyclic graph* (*TDAG*), standing for the *directory structure* and the *files* stored, and a domain $\mathbf{NC}_\perp \stackrel{\text{def}}{=} \mathbf{N} \oplus \mathbf{Char}^*$ for the denotation of the natural numbers and character strings,[10] forming the universe of interpretation, an *interpretation function* and a *valuation* corresponding to the *environment*. In this section we will mainly be concerned with the graphs belonging to a machine state, the other tree components will be explained in detail in section **7**. A typed directed acyclic graph is defined as follows:

[10] The exact meaning of the above notation will be defined later, cf. definition **5.4**.

4.1. Definition

1. Given strings v and u, v is a *prefix* of $u \overset{\text{def}}{\Leftrightarrow} \exists w.u \equiv vw$.
2. A *tree domain* D is a non-empty subset of strings (*tree addresses*) in $\mathbf{N}^*$ such that:
   a. for each $u \in D$, every prefix of u is also in D;
   b. for each $u \in D$, for every $i \in \mathbf{N}^*$ if $ui \in D$ then, for every j such that $1 \leq j \leq i$, uj is also in D.
3. Two tree addresses are *independent* if neither is a prefix of the other.
4. A tree address u is *terminal* $\overset{\text{def}}{\Leftrightarrow}$ there is no tree address v in D such that u is a prefix of v.
5. Given a set T of types and $\Sigma = \bigcup_{\tau \in T} \Sigma_\tau$ of labels, a *typed tree* is a total function $t_T : D \to \Sigma$, where D is a tree domain.
6. A *typed directed acyclic graph* is an ordered pair $\langle t_T, R \rangle$, where t_T is a typed tree and R is an equivalence relation on D ($\mathrm{dom}(t_T)$) such that for all $u, v \in \mathrm{dom}(t_T)$, if $(u, v) \in R$, then:
   a. $ui \in \mathrm{dom}(t_T) \Leftrightarrow vi \in \mathrm{dom}(t_T)$;
   b. $ui \in \mathrm{dom}(t_T) \Rightarrow (ui, vi) \in R$;
   c. $t_T(u) = t_T(v)$.

Not all TDAG's are acceptable in machine states. In our case, the TDAG associated with $\mathbf{NC}_\perp$, an interpretation function and a valuation has some further special properties, as shown by the following definition.[11] We suppose that $T = \{\mathrm{dir}, \mathrm{file}, \mathrm{Char}^*\}$, i.e., the relevant types are *directory*, *file* and *character string*.

4.2. Definition

$\langle \mathrm{tdag} \oplus \mathbf{NC}_\perp, \rho, v \rangle \in \mathrm{MS} \overset{\text{def}}{\Leftrightarrow} \mathrm{tdag} = \langle t_T, R \rangle$ is a TDAG, and

1. $\rho\colon \mathrm{Con} \mapsto \mathrm{dom}(t_T) \oplus \mathbf{NC}_\perp$
2. $v\colon \mathrm{Var} \mapsto \mathrm{dom}(t_T) \oplus \mathbf{NC}_\perp$
3. $t_T(u) \in \Sigma_{\mathsf{dir}} \Rightarrow \forall i \in \mathbf{N}.t_T(ui) \in \Sigma_{\mathsf{dir}} \vee t_T(ui) \in \Sigma_{\mathsf{file}}$;
4. $t_T(u) \in \Sigma_{\mathsf{file}} \Rightarrow t_T(u1) \in \Sigma_{\mathrm{Char}^*} \wedge \neg\exists i \in \mathbf{N} \setminus \{1\}.ui \in \mathrm{dom}(t_T)$;
5. $t_T(u) \in \Sigma_{\mathrm{Char}^*} \Rightarrow \neg\exists i \in \mathbf{N}.ui \in \mathrm{dom}(t_T)$;
6. $t_T(0) \in \Sigma_{\mathsf{dir}}$;
7. $1, 11, 111 \in \mathrm{dom}(t_T)$, $t_T(1) \in \Sigma_{\mathsf{dir}}$, $t_T(11) \in \Sigma_{\mathsf{file}}$, $t_T(111) \in \Sigma_{\mathrm{Char}^*}$, and $\neg\exists i \in \mathbf{N}.1i \in \mathrm{dom}(t_T) \vee 11i \in \mathrm{dom}(t_T) \vee 1i \in \mathrm{dom}(t_T)$.

The above definitions formulate the following constraints on what ordered triples of universe, interpretation function and valuation we accept as machine

[11] The identity of the labels does not play any role in what follows. $\oplus$ in clauses 1 and 2 means roughly the disjoint union of the two domains. Although the domain consists of the disjoint union of a TDAG and $\mathbf{NC}_\perp$, we are only interested in the disjoint union taken with the domain of the TDAG, as the subsequent clauses show. For the exact definition, see definition **5.4**.

states proper. The interpretation and the valuation associated with the universe are functions that assign either a numerical value, a character string or a tree address to a constant or a variable of the language to be given in section **7**, depending on its type, as we shall see. Furthermore, in an MS labels associated with the terminal addresses of the underlying tree have to be of type 'dir' or 'Char*',[12] i.e., empty directories or finite lists of characters corresponding to contents of files.[13] We have to impose some further constraints guaranteeing that character strings are only immediately prefixed[14] by files and the latter are immediately prefixed by directories and that files only immediately prefix one character string which immediately prefixes nothing. As the sorts form domains of their own, additionally, t_T has to contain three special elements: $\perp_{\mathbf{Char}^*}$, $\perp_{\mathbf{file}}$ and $\perp_{\mathbf{dir}}$ — their tree addresses are 1, 11 and 111, respectively —, neither being the prefix of any other tree address. These will serve as the so-called bottom elements of their respective domains — as required by domain theory (cf. sections **5–6**), but they will also be put to special use in our semantics, as will be explained later on.

We will provide the compositional Unix command language with a so-called *denotational semantics*. This makes it necessary to introduce some concepts before specifying what the domains of the semantic values of the various expressions in our language will be.

5. Denotational Semantics

We will use denotational semantics — as worked out and described in Scott and Strachey (1971) — for the description of the relevant fragment of a Unix command language. To illustrate the basic points, let us take a look at the following two programs:

5.1. Example

$$F(n) \Leftarrow \text{If } n = 0 \text{ then } n \text{ else } F(n-1)$$
$$G(n) \Leftarrow 0$$

Obviously, the two programs do quite different things. The program F — on receiving an argument n of type N — will *recursively* compute a value, namely the value 0. Program G, on the other hand, will immediately produce the same result. Although we see that the two programs produce the same output on appropriate

---

[12] We will use the terms *file, directory* and *character string* to refer to tree addresses labelled with objects of the appropriate type.

[13] As costumary, we think of empty files as containing the empty string of characters, i.e., the string of length 0.

[14] Let $u, v \in \mathbf{N}^*$. v is an *immediate prefix* of $u \overset{\text{def}}{\Leftrightarrow} \exists i \in \mathbf{N}.u \equiv vi$.

input, i.e., they are equivalent under the standard set theoretic interpretation of functions, computationally they are as different as any two programs can be.[15] The idea behind denotational semantics is exactly this: for many purposes it is better if we can abstract away from accidental properties of programming languages and the realizations of specific programs, so that we can regard programs essentially as realizations of some (set theoretic) functions on domains appropriate for whatever can serve as the input and the output in the language under investigation.

But things are more complicated than they seem at first sight. If we interpret the functions to be of type $f: \mathbf{N} \mapsto \mathbf{N}$, we have no problems. But what happens if we let their type be $f: \mathbf{Z} \mapsto \mathbf{Z}$? The program G will still produce 0 on every input. But F is in trouble as when it is given some $n < 0$ as an argument, it will go straight into an infinite loop. Why is that a problem for our semantics? Because we have to do something about the infinite loop, and the semantics that we chose forces us to give a denotation to this result — a denotation that can appear as values of functions. Additionally, it has to be of type $\mathbf{Z}$ to meet the constraints. For this purpose we introduce a special constant in every domain, called *bottom* ($\perp$).

Furthermore, we will need an ordering which roughly mirrors the relations of information content of the elements of the domain. This gives us an algebraic structure called a *Scott domain*. The official definition of Scott domains is as follows:[16]

5.2. Definition

$$\mathrm{sd} \stackrel{\mathrm{def}}{=} \langle U, \perp_{\mathrm{sd}}, \sqsubseteq \rangle \in \mathrm{SD} \stackrel{\mathrm{def}}{\Leftrightarrow} U \neq \emptyset, \perp_{\mathrm{sd}} \in U, \sqsubseteq \text{ a } \textit{cpo}, \text{ and } \forall x \in U. \perp_{\mathrm{sd}} \sqsubseteq x.$$

Examples are the domains $\mathbf{N}_\perp$ and $\mathbf{T}_\perp$, i.e., the domains of natural numbers and truth values with their respective bottom elements. These domains are also examples of another important notion, the so-called *flat domains*, defined as follows:

---

[15] In what follows, we will use the terms *extensional equivalence* vs. *intensional equivalence*: F and G are extensionally, but not intensionally, equivalent.

[16] U is the universe of the domain containing at least $\perp_{\mathrm{sd}}$, the information content of which is minimal according to the *complete partial ordering* $\sqsubseteq$. A *cpo* is a *po* which has *limits* $\bigsqcup_n x_n$ for all (countable) increasing sequences $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$. Certain further conditions on domains are imposed in Gunter and Scott (1990), but these need not concern us here, as they are meant primarily to ensure that the class of domains are closed under various constructions.

5.3. Definition

$$\mathrm{sd} \in FD \overset{\text{def}}{\Leftrightarrow} \forall x, y \in U.x \neq \bot_{\mathrm{sd}} \wedge y \neq \bot_{\mathrm{sd}} \Rightarrow x \not\sqsubseteq y.$$

It is obvious that if we take the ordering to be about the information content of the elements of the respective domains, then neither $\bot \sqsubseteq \top$, nor $\top \sqsubseteq \bot$, i.e., neither truth value carries more information than the other, whereas lack of information about a truth value certainly carries less information then they do and, similarly, no natural number is less informative than any other, except for the bottom element representing the 'result' of non-terminating computations.

If we take some previously given domains as basic, all other domains can be defined using certain operations on domains. These other domains include function domains, product domains and sum domains. Some of the relevant operations are defined below:[17]

5.4. Definition

- $d_1 \to d_2$ the domain of all functions from d_1 into d_2, where

$$f \sqsubseteq_{d_1 \to d_2} g \overset{\text{def}}{\Leftrightarrow} \forall x \in d_1.f(x) \sqsubseteq_{d_2} g(x).$$

  Thus $\bot_{d_1 \to d_2}$ is the function that maps every element of d_1 into $\bot_{d_2}$;
- $d_1 \times d_2$ the Cartesian product domain where

$$(x_1, x_2) \sqsubseteq_{d_1 \times d_2} (y_1, y_2) \overset{\text{def}}{\Leftrightarrow} \forall i \in \{1, 2\}.x_i \sqsubseteq_{d_i} y_i;$$

- $d_1 \oplus d_2$ the 'coalesced' sum, where elements originating from different d_i's are incomparable and both $\bot_{d_i}$ are identified with $\bot_{d_1 \oplus d_2}$;
- $d_\bot$ the lifted domain obtained by adding a new bottom element under d;
- d^* the lists of *finite* length — including strings of length 0 — with non-$\bot$ components in d.

There are two more notions that are important in the theory of domains as

[17] d_1, d_2 denote arbitrary domains. The standard function space is the space of continuous functions. Continuous functions are defined as follows: A function f is *continuous* iff

$$f(\sqcup x_n) = \sqcup f(x_n).$$

This notion is important from a technical point of view, as there are non-trivial domains (the so called *reflexive* domains) which satisfy the following equation: $d \cong d \to d$ and can serve as the denotation of some special constructs, but this will not concern us further in the paper.

well as in what will follow:

5.5. Definition

1. A function f is *monotone* $\overset{\text{def}}{\Leftrightarrow} x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.
2. A function f is *strict* $\overset{\text{def}}{\Leftrightarrow} f(\bot) = \bot$.

These properties are defined for functions on domains but there is a very intuitive analogy with computer programs. The first property is one we generally expect computer programs to satisfy, namely that they respect the richness of the input,[18] i.e., an input that is richer — according to some obvious ordering — is never taken into an output that is poorer than the output for some poorer input. The second property is less obvious, but for programs it means that we cannot design a program that saves us if it is given some erronous input, e.g., if its input is provided by the output of some program that does not terminate — as would be the case if we gave the output of program F in **5.1** on input -7 as the input to itself[19]. If we give the above output as an input to the program G in **5.1**, then its behaviour depends on whether we suppose it to operate *call-by-value* or *call-by-name*. In the former case, we get the same result as above; in the latter, we get a program that is monotone but not strict, since it assigns the same value to every input — thus satisfying the condition of monotonicity —, but it does not respect the bottom element. Similarly, it is easy to define a numerical program that is strict but not monotone — take one that takes every natural number except $\bot$ into some $n \in \mathbf{N}$ but it takes some $k \in \mathbf{N}$ into $n - 1$ (and $\bot$ into $\bot$). Thus we see that the two properties are independent.

One more remark has to be made at this point. We said before that denotational semantics is used so that we can abstract away from certain accidental properties of programs, i.e., we can see extensionally equivalent programs as having the same denotation. This will pose the problem that certain programs of the Unix command language are extensionally equivalent, but they have different side effects that we may be interested in capturing. For example, a program that simply displays the content of a file does not affect the machine state in any obvious way. So we can either take the decision to drop denotational semantics as our tool or we can simply not take account of these features of programs. But we can also try to mirror certain intensional differences — i.e., differences due to the implementation of programs that do not show under the set theoretical representation but which we consider relevant — as extensional ones, thus sticking to denotational semantics. In what follows, we take the latter path.

---

[18] In our case, inputs and outputs will be *machine states*.

[19] In our case this means that we can never recover from the *error state*.

6. The Semantic Domains

To make MS into a Scott domain, we need a bottom element $\perp_{\mathrm{MS}}$ and a *cpo*. The former is the unstructured *error state* ($\perp_{\mathrm{MS}}$); the latter is defined as follows:

6.1. Definition

1. $v_1 \leq v_2 \overset{\text{def}}{\Leftrightarrow} \forall x.(v_1(x) = \perp \wedge v_2(x) \neq \perp) \vee (v_1(x) = v_2(x))$;
2. $\mathrm{ms}_1 \sqsubseteq_{\mathrm{MS}} \mathrm{ms}_2 \overset{\text{def}}{\Leftrightarrow}$
   $\mathrm{ms}_1 = \perp_{\mathrm{MS}} \vee$
   $\mathrm{ms}_i = \langle tdag_i \oplus \mathbf{NC}_\perp, \rho_i, v_i \rangle$ (for $i \in 2$) $\wedge\ tdag_1 = tdag_2 \wedge \rho_1 = \rho_2 \wedge v_1 \leq v_2$.

That is, the error state is less 'informative' than any other state, and whereas all other states with different underlying trees or interpretation functions are incomparable, in comparable states the ordering is simply inherited from the ordering on the valuation, which says that a valuation is more informative than another if and only if it is 'defined' in some sense for more values.[20]

Now we are ready to define the semantic domains for the language of our Unix shell:[21]

6.2. Definition

1. $[\![n]\!] \in \mathbf{N}_\perp$;
2. $[\![c]\!] \in \mathrm{dom}(t_T)$;
3. $[\![var]\!] \in \mathrm{dom}(t_T) \oplus N_\perp$;
4. $[\![cm_n]\!] \in U_n \to \ldots \to U_1 \to \mathrm{MS} \to \mathrm{MS}$;
5. $[\![opt]\!] \in (\mathrm{MS} \to \mathrm{MS}) \to \mathrm{MS} \to \mathrm{MS}$;
6. $[\![fl]\!] \in (\mathrm{MS} \to \mathrm{MS}) \to \mathrm{MS} \to \mathrm{MS}$;
7. $[\![opl]\!] \in (U_n \to \ldots \to U_1 \to \mathrm{MS} \to \mathrm{MS}) \to U_{n+1} \to \ldots \to U_1 \to \mathrm{MS} \to \mathrm{MS}$;
8. $[\![cml]\!] \in \mathrm{MS} \to \mathrm{MS}$;
9. $[\![opr]\!] \in (\mathrm{MS} \to \mathrm{MS}) \to \mathrm{MS} \to \mathrm{MS}$.

There is little to say about the domain of integers; constants will evaluate to distinguished nodes of the tree, variables to nodes or natural numbers in accordance with their types. Command lines (commands) will be interpreted as functions from machine states to machine states, whereas n-argument command names yield commands when supplied with the appropriate number of arguments. Options and

---

[20] This is justified by the fact that the relevant information is basically stored in the valuation function, whereas the underlying tree and the interpretation function carry little information.

[21] Cf. definition **2.2.** Furthermore, we use the convention that bracketing is right associative. For example, $X \to Y \to Z = (X \to (Y \to Z))$.

flags, like operators, are functions from commands to commands; nevertheless, we shall see that there is a difference between operators and options/flags. Option letters create new argument places. By the definition of the domains resulting from coalesced sum, Cartesian product and function formation[22], and the flatness of $NC_\perp$, $T_\perp$ and $dom(t_T)$, the ordering relations and the bottom elements are given. For example, the least 'informative' program ($\perp_{MS \to MS}$) is the one that takes every machine state into the error state.

The interpretation of the expressions of the language $L^{(cml)}$ will proceed via a translation function into the language of specifications — the topic of the following section. That is, command lines will be translated into the specification language first, then that language will be interpreted using the semantic domains defined here.

7. A Language for Program Specifications

As we have said above, complex expressions will receive a denotation in two steps. First we define a translation function $\tau: L^{(cml)} \mapsto L^{(spec)}$, i.e., we translate expressions of the shell language into expressions of the language of *specifications*. These expressions will be given a denotation via an interpretation function and a valuation. As we shall see, these will be the desired denotations of the shell expressions. We will proceed in two steps. We first specify an auxiliary language $L^{(ps)}$ and a function $\tau_1: L^{(cml)} \mapsto L^{(ps)}$ which will serve as the basis for specifying the language $L^{(spec)}$ and the function τ.

Commands (*cml*) will be translated into program specifications (PS), which can be interpreted directly in the semantics. The translations of all other expressions (such as flags and option letters) will be given relative to PS. First of all we need a typed dynamic first-order language with equality (TDFOLE)[23] that will be sufficient to specify — i.e., to describe — functions from machine states to machine states. The set of types is defined as follows:

7.1. Definition

1. t, *dir*, *file*, *natnum*, *char**, $\in T$;
2. $\alpha, \beta \in T \Rightarrow \langle \alpha\ \beta \rangle \in T$.

The types *dir* and *file* are self-explanatory, t is the type *truth value* — i.e., the type of formulae —, *natnum* is the type of *natural numbers* and *char** stands for character strings. $\langle \alpha\ \beta \rangle$ is the type of functions from objects of type β to objects

[22] Cf. definition **5.4.**

[23] The language and its semantics will be very similar to the one given in Groenendijk and Stokhof (1991) with some modifications required by the typing.

of type α. The typed first order language based on the above set T is defined as follows:

7.2. Definition

1. $L^{(\mathrm{ps})} \stackrel{\mathrm{def}}{=} \langle \mathrm{LC_{ps}}, \mathrm{Con}, \mathrm{Var}, \mathrm{Expr} \rangle$;
2. $\mathrm{LC_{ps}} \stackrel{\mathrm{def}}{=} \{(,), ., =, \neg, \wedge, \exists\}$;
3. $\mathrm{Con} \stackrel{\mathrm{def}}{=} \bigcup_{\tau \in \mathrm{T}} \mathrm{Con}_\tau$;
   a. $\mathrm{Con}_t \stackrel{\mathrm{def}}{=} \{\top\}$;
   b. $\mathrm{Con}_{dir} \stackrel{\mathrm{def}}{=} \{\mathrm{root}, \bot_{\mathrm{dir}}\}$;
   c. $\mathrm{Con}_{file} \stackrel{\mathrm{def}}{=} \{\mathrm{tty}, \mathrm{mail}, \bot_{\mathrm{file}}\}$;
   d. $\mathrm{Con}_{natnum} \stackrel{\mathrm{def}}{=} \mathbf{N}$;
   e. $\mathrm{Con}_{char^*} \stackrel{\mathrm{def}}{=} \mathbf{Char}$;
   f. $\mathrm{Con}_{\langle natnum\ file \rangle} \stackrel{\mathrm{def}}{=} \{\mathrm{write\_permission}\}$;
   g. $\mathrm{Con}_{\langle char^*\ file \rangle} \stackrel{\mathrm{def}}{=} \{\mathrm{content}\}$;
   h. $\mathrm{Con}_{\langle \langle char^*\ char^* \rangle\ char^* \rangle} \stackrel{\mathrm{def}}{=} \{\frown\}$;
4. $\mathrm{Var} \stackrel{\mathrm{def}}{=} \bigcup_{\tau \in \mathrm{T}} \mathrm{Var}^c_\tau \cup \bigcup_{\tau \in \mathrm{T}} \mathrm{Var}^s_\tau$;
   a. $\mathrm{Var}^s_{dir} \stackrel{\mathrm{def}}{=} \{\mathrm{HOME}, \mathrm{CWD}, dir_1, \ldots\}$;
   b. $\mathrm{Var}^s_{file} \stackrel{\mathrm{def}}{=} \{\mathrm{KBD}, \mathrm{SCREEN}, file_1, \ldots\}$;
   c. $\mathrm{Var}^s_{natnum} \stackrel{\mathrm{def}}{=} \{\mathrm{WRITECHECK}, \mathrm{EXISTCHECK}, \ldots\}$;
   d. $\mathrm{Var}^c_\alpha = \{x.c\colon x \in \mathrm{Var}^s_\beta \wedge c \in \mathrm{Con}^s_{\langle \alpha\beta \rangle}\}$;
5. $\mathrm{Expr} \stackrel{\mathrm{def}}{=} \bigcup_{\tau \in \mathrm{T}} \mathrm{Expr}_\tau$;
6. $\mathrm{Con}_\tau \cup \mathrm{Var}^\alpha_\tau \subseteq \mathrm{Expr}_\tau$;
7. $\Phi \in \mathrm{Expr}_{\langle \alpha\ \beta \rangle}, \eta \in \mathrm{Expr}_\beta \Rightarrow \Phi(\eta) \in \mathrm{Expr}_\alpha$;
8. $\eta, \zeta \in \mathrm{Expr}_\alpha \Rightarrow \eta = \zeta \in \mathrm{Expr}_t$;
9. $\Phi, \Psi \in \mathrm{Expr}_t \Rightarrow \neg(\Phi), (\Phi \wedge \Psi) \in \mathrm{Expr}_t$;
10. $\Phi \in \mathrm{Expr}_t, \xi \in \mathrm{Var}^\beta_\alpha \Rightarrow \exists\xi.\Phi \in \mathrm{Expr}_t$.

The constants and simple variables of the language serve to name the elements of the machine states — i.e., files, directories, natural numbers and character strings — in accordance with our requirements. Our examples of special variables are 'HOME' for the user's home directory; 'CWD' for the current working directory; 'KBD' for the current keyboard input file ; 'SCREEN' for the current screen output file. 'root', 'mail' and 'tty' are special files and directories. The use of the remaining constants and simple variables should be obvious from their semantics that we specify later on. The functional constants are again self-explanatory, except for $\frown$ which is the symbol of *concatenation*. The denotation of $x \frown y$ is the concatenation of (the strings) x and y. We usually omit it, and indicate concatenation by mere juxtaposition. Var^c is the set of *complex variables*. The

value of a complex variable depends on its components. The operator '.' is similar to those operators of programming languages which select a particular *member* of a *structure*. We can think of unary name functions as selectors of members of such structures. We stipulate that

$$x.c \in \mathrm{Var}^c \Rightarrow x.c \equiv c(x).$$

That is, the values of name functions applied to variables can be automatically referred to by complex variables. For example, the content of the file *file* can be referred to either as 'content(*file*)' or '*file*.content'. The operator '.' associates to the left (i.e., $x.c.d \equiv (x.c).d$). Apart from '.' and '$\frown$', the language itself is given by the standard construction rules for expressions of type τ in a TDFOLE. In what follows we will be especially interested in expressions of type t[24].

We need certain further operators defined in terms of the above:

7.3. Definition

1. $(\Phi \vee \Psi) \stackrel{\text{def}}{=} \neg(\neg\Phi \wedge \neg\Psi)$;
2. $(\Phi \rightarrow \Psi) \stackrel{\text{def}}{=} \neg(\Phi \wedge \neg\Psi)$;
3. $!(\Phi) \stackrel{\text{def}}{=} \neg(\neg(\Phi))$.

The definition of $\vee$ and $\rightarrow$ is standard, whereas '!' is a unary logical sentential operator, i.e., it takes formulae into formulae.[25]

The semantic value of the well-formed expressions of the language in a machine state *ms* is produced via the function $[\cdot]^{\mathrm{ms}}$. First we define a function D that assigns semantic domains to types, i.e., it specifies which kinds of objects serve as the denotation of expressions given the set of machine states[26]:

7.4. Definition

1. $\mathrm{D}(t) \stackrel{\text{def}}{=} \mathcal{P}(\mathrm{MS})$;
2. $\mathrm{D}(\mathit{file}) \stackrel{\text{def}}{=} \{u{:}\, t_{\mathrm{T}}(u) \in \Sigma_{\mathsf{file}}\}$;
3. $\mathrm{D}(\mathit{dir}) \stackrel{\text{def}}{=} \{u{:}\, t_{\mathrm{T}}(u) \in \Sigma_{\mathsf{dir}}\}$;
4. $\mathrm{D}(\mathit{natnum}) \stackrel{\text{def}}{=} \mathbf{N}_{\perp}$;
5. $\mathrm{D}(\mathit{char}^*) \stackrel{\text{def}}{=} \mathbf{Char}^*$;
6. $\mathrm{D}(\langle\alpha\ \beta\rangle) \stackrel{\text{def}}{=} \mathrm{D}(\beta) \rightarrow \mathrm{D}(\alpha)$.

That is, the denotation of a formula is a set of machine states, whereas names of files, directories, natural numbers and character strings evaluate to elements of the

[24] In what follows, we will refer to expressions of type t as *formulae*.

[25] This is Groenendijk and Stokhof's closure operator $\diamond$.

[26] Cf. definitions **4.1** and **4.2**.

appropriate type of the universe — e.g., a file name evaluates to a node of type *file* of the underlying tree of the *tdag* — whereas functional expressions evaluate to functions of the appropriate type.

Now we are ready to define the semantics of the well formed expressions of the language $\mathrm{L}^{(\mathrm{ps})}$. First we give the definition of expressions other than formulae:[27]

7.5. Definition

1. $c \in \mathrm{Con} \Rightarrow [\![c]\!] \stackrel{\mathrm{def}}{=} \rho(c)$;
2. $x \in \mathrm{Var}^s \Rightarrow [\![x]\!] \stackrel{\mathrm{def}}{=} v(x)$;
3. $x.c \in \mathrm{Var}^c \Rightarrow [\![x.c]\!] \stackrel{\mathrm{def}}{=} [\![c]\!]([\![x]\!])$;
4. $[\![\mathrm{EXISTCHECK}]\!] \stackrel{\mathrm{def}}{=} n \in 2$;
5. $[\![\mathrm{root}]\!] \stackrel{\mathrm{def}}{=} 0 \in \mathrm{dom}(t_T)$;
6. $[\![\bot_{\mathrm{dir}}]\!] \stackrel{\mathrm{def}}{=} 1 \in \mathrm{dom}(t_T)$;
7. $[\![\bot_{\mathrm{file}}]\!] \stackrel{\mathrm{def}}{=} 11 \in \mathrm{dom}(t_T)$;
8. $[\![\bot_{\mathrm{Char}^*}]\!] \stackrel{\mathrm{def}}{=} 111 \in \mathrm{dom}(t_T)$;
9. $[\![\mathrm{write\_permission}]\!] \in F \to 2$, where $F \subset \mathrm{dom}(t_T)$ such that $t_T[F] = \Sigma_{\mathrm{file}}$.

Thus the semantic values of constants and simple variables are produced by the interpretation and valuation functions, respectively. The values of complex variables are determined as was seen before. The remaining clauses can be regarded as constraints on v and ρ. 'EXISTCHECK' is a variable that can only be set to 0 or 1 (the same holds for 'WRITECHECK'); 'root' has to denote the root of the TDAG. The name constants will represent 'immutable' objects in the machine. Some of them (especially 'mail' and 'tty') will help us avoid complications in connection with programs that do not change a machine state under the standard interpretation (since normally we are only interested in their side effects): we conceive of them as files that can grow indefinitely as strings are concatenated to their content (when mail is sent or character strings are displayed, respectively). $\bot_\tau$

[27] We assume that

$$\forall x_\alpha.[\![x]\!]^{\bot_{\mathrm{ms}}} = \bot_\alpha, \text{ where } \alpha \in \mathrm{T} \setminus \{t\}$$

i.e., the denotation of all well-formed expressions except for formulae in the error state is the bottom element of the appropriate type, as this will not influence what follows in any way. The definition below applies to all other cases. We will drop the superscript 'ms' and the type subscripts when this gives rise to no misunderstanding. $v[X]$ stands for the range of the function v when constrained to the set X.

denotes the bottom element of type τ; these are 'degenerate' objects such as non-existent files; their use will be explained later on. 'write_permission' is a function from tree addresses to 0 or 1, thus relating a tree address of type *file* to its write permission.[28]

The semantic value of formulae in a machine state will be the set of machine states that can result after the formula has been processed. Thus we specify the meanings as sets of ordered pairs of machine states. The definition runs as follows:

7.6. Definition

1. $\forall \Phi \in \mathrm{Expr}_t.\langle \perp_{\mathrm{MS}}, \perp_{\mathrm{MS}} \rangle \in [\![\Phi]\!]$;
2. $\langle \mathrm{ms}_1 \mathrm{ms}_2 \rangle \in [\![\top]\!] \overset{\mathrm{def}}{\Leftrightarrow} \mathrm{ms}_1 = \mathrm{ms}_2$;
3. $\langle \mathrm{ms}_1, \mathrm{ms}_2 \rangle \in [\![t_1 = t_2]\!] \overset{\mathrm{def}}{\Leftrightarrow} \mathrm{ms}_1 = \mathrm{ms}_2 \wedge [\![t_1]\!]^{\mathrm{ms}_1} = [\![t_2]\!]^{\mathrm{ms}_1}$;
4. $\langle \mathrm{ms}_1, \mathrm{ms}_2 \rangle \in [\![\neg\Phi]\!] \overset{\mathrm{def}}{\Leftrightarrow} \mathrm{ms}_1 = \mathrm{ms}_2 \wedge \neg\exists \mathrm{ms}_3.\langle \mathrm{ms}_1, \mathrm{ms}_3 \rangle \in [\![\Phi]\!]$;
5. $\langle \mathrm{ms}_1, \mathrm{ms}_2 \rangle \in [\![\Phi \wedge \Psi]\!] \overset{\mathrm{def}}{\Leftrightarrow} \exists \mathrm{ms}_3.\langle \mathrm{ms}_1, \mathrm{ms}_3 \rangle \in [\![\Phi]\!] \wedge \langle \mathrm{ms}_3, \mathrm{ms}_1 \rangle \in [\![\Psi]\!]$;
6. $\langle \mathrm{ms}_1, \mathrm{ms}_2 \rangle \in [\![\exists x.\Phi]\!] \overset{\mathrm{def}}{\Leftrightarrow} \mathrm{tdag}_1 = \mathrm{tdag}_2 \wedge \rho_1 = \rho_2 \wedge$
   $\wedge \exists \mathrm{ms}_3.(\mathrm{tdag}_3 = \mathrm{tdag}_1 \wedge \rho_3 = \rho_1 \wedge v_3[x]v_1 \wedge \langle \mathrm{ms}_3, \mathrm{ms}_2 \rangle \in [\![\Phi]\!])$.

Clause 1 states that the error state verifies every formula and no formula can recover from it. The formula $\top$ denotes the diagonal relation on the set MS, i.e., it is always true without any dynamic effects. The remaining clauses are the standard ones for DPL, though clause 6 looks a bit more complicated, but this is the only clause introducing dynamic effects, and it simply says that we are only interested in changes of the valuation function[29] if this leads to a valuation that can serve as an input to the embedded formula. This justifies what we said above, namely that the denotation of a formula in a machine state is a set of valuations.

Now it is easy to compute the semantic clauses for the defined operators:

7.7 Facts

1. $\langle \mathrm{ms}_1, \mathrm{ms}_2 \rangle \in [\![\Phi \vee \Psi]\!] \Leftrightarrow$
   $\mathrm{ms}_1 = \mathrm{ms}_2 \wedge \exists \mathrm{ms}_3.\langle \mathrm{ms}_1, \mathrm{ms}_3 \rangle \in [\![\Phi]\!] \vee \langle \mathrm{ms}_1, \mathrm{ms}_3 \rangle \in [\![\Psi]\!]$;
2. $\langle \mathrm{ms}_1, \mathrm{ms}_2 \rangle \in [\![\Phi \rightarrow \Psi]\!] \Leftrightarrow \mathrm{ms}_1 = \mathrm{ms}_2 \wedge \forall \mathrm{ms}_3.\langle \mathrm{ms}_1, \mathrm{ms}_3 \rangle \in [\![\Phi]\!] \Rightarrow$
   $\exists \mathrm{ms}_4.\langle \mathrm{ms}_3, \mathrm{ms}_4 \rangle \in [\![\Psi]\!]$;
3. $\langle \mathrm{ms}_1, \mathrm{ms}_2 \rangle \in [\![!\Phi]\!] \overset{\mathrm{def}}{\Leftrightarrow} \mathrm{ms}_1 = \mathrm{ms}_2 \wedge \exists \mathrm{ms}_3.\langle \mathrm{ms}_1, \mathrm{ms}_3 \rangle \in [\![\Phi]\!]$;

---

[28] We are making unforgivable simplifications here. Among others, we simply ignore the difference between character files and special files (such as *character devices*); also, we ignore other types of permissions altogether (normally the permissions of a file are encoded in four octal digits in the file system).

[29] $v_1[x]v_2$ means that the two valuations are the same except perhaps for the value they assign to x.

As for the first two definitions, there is little to say. In the case of clause 3. it should be now obvious why Groenendijk and Stokhof call it the closure operator: it closes off any dynamic effects a formula may have had. Now we have a DFOLE that has enough expressive power to describe relations between machine states. We will use this language to specify the semantics of programs. But we have to face two further problems. The denotation of a formula is a partial relation, i.e., it is neither functional nor complete. But we think of programs as *total functions* from machine states to machine states — i.e., programs are defined everywhere, and they are deterministic. This means that not every formula of the above language is appropriate as a translation of a program. To single out the class that we need, we will introduce a representation for the formulae and impose the relevant constraints on this representation, which is basically a shorthand for the formulae of $L^{(ps)}$.

8. Program Specifications

We will take the formulae that represent the translations of our programs apart and give them a representation in terms of their parts. The sentences of this representation will be the ones of $L^{(ps)}$, but we will not use all the power of this language. But now we will think about this language as an ordinary typed first order language with equality with its standard semantics. Two sentences of this new representation will play a key role in specifying programs. The first one, which we will call the *precondition* (PC) of the program, will contain the input conditions for the execution of a program; the other, called the *maximal change* (MC), specifies its output conditions. The intended interpretation is as follows: a formula ϕ is applicable to a machine state ms — i.e., $ms \in \mathrm{dom}([\![\phi]\!])$ — if and only if the machine state satisfies all sentences in the program's PC,[30] and if a program is not applicable to a machine state, we will take it to have no effect.[31] This is basically the same behaviour as that of standard shells, where an error message is issued in such a situation, but the machine state is not affected. The only way a program can lead to the error state is by leading out of the set of machine states, e.g., by removing one of the objects required by definition **4.1**. The maximal change brought about by the program is that sentences in the MC of the program are satisfied by the new machine state, and all other sentences not

[30] We take this to mean that all formulae in this component are satisfied by the machine state under some appropriate first-order definition, i.e., $\mathrm{ms} \models \Gamma \stackrel{\mathrm{def}}{\Leftrightarrow} \forall\gamma \in \Gamma.\mathrm{ms} \models \gamma$.

[31] We do that in order to get complete functions in accordance with the requirements of definition **6.2**. The general idea is that we explicitly list the presuppositions imposed by a program on the input machine states.

affected by MC retain their truth value.[32]

In actual fact, program specifications will be more complex. First, the PC will not be checked against the initial machine state directly, but a modified machine state, in which some variables are assigned local values for the execution of the program. So each program specification will contain a component describing a modification of the valuation of the initial machine state. We will call this component the *local environment* (*LENV*) of the program. The role of LENV is that we do not expect the input machine state to verify it, nor do we want it to live on in the output machine state, unless as a consequence of some property of the MC in the program specification. Second, since MC is just a sentence in a FOLE, we have to keep a separate component describing the *dynamic* aspect of the change of state effected by the program, i.e., the list of those variables the semantic value of which may change from the input state to the output state (through changes in the valuation). We will call this component the *environment change* (*ENVC*) that the program can effect.

So program specifications will be quadruples of the form

$$\langle \mathrm{LENV}; \mathrm{PC}; \mathrm{MC}; \mathrm{ENVC} \rangle,$$

where $\mathrm{LENV} \in \mathrm{Var} \to (\mathrm{Var} \cup \mathrm{Con} \cup \{*\})$ (where '$*$' represents the undefined funcion value). We will use the notation $\mathrm{ms} + \mathrm{LENV}$ to refer to the modified machine state which differs from 'ms' in its valuation only, and

$$* \neq \xi = \mathrm{LENV}(x) \Rightarrow [\![x]\!]^{\mathrm{ms}+\mathrm{LENV}} = [\![\xi]\!]^{\mathrm{ms}}.$$

On the other hand, $\mathrm{ENVC} \subseteq \mathrm{Var}$. As a matter of course, if a variable is in ENVC then, even if LENV assigns it a local value, its old value is not restored after the computation.

The component called MC does not use the full force of our language $L^{(\mathrm{ps})}$. This is due to the fact that the operation of a program is to be *deterministic*. Therefore, a sentence in MC does not contain *negation*: there may be several ways of falsifying a formula. (In this way, we also exclude conditionals and disjunctions, which also lead to non-determinism, because they are defined in terms of negation.) Another problematic type of sentence in our FOLE is *equality*: there are two ways of verifying the equality of two variables, namely, the valuation of either one (or both) can be modified in order to make their values identical. Accordingly, we will stipulate that at most one variable on either side of an equality is in ENVC, and all variables of ENVC appear in some equality — otherwise we could change the machine state arbitrarily with respect to the variables in ENVC but not in MC.

[32] Except for those changes that MC entails, of course.

This way, an equality in the MC will correspond uniquely to a change of machine state (if a change is to be effected at all).[33]

We will refer to the language of MC as $L^{(\mathrm{mc})}$ and the language of PC as $\mathrm{L}^{(\mathrm{pc})}$. We now give the formal definitions for the above concepts and a function $[\cdot]\colon \mathrm{PS} \mapsto (\mathrm{MS} \to \mathrm{MS})$, which interprets the above quadruples:

8.1. Definition

1. $L^{(\mathrm{mc})} \stackrel{\mathrm{def}}{=} \langle \mathrm{LC}_{(\mathrm{mc})}, \mathrm{Con}, \mathrm{Var}, \mathrm{Expr}_{\mathrm{mc}} \rangle$,
   where $\mathrm{LC}_{(\mathrm{mc})} \stackrel{\mathrm{def}}{=} \mathrm{LC}_{\mathrm{ps}} \setminus \{\neg\}$; otherwise it follows definition **7.8**;
   As we have mentioned above, there are two additional constraints on sentences in $L^{(\mathrm{mc})}$, namely, at most one variable on either side of an equality is in ENVC and every variable of ENVC appears in some equality.
3. $\mathrm{PS} \stackrel{\mathrm{def}}{=} \langle \mathrm{LENV}; \mathrm{PC}; \mathrm{MC}; \mathrm{ENVC} \rangle$,
   where $\mathrm{PC} \subseteq \mathrm{Form}$, $\mathrm{LENV} \subseteq \mathrm{Var} \to (\mathrm{Var} \cup \mathrm{Con} \cup \{*\})$, $\mathrm{MC} \subseteq \mathrm{Form}_{\mathrm{mc}}$ and $\mathrm{ENVC} \subseteq \mathrm{Var}$;
4. Let $\mathrm{ms}_i \in \mathrm{MS}$, and $\mathrm{ps} = \langle \mathrm{lenv}; \mathrm{pc}; \mathrm{mc}; \mathrm{envc} \rangle \in \mathrm{PS}$. Then
   $\langle \mathrm{ms}_1, \mathrm{ms}_2 \rangle \in [ps] \stackrel{\mathrm{def}}{\Leftrightarrow}$
   (a) $\mathrm{ms}_1 + \mathrm{lenv} \models \mathrm{pc} \wedge \mathrm{ms}_2 \models \mathrm{mc} \wedge \mathrm{ms}_1[\mathrm{envc}]\mathrm{ms}_2$; or
   (b) $\mathrm{ms}_1 + \mathrm{lenv} \not\models \mathrm{pc} \wedge \mathrm{ms}_1 = \mathrm{ms}_2$
   The notation $\mathrm{ms}_1[\mathrm{envc}]\mathrm{ms}_2$ is a shorthand for 'the valuations of ms_1 and ms_2 differ at most in the values that they assign to the variables in envc'.

As we said before, this ordered quadruple encodes some local assumptions (LENV), the presuppositional content of the program (PC), and the effected change (MC and ENVC). How exactly this is done is shown in the next section, using some examples. As we mentioned above, we take these constructions to be abbreviations for sentences of the language $\mathrm{L}^{(\mathrm{ps})}$, and we spell out the corresponding formulae of the above language as illustrations in some cases. Officially however the abbreviation reads as follows:[34]

---

[33] If a variable $x \in$ ENVC did not occur in an equality within MC, but another sub-formula, say, $F(x)$, then we would face non-determinism again: $P(x)$ can be verified in as many ways as there are possible values of x that make $P(x)$ true.

[34] The operator FV assigns to an expression the set of free variables it contains.

8.2. Definition
$\langle \text{LENV}; \text{PC}; \text{MC}; \text{ENVC} \rangle \stackrel{\text{def}}{=} \neg\phi \vee \phi$, where
$\phi \equiv !(\exists x_1, \ldots, x_n.\text{LENV}' \wedge \text{PC}) \wedge$
$\quad \wedge \exists y_1, \ldots, y_m(!(\exists z_1, \ldots, z_k.\text{LENV}' \wedge \text{MC}))$,
with $\{x_1, \ldots, x_n\} = \text{dom}(\text{LENV})$, $\{y_1, \ldots, y_m\} = \text{ENVC}$, $\{z_1, \ldots, z_k\} = \text{dom}(\text{LENV}) \setminus \text{ENVC}$, and

$$\text{LENV}' = `x_1 = \text{LENV}(x_1) \wedge \ldots \wedge x_n = \text{LENV}(x_n)'.$$

The above expression expresses exactly what we have described in this section. First we check the precondition under the local changes — the closure operator here serves to close off dynamic effects of the first conjunct — then we reassign the variables of ENVC and perform the checking again under the modified valuation and close off unwanted dynamic effects. The purpose of using the set LENV\ENVC in the translation is to avoid unwanted reassignment to the variables in ENVC. The first — negated — disjunct serves to achieve the effect of totalising the relation. It is easy to see that either a machine state satisfies the precondition, in which case it will be in the domain of the formula due to the second disjunct, or it does not, in which case it will be due the first disjunct that the denotation contains an ordered pair consisting of this machine state. Thus we have a total functional expression, exactly as we wanted. It is also easy to see that the denotation of program specifications under $[\cdot]$ and the TDFOLE formulae under $[\![\cdot]\!]$ will be the same. But we still use our quadruples for the sake of perspicuity.

8.1. Some Examples

Now we are ready to look at a few examples. As a matter of course, we will make gross simplifications again to avoid complications. We will also drop type subscripts on variables when they are obvious.

8.3. Example
$\tau_1(\texttt{rm}\ \mathit{file}) \stackrel{\text{def}}{=} \langle \emptyset;$
$\quad (\text{EXISTCHECK} = 1 \rightarrow \mathit{file} \neq \perp) \wedge$
$\quad (\text{WRITECHECK} = 1 \rightarrow \mathit{file}.\text{write\_permission} = 1);$
$\quad \mathit{file} = \perp;$
$\quad \{\mathit{file}\}\rangle.$

This definition says the following. First, we assume no changes for the local environment. Second, a machine state satisfies the input condition of this program if and only if the value of *file* — i.e., the first argument — is an existing file (if EXISTCHECK is set to 1), and the user has write permission to it (if WRITECHECK is set to 1). The maximal change that the program effects is that the file's value

is the non-existent file in the output state ($\perp_{\text{file}}$ denotes non-existent files, and *file* is the only member of the environment change).

To show how the mechanism works, we spell out this formula in $L^{(ps)}$:

8.4. Example

$$!(\text{EXISTCHECK} = 1 \rightarrow \neg(\mathit{file} = \perp)) \wedge$$
$$(\text{WRITECHECK} = 1 \rightarrow \mathit{file}.\text{write\_permission} = 1)) \wedge$$
$$\exists \mathit{file}[!(\mathit{file} = \perp)]$$

By calculating the semantics of this formula according to the rules given in definition **7.6**, it is easy to see that it expresses exactly the conditions on pairs of machine states spelt out above. The closure operators are vacuous in this case, but they will be needed later on, when LENV will not be empty, to close off dynamic effects, as we explained above. As the mechanism should be obvious, we do not give these translations later except when we want to illustrate some point explicitly.

8.5. Example

$$\tau_1(\texttt{cat}\ \mathit{file}) \stackrel{\text{def}}{=} \langle \emptyset;$$
$$\mathit{file} \neq \perp \wedge \text{SCREEN} \neq \perp;$$
$$\text{SCREEN.content} = \text{SCREEN.content}^\frown \mathit{file}.\text{content};$$
$$\{\text{SCREEN.content}\}\rangle;$$

This example works as follows. We assume no local changes to the environment; the file referred to by the argument as well as the file that the variable SCREEN refers to must exist; the content of *file* must be concatenated at the end of the content of the stream referred to by the SCREEN variable (normally, the file associated with the user's screen, i.e., `tty`). Finally, at most the content of this stream will be different from the input state to the output state, as the last component of the program specification shows. Now, in actual fact, the command `cat` only affects the state of the machine if the content of its output file is stored on disk. The user's screen is usually not such a file. Nevertheless, for the sake of uniformity, we consider it as if it contained the concatenation of everything that has appeared on the screen before.

8.6. Example

$$\tau_1(\texttt{cc}\ \mathit{file}) \stackrel{\text{def}}{=} \langle \{\langle \text{OUTPUTFILE}, \texttt{a.out}\rangle\};$$
$$\mathit{file} \neq \perp;$$
$$\text{OUTPUTFILE.content} = cc(\mathit{file}.\text{content});$$
$$\{\text{OUTPUTFILE.content}\}\rangle.$$

This is our first example containing a non-empty LENV component, which locally assigns the value `a.out` to the variable OUTPUTFILE. As we know, this is the

default name of the output of the program `cc` (the C compiler). (The symbol '`a.out`' is actually meant to be a variable that evaluates to the file named that way in the directory structure.) So one of the uses of LENV will be to assign default values to variables in analogous cases. The precondition says that the input file has to exist, and the change effected is to store the compiled version of the source program to OUTPUTFILE. Note that *cc* in the MC component is the actual C compiler, invoked by the shell. It is not to be confused with `cc`, which introduces the command line that the shell processes. The shell looks up `cc` in its lexicon and acts accordingly, whereas it simply passes *cc* to the operating system with the appropriate parameters. So, in an actual implementation, the shell will perform the following translation:

$$\texttt{cc}\ \mathit{file} \rightarrow \mathit{cc}\ \text{-}o\ \texttt{a.out}\ \mathit{file}.$$

As here is the first case with a non-empty LENV, we will give the DFOLE translation again:

8.7. Example

$$!(\exists \text{OUTPUTFILE}[\text{OUTPUTFILE} = \texttt{a.out}] \wedge \neg(\mathit{file} = \bot)) \wedge$$
$$\exists \text{OUTPUTFILE.content}$$
$$[!\exists \text{OUTPUTFILE}$$
$$[\text{OUTPUTFILE} = \texttt{a.out}] \wedge$$
$$\text{OUTPUTFILE.content} = \text{cc}(\mathit{file}.\text{content})]$$

Here again we can calculate the semantic value of the DFOLE formula to verify that it coincides with the intended interpretation of our quadruple. Furthermore, we can now see how the closure operator closes off unwanted dynamic effects.

Note that the output stream SCREEN in the example **8.5** also has a default value (namely, `tty`). The two different treatments of SCREEN vs. OUTPUTFILE in examples **8.5–8.6** reflect a distinction that we intend to make between *two types of default values*. The first type, called *deictically available defaults* (*DAD*), are similar to *here* and *now* in natural languages. The default value of SCREEN belongs to this type. Similar default values include KBD (the user's keyboard is the default value for the current input stream), HOME (defaults to the user's home directory) etc. The other type is called *non-deictically available defaults* (*NAD*), which contain all the other default values (for example, the default name of the output file produced by the C compiler in the above example). These are determined by command names lexically. In this respect (but only in this respect) they are similar to lexically determined properties of missing arguments in natural language. For example, the direct object of the verb *eat* in *I am eating* has the default property 'food', which can be overridden by an explicit direct object, as in *I am eating sand*. On the other hand, the optional 'source' argument of the verb

leave as in *He left* defaults to 'here', a deictically available default, and can also be overridden by an explicit argument, as in *He left Los Angeles.* The different treatments of DAD and NAD will allow us to make a similar distinction in our shell language.

9. Putting Command Lines Together

So far, we presented the language of program specifications, which serve as the interpretation of command lines (*cml*, cf. definition **6.2.9**). That is, $[\![cml]\!] \stackrel{\text{def}}{=} [\![\tau_1(cml)]\!]^{\text{ms}}$. We have also seen the objects denoted by arguments. What remains to be done is to explain how other parameters, i.e., flags and options are combined with the lexical program specifications.

To give specifications for these, we need a richer, type theoretical language $L^{(\text{spec})}$. The set of types remains the same as in the case of $\text{L}^{(\text{ps})}$[35] The language itself is the same except that we introduce a new logical constant λ that will serve to construct functions and we now allow application to work in both directions. Further we allow an infinite set of simple variables in all types that we do not indicate explicitly, as they are not excluded by the definition of $\text{L}^{(\text{ps})}$. The following definition only gives the new clauses:[36]

9.1. Definition

1. $L^{(\text{spec})} \stackrel{\text{def}}{=} \langle \text{LC}_{\text{spec}}, \text{Con}, \text{Var}, \text{Expr} \rangle$;
2. $\text{LC}_{\text{spec}} \stackrel{\text{def}}{=} \text{LC}_{\text{ps}} \cup \{\lambda\}$;
7. $\Phi \in \text{Expr}_{\langle \alpha\ \beta \rangle}, \eta \in \text{Expr}_{\beta} \Rightarrow \Phi(\eta), (\eta)\Phi \in \text{Expr}_{\alpha}$;
11. $\Phi \in \text{Expr}_{\alpha}, \xi \in \text{Var}_{\beta} \Rightarrow \lambda\xi.\Phi \in \text{Expr}_{\langle \alpha\ \beta \rangle}$.

Since the set of types is the same, the function D is not altered, either,[37] and the semantics of the expressions present already in $\text{L}^{(\text{ps})}$ is also unchanged,[38] so we only give the new clauses of the definition for the interpretation function $[\![\cdot]\!]$.

9.2. Definition

1. $[\![\Phi_{\langle \alpha\ \beta \rangle}(\eta_{\beta})]\!] = [\![(\eta_{\beta})\Phi_{\langle \alpha\ \beta \rangle}]\!] \stackrel{\text{def}}{=} [\![\Phi]\!]([\![\eta]\!])$;
2. $[\![\lambda\xi.\Phi]\!] \stackrel{\text{def}}{=} \{\langle \eta, \zeta \rangle \in \text{D}(\beta) \times \text{D}(\alpha) \colon \zeta = [\![\Phi]\!]^{\text{ms}}_{v:\xi/\eta}\}$.

The semantic domains agree with the requirements of definition **6.2**. The

---

[35] Cf. definition **7.1.**

[36] For the remaining clauses cf. definition **7.2.**

[37] Cf. definition **7.4.**

[38] Cf. definitions **7.5** and **7.6.**

semantics of compound expressions is given by functional application and the λ-operator corresponds to function abstraction.[39]

In what follows, we will be interested in the semantics of command lines in terms of their constituents, i.e., the command names and the parameters.[40] Again, we assume that there is a translation function τ that works pointwise, i.e., it translates the expressions of the language $L^{(cml)}$ into expressions of the language $L^{(spec)}$. It is easy to show that $[\![\lambda\xi.\Psi(\eta)]\!] = [\![\Psi[\xi/\eta]]\!]$. That is, carrying out β-conversion is licensed by the semantics. Further, we will still represent program specifications by our quadruples and use four functions — 'lenv', 'pc', 'mc' and 'envc' — to refer to its components which we did not introduce into our language explicitely to avoid complications.

As we said earlier, the denotation of a command name is looked up in the *lexicon* associated with the shell. For the sake of simplicity, we are assuming that the lexicon is a static list of specifications rather than a dynamic database (i.e., we do not consider the possibility of *lexical rules*). Therefore, the specification of an n-argument command name (cm_n) as looked up in the lexicon is a lambda-expression of the form $\lambda x_1 \ldots \lambda x_n.\mathrm{PS}$.[41] As a consequence, we cannot account, for the time being, with the mechanisms governing *optional arguments* (and the *deictically available defaults* associated with them). For example, the three versions of `cat` (with no command-line argument, with one argument and with two arguments) must be distinguished as if they were three different command names ($\mathtt{cat}_0$, $\mathtt{cat}_1$ and $\mathtt{cat}_2$, respectively):

9.3. Examples

1. $\tau(\mathtt{cat}_0) \stackrel{\text{def}}{=} \langle \emptyset;$
   $\mathrm{KBD} \neq \bot \wedge \mathrm{SCREEN} \neq \bot;$

---

[39] $v{:}\,x/u$ is the same function as v, except that it assigns u to x, i.e.,:

$$v{:}\,x/u(y) \stackrel{\text{def}}{=} \begin{cases} u & \text{if } x = y; \\ v(y) & \text{elsewhere.} \end{cases}$$

$[\![\cdot]\!]^{\mathrm{ms}}_{[v:x/u]}$ is the interpretation function associated with a machine state under the modified valuation. $\Psi[\xi/\eta]$ is the expression that we get by β-conversion, i.e., by substituting η for all free occurences of ξ in Ψ.

[40] As we mentioned earlier, we will ignore certain expressions, such as *opr*'s. But it is easy to see what the appropriate type for those specifications would be.

[41] Again, subscripts on variables and constants indicating their type will be dropped when it is obvious from the context.

SCREEN.conent = SCREEN.content$^\frown$KBD.content);
{SCREEN.content}⟩;

2. $\tau(\mathtt{cat}_1) \stackrel{\text{def}}{=} \lambda x_{file}.\langle\emptyset;$
$\wedge\, x \neq \bot \wedge \text{SCREEN} \neq \bot;$
SCREEN.content = SCREEN.content$^\frown x$.content;
{SCREEN.content}⟩;

3. $\tau(\mathtt{cat}_2) \stackrel{\text{def}}{=} \lambda x_{file} \lambda y_{file}.\langle\emptyset;$
$\wedge\, x \neq \bot \wedge \wedge y \neq \bot;$
y.content = y.content$^\frown x$.content;
{y.content}⟩;

Assuming that the translation of a file name is the file name itself, the full construction consisting of $\mathtt{cat}_1$ and the file name *file* will get the translation shown in our earlier example **8.5**.

It is easy to see what the lexical rules will do when they will exist: They will abstract over variables with deictically available default values (such as KBD or SCREEN), thereby converting them into obligatory arguments. Which variable must correspond to the first, second and third argument place is determined by lexical principles.[42] Alternatively, we could assume that the addition of optional arguments is a syntactic operation accompanied with a semantic operation working in parallel. To do that, we have to assume that the set DAD of deictically available defaults has two subsets, corresponding to the two possible non-vocative arguments of a command name (cf. footnote 42 above):

$$\text{DAD} = \text{DAD}_1 \cup \text{DAD}_2$$

(the subsets need not be disjoint). For example, variables with deictically available default values corresponding to 'here', 'now', 'me' and 'input' (e.g., HOME, CWD, HOST, KBD) are typically in DAD_1; variables that default to 'output' (e.g., SCREEN, PRINTER) are in DAD_2. Moreover, we also have to stipulate that exactly one variable in each of the two subsets occurs in the program specification of the command. If these conditions are satisfied, then we can interpret the addition of the first optional argument as replacing that member of DAD_1 which

[42] We assume that command names should have at most three argument places as a rule, as is the usual situation in natural languages. However, the first argument (the 'subject', so to say) is always the operating system itself, since commands are in the *imperative* mood. So an implicit 'subject' in the 'vocative' case is to be assumed in front of each command. We also believe that argument places should be associated with certain types of roles in a systematic way, as is the case with the so-called *thematic roles* in natural languages.

figures in the program specification, and the second optional argument overriding the relevant member of DAD_2. Note that the systematic association of argument places with role types (either lexically or through the syntax) is a requirement in terms of our interpretation of the Principle of Compositionality.

Option letters do something very similar to the lexical rules informally mentioned above, except that they affect *non-deictically available defaults*. For example, the option letter `-o` used with the command name `cc` is used to introduce the output file name, thus overriding the default value `a.out`. However, the way such an option letter operates is different from what we outlined in connection with lexical rules. Instead of just abstracting over a variable name, it abstracts over the *value assigned* to a variable in the LENV (local environment) component of the program specification:

9.4. Example

$$\tau(-\mathtt{o}) \stackrel{\text{def}}{=} \lambda\xi.\lambda x_{file}.\xi[\text{lenv}(\xi)/\text{lenv}(\xi)\text{: OUTPUTFILE}/x].$$

That is, the option letter `-o` will add an argument place to the command line that it is attached to, and modify the LENV of the corresponding program specification in such a way that it assigns the newly introduced lambda-variable to the variable OUTPUTFILE (we use the same notation as for the modification of valuations in the semantics, cf. footnote 39). Therefore, the semantics of `-o` is entirely uniform: supplying an option introduced with `-o` will be simply idle if the original program specification does not assign a value to OUTPUTFILE, as expected, whereas it will override the non-deictically available default otherwise.[43] Accordingly, the denotation that we assign to a command line of the form `cc -o` *objfile sourcefile* will be as follows:[44]

9.5. Example

$\tau(\mathtt{cc}\ \mathtt{-o}\ \textit{objfile}\ \textit{sourcefile}) \stackrel{\text{def}}{=} \langle\{\langle \text{OUTPUTFILE}, \textit{objfile}\rangle\};$
$\textit{sourcefile} \neq \perp;$
$\text{OUTPUTFILE.content} = cc(\textit{sourcefile}.\text{content});$
$\{\text{OUTPUTFILE.content}\}\rangle.$

Most importantly, as can be seen in the above examples, the difference between the behaviour of optional arguments vs. options is reflected by the shape

[43] As a matter of fact, we could also say that the shell issues an error message instead of passing the command to the operating system. In natural languages, using an optional parameter when it is not appropriate gives rise to an anomaly. The procedure that we are taking here is just a matter of convenience. Note, however, that the eventual anomaly would be semantic rather than syntactic.

[44] Cf. **8.6** and **9.4**

of the program specifications that they yield. Lexical rules replace variables with lambda-variables, whereas options just rebind them in the local environment. As a consequence, options leave open the possibility of further rebinding, so, e.g., the following equivalence will hold:

$$\texttt{cc -o } \mathit{objfile}_1 \texttt{ -o } \mathit{objfile}_2 \equiv \texttt{cc -o } \mathit{objfile}_2.$$

Lexical rules, on the other hand, make it impossible to rebind the affected variables in any way (e.g., by using options).

Flags (*fl*) are of type $\langle t\ t \rangle$. Just like options, they modify the LENV component, but they determine the value that they assign to variables in the local environment (rather than taking an argument to that effect). For example, the flag `-f` flag forces both WRITECHECK and EXISTCHECK to be evaluated to 0 in the local environment:

9.6. Example

$$\tau(\texttt{-f}) \stackrel{\text{def}}{=} \lambda\xi.\xi[\text{lenv}(\xi)/((\text{lenv}(\xi)\text{: WRITECHECK}/0)\text{: EXISTCHECK}/0)].$$

This is an expression of the appropriate type, i.e., it yields a program specification when applied to a program specification, as in the following example:[45]

9.7. Example

$$\begin{aligned}&\tau(\texttt{rm } \mathit{file} \texttt{ -f}) \stackrel{\text{def}}{=} \langle\{\langle \text{WRITECHECK}, 0\rangle, \langle \text{EXISTCHECK}, 0\rangle\};\\ &\quad (\text{EXISTCHECK} = 1 \rightarrow \mathit{file} \neq \bot) \wedge\\ &\qquad (\text{WRITECHECK} = 1 \rightarrow \mathit{file}.\text{write\_permission} = 1);\\ &\quad \mathit{file} = \bot;\\ &\quad \{\mathit{file}\}\rangle.\end{aligned}$$

Obviously, the effect of the flag `-f` is that the tests on the existence of the file to be removed and the write permission for it will always succeed.

10. Conclusions

The intuitive non-compositionality of the Unix command language is due to the fact that, in every command line, the interpretation of the parameters is the 'internal affair' of the program corresponding to the command name. Even if the meaning of a command line is some function of the meanings of its constituents, one has the clear intuition that not all functions yield equally 'compositional' semantics. If we allow functions defined pointwise, then the traditional principle of compositionality becomes vacuous. On the other hand, it is difficult to make

[45] Cf. the examples **8.3** and **9.6**.

sense of the concept of 'more natural' or 'simpler' functions from the mathematical point of view. Accordingly, there are no natural means to limit the action that a computer program can perform. What we can do, though, and what we have done in this paper, is interpreting the constituents of command lines as well as their ways of combination in a uniform manner. In this way, the interpretation of command lines is compositional in the sense that it may not be construction specific, irrespective of what the actual program carried out by the machine will do.

How is the behaviour of our compositional Unix shell different from a traditional, non-compositional one? Instead of pre-defining a set of flags, option letters, optional arguments etc. for each command name, we could have 'manual pages' for flags, option letters and the like, which would describe what they do in *any* command line. If certain combinations of commands and parameters do not make sense, they will qualify as *semantic anomalies* rather than *syntactic errors*, just like in natural languages. (Although, in actual fact, we have treated certain anomalies as just ineffective in the above.) For example, *I knew the answer with a knife* is anomalous because the verb *know* does not license an instrument 'option' just the same as the option in `cat -file` *auxfile* does not make sense because the program `cat` does not use any auxiliary (command or expression) file. Listing what arguments, options etc. *do* make sense in combination with `cat` is as absurd as it would be to list all the adjuncts that make sense with the verb *know* in a dictionary of English.

By the same token, our treatment of Unix commands also has implications as to the 'manual pages' of natural-language predicates, i.e., their syntactic and semantic description. Even though the complexity of their meanings and the multitude of the types of construction that they occur in cannot be compared to the simplified command language that we have examined in this paper, the concept of compositionality that we have forwarded here has a consequence for their study. Namely, a natural-language predicate used in two different ways must be attributed a single meaning unless we are entirely certain that homonymy or two different constructions are responsible for the two different uses.

References

Groenendijk, J. and M. Stokhof. 1991. 'Dynamic Predicate Logic'. *Linguistics and Philosophy* **14**.

Gunter, C.A. and D.S. Scott. 1990. 'Semantic domains'. In: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*. Vol. B. North-Holland, Amsterdam.

Mosses, P.D. 1990. 'Denotational semantics'. In: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*. Vol. B. North-Holland, Amsterdam.

Scott, D.S. and C. Strachey. 1971. 'Toward a mathematical semantics for computer languages'. *Proc. Symp. on Computers and Automata* **21**, 19–46. Microwave Research Institute Symposia Series, Polytechnic Institute of Brooklyn.