Science of Computer Programming 11 (1988/89) 263-297
North-Holland

FROM INFORMAL REQUIREMENTS TO A RUNNING PROGRAM: A CASE STUDY IN ALGEBRAIC SPECIFICATION AND TRANSFORMATIONAL PROGRAMMING

H. PARTSCH
Informatica, Katholieke Universiteit Nijmegen, Tournooiveld, 6525 ED Nijmegen, The Netherlands

Communicated by M Sintzoff
Received November 1987
Revised July 1988

Abstract. Algebraic specification and transformational programming have been advocated as new approaches to the development of software, in order to solve some of the technical problems in software engineering such as "early validation", correctness of implementations, re-usability of software, or re-usability of software design By means of a nontrivial example, viz an interactive text editor, we demonstrate that the combined use of these approaches allows to bridge the gap between verbally stated requirements and a running program, even for non-toy, realistic problems

1. Introduction

Algebraic specification and transformational programming have been advocated as new approaches to solving some of the technical problems in software engineering such as "early validation", correctness of implementations, re-usability of software, or re-usability of software design.

A lot of effort has been invested and is still spent on research in algebraic specification and transformational programming (for references to specific literature, cf. e.g. [9], [15] or [6]). We will follow the basic philosophy and the methodological lines as investigated in the Munich CIP project (cf. [4] and [13]).

For denoting algebraic types and programs we essentially use the Algol-like variant of the language CIP-L (cf [2]), the essence of which we assume the reader to be familiar with. We also assume having available algebraic definitions of certain basic data structures (for respective formal definitions, cf. [2] and [12]) such as.

- (extended) *sequences* (defined by a type ESEQU) with

ε	denoting the empty sequence,
first resp. **last**	denoting the first (resp. last) element,
rest resp. **lead**	denoting the remainder, after removing the first (resp last) element, and
$+$	denoting concatenation (where attaching a single element is subsumed as a special case).

(**first**, **last**, **rest**, and **lead** are partially defined operations being undefined for empty sequences Furthermore, in order to avoid excessive bracketing, these four operations are assumed to have higher priority than concatenation and test operators.)

- *stacks* (defined by a type STACK) as sequences with restricted access, with

ε	denoting the empty stack,
top	denoting the top element of the stack,
pop	denoting the remainder of the stack, after removing the top element, and
push	denoting the addition of an element to a stack.

(Again, **top** and **pop** are partially defined operations being undefined for empty stacks.)

- *tuple structures*, where ⟨..⟩ is used for denoting the tuple constructor and individual identifiers are used for the respective selectors. In the sequel, we will just use triples (defined by an appropriate type TRIPLE).

Rather than using the syntax of type instantiation as in CIP-L, we shortly write e g.

mode input = ESEQU(**char**)

to stand for an instantiation of the type scheme ESEQU with **char** (where just the sort **sequ** is renamed to **input** and all other operations remain the same). Apart from this slight deviation, we use the CIP-L type mechanism as it is defined in [2] and [17].

When demonstrating any methodology by means of examples, one is always bothered by a trade-off between the size of an example and its comprehensibility. Small problems are easily understood and thus can be used to stress methodological aspects in all details. However, it is hard to convince a practitioner on the basis of such small, unrealistic toy examples. Large problems, on the other hand, need a lot of effort to explain the problem proper, to motivate design decisions, or to discuss technical details, and run the risk of losing the methodological essence.

The example we are going to deal with—an interactive text editor—can be seen as a kind of a compromise between small problems and realistic ones: it is realistic, as it is "actually a subset of an editor which has been implemented on several machines" (cf. the problem formulation in [16]), yet it is not too large for demonstrating methodological principles without getting lost in problem-specific technical details Since we mainly aim at conveying methodology, rather than just doing an example, we will deliberately skip details which we believe to be obvious (with straightforward solutions) or which would lead to an inadequate level of detail.

According to its title the emphasis of this paper is on demonstrating how to formalize an informally stated problem with the algebraic specification technique and on how to transform the formal specification into a running program. The first aspect already has been tackled with the same example in [11]. Nevertheless, we will deal with this part here again for two reasons; first, we want to make this paper self-contained, second, we have made some (minor) changes in the formal specification, mainly to keep the presentation at a reasonable length.

The text editor is one example out of a collection of examples presented at the 26th meeting of IFIP working group WG 2.1. This collection was intended and successfully used as a common basis for discussing various specification and program development techniques. In several discussions, this particular example has turned out to be a real challenge for specification languages and specification methodologies as well as for program development techniques, probably due to the difficulty inherent in giving a reasonably high-level treatment of a system with an obviously low-level procedural appearance.

The verbal statement of the problem is literally quoted from [16]

EXAMPLE NUMBER 4—TEXT EDITOR

This problem asks for the implementation of a simple line oriented editor for use in an interactive environment The specification is actually a subset of an editor which has been implemented on several machines

The input device is a keyboard/display Input from the keyboard is obtained one character at a time and is one of the following characters letters, digits, blank, {cr} (line return), {esc} (escape), {bs} (backspace)

The display device is controlled by outputting single characters from the following set letters, digits, blank, {cr} (line return), {bs} (backspace), {bel} (sound alarm)

Note that the keyboard and display are completely independent, "echoing" of input characters must be done by the program The effect of sending a {cr} is to roll the screen up and set the cursor to the start of the next line {bs} moves the cursor back one (no effect if at start of line) {bel} sounds an alarm, but has no effect on the cursor position

The current text is a sequence of lines which can be modified by entering edit commands There is no need to consider the problem of opening files, reading text etc Assume that the current text is available as a variable (or equivalent)

The editor outputs a prompt character "?" to invite entry of commands The following commands are available

Note that most commands are not terminated by a {cr}

b	position to start of first line in file and print out the first line
e	position past last line of fi[illegible]and display ⟨end-of-file⟩.
m⟨*old*⟩{cr}⟨*new*⟩{cr}	search for occurrence of the string ⟨*old*⟩ in the current line, and replace it by string ⟨*new*⟩ Display the modified line
i	enter insert mode The cursor moves to the start of the next line and text can be entered (one or several lines) which is inserted following the current line Each new line is terminated by a {cr} To leave insert mode, {esc} is typed after the last inserted line
k	delete the current line and store it in a stack (see u command) Display the line after the one deleted
u	retrieve the line on top of the delete stack and insert it just before the current line, then display the retrieved line (note the sequence ku leaves the file as it was)
n+	move n (decimal integer) lines forward in the file and type new current line
n−	move n (decimal integer) lines backward in the file and type new current line

s⟨*key*⟩{cr}	search forward from the current position for a line containing the string ⟨*key*⟩, and display the line
a⟨*rep*⟩{cr}	alter the ⟨*key*⟩ just found by the search (s) command to the given replacement ⟨*rep*⟩ Display the modified line
f	forget (*undo*) the effect of the m or a command just entered
w	display window consisting of the nine lines either side of the current line

The {bel} character is transmitted to the display if any command syntax error is detected or for an unsuccessful search etc The backspace key may be pressed during entry of any text string, the effect is to erase the previous character and backup the cursor If the backspace key is used to erase the m or s character which starts a command, then a completely new type of command can be entered

We decided to use this informal problem description without any changes, since we aimed at a "realistic" case study:

- It is a fact of real-world software development that an informal problem description and a formalization of the problem are given by different persons.
- Dealing with a problem stated by someone else forces oneself to solve even nasty problems and thus prevents oneself from cheating by adapting the problem statement to fit a nice and elegant treatment.
- Formalizing an informal problem statement given by someone else entails a lot of design decisions which would never have appeared in a self-made problem statement. These design decisions give us an opportunity to comment on alternatives with respect to the specification

2. Formal specification

Given an informal description of a problem, in any approach to formal specification a key question is "how do we find the formalization in a systematic way?". Therefore we will start this section with some general considerations on methodology

2.1. Methodological considerations

For our particular case study and similar ones (cf., e.g., [3]) a "strictly top-down" approach using "hierachical decomposition" and "stepwise detailization" turned out to be adequate and profitable.

After having decided on an adequate "concept" (cf. [12]) underlying the formal specification, the formalization process starts by first describing the overall behaviour of the system as a function on suitably defined abstract objects which model the interface between the system and its environment.

At least for non-toy systems it will usually be hard to define this function just in terms of those objects (characterizing the system's interface) and their basic operations. Therefore, in a first decomposition step further "internal" object kinds and operations are introduced to be used in the definition of the system's function In this way a complex system function is decomposed into smaller, and hopefully more easily manageable functions.

Frequently part of the definition of the newly introduced operations over such "internal" object kinds follows immediately from the original, informal description ("detailization"). For the remaining parts of the respective definition (which often are only detected after a check for formal completeness) we proceed as above, i.e. we give meaning to functions by introducing new object kinds and new operations (i e. further "decomposition") which, in turn, are defined by detailization and/or further decomposition. This iterated process ends as soon as we have arrived at a suitable level of pragmatics where either all object kinds and operations are completely defined or only refer to object kinds and operations that are supposed to be known.

Within the framework of algebraic specification the combination of decomposition and detailization just described amounts to introducing a new type: "decomposition" roughly corresponds to introducing the signature of a type (i e. the syntactic part) whereas "detailization" aims at providing meaning in the form of appropriate axioms for the object kinds and operations introduced by the preceding decomposition step. Particularly helpful in this context is the notion of "sufficient completeness" (cf. [17]) which supports establishing completeness for each level of a hierachical specification during construction (cf. also [14]).

Proceeding top-down here has the additional advantage that new object kinds and operations are only introduced "by need" rather than by somehow combining existing operations into more complex ones without knowing whether they are actually needed as in the strict bottom-up case. (Note that this does not imply that top-down processing forbids the use of existing types and type schemes.) In this sense top-down processing minimizes the number of such "internal" object kinds and operations.

2.2. *The overall behaviour of the editor*

In contrast to many other problems, finding a suitable concept for the specification is a critical issue for the editor problem, since it is an interactive system.

2.2.1. *A concept for specifying interactive systems*

Interactive systems are communicating systems. From this viewpoint it would be straightforward to specify an interactive system using the formalisms developed for describing communicating systems (e.g. CSP [7], CCS [10] or recursive stream equations [5]). On the other hand, interactive systems are very specific communicating systems with only two (communicating) processes, viz. the user and the system.

In addition, communication between these two processes takes place in a restricted, rather "disciplined" way, viz in the form of a "dialogue".

A dialogue in an interactive system can be seen (cf. [8]) as an alternating sequence $i_1 o_1 i_2 o_2 \cdots i_k o_k$ of "inputs" i_n and "outputs" o_m. An i_n usually originates from the user's unconstrained will, but an o_m surely depends on the respective i_m and probably also on the "history", i.e , the i_n with $n < m$:

$$\left.\begin{array}{l} \text{input } i_1 \to \text{output } o_1 \\ \text{input } i_2 \to \text{output } o_2 \\ \quad\vdots \qquad\quad \vdots \\ \text{input } i_n \to \text{output } o_n \end{array}\right\} \text{"visible effect"}$$

Thus, in an abstract view an interactive system can be seen as a "function" from input sequences to output sequences. Consequently, its behaviour can be adequately specified by defining the system's reaction to arbitrary input sequences

Following these lines the overall behaviour of the editor can be specified by a function

funct *edit* = (**input** *in*) **output**

where the object kind **input** characterizes sequences of characters, i.e.

mode input = ESEQU(**char**)

and the object kind **output** characterizes sequences of certain actions to be observed by the user of the editor, i.e.

mode output = ESEQU(**action**).

Here, **char** and **action** are considered as primitive object kinds. A further detailization can be found later.

2.2 2 Decomposition into subtasks

The formalization of the problem stated in Section 1 now aims at giving a definition for *edit* Since, due to the complexity of the problem, such a definition is not at all straightforward, we follow the methodological line sketched above, and decompose the function *edit* into smaller, better manageable logical units, viz.

- mapping from (concrete) input character sequences to sequences of (abstract) commands;
- effects of commands (or sequences of commands) within the system;
- mapping from (abstract) effects to (concrete) physical output.

The objects to be manipulated by sequences of commands are texts, outputs (whatever that means), and an internal stack (see commands k and u). For the current level of refinement, however, we can abstract from these details and combine the objects that are manipulated into the notion of a "state". Thus the essential part of the editor will be a mapping

funct *effect* = (**commsequ** *cs*, **state** *s*) **state**,

(where **commsequ** and **state** denote the object kinds characterizing "sequences of commands" and "states") which associates a new state with each sequence of abstract commands and a given state.

We use

funct *parse* = (**input** *in*) **commsequ**

for mapping concrete input to abstract commands,

state *initstate*

for providing an initial "state", and

funct *unparse* = (**state** *s*) **output**

for mapping abstract output to a concrete one, such that the overall behaviour of the editor can be completely specified by

edit(*in*) = *unparse*(*effect*(*parse*(*in*), *initstate*)).

According to our global development strategy, we now aim at providing definitions for the newly introduced operations *effect*, *parse*, and *unparse*, and for the constant *initstate*. As outlined above, this either means considering individual cases or referring to further new operations.

2.3. Translation between external and internal representation

The translation between external and internal representation of sequences of commands is to deal with the operations *parse* and *unparse*. However, we will just concentrate on the function *parse* and not deal with the formal specification of *unparse* mainly for two reasons:

- The informal specification is fairly imprecise about output Thus, a formalization would require a proper extension of the informal description that goes substantially beyond the original problem description. Our goal, however, is to solve the task as given in [16].
- If the information about output had been more precise and complete, a formalization of *unparse* would be rather analogous to the one of *parse* (as dealt with below) and thus would add no real substance to our considerations.

2.3.1 Definition of the data structures involved

In order to be able to give a formal specification of

funct *parse* = (**input** *in*) **commsequ**,

we first have to define the data structures involved.

For being complete, the definition

mode input = ESEQU(**char**)

(as introduced above) requires a further definition of admissible characters

Therefore, according to the informal requirements, we define

mode letter = (a, b, c, . .),

and

mode digit = (0, 1, 2, . .)

The set of possible input characters is then characterized by

mode char = **letter**|**digit**|**blank**|{bs}|{esc}|{cr}

(where we have simply adopted the notation for the special characters that was used in the informal description).

For the definition of *parse* we will also need input strings that do not contain the characters {cr} and {esc}. These strings are defined by

mode string = ESEQU(**char** *c*. *c* ∉ {cr, esc}).

Furthermore we need sequences composed of digits and {bs}, defined by

mode dstring = ESEQU(**digit**|{bs}).

According to the informal description commands are either simple, consisting of a single letter (such as, e g., b) or are "parameterized", consisting of a letter and "arguments" (such as, e.g , m⟨*old*⟩⟨*new*⟩). For the definition of commands we use

mode command = (b|e|k|u|f|w|m|s|a|i|+|−|i′|ec),

where

- b, e, k, u, f, and w are to denote the respective (simple) commands;
- m, s, a, i, +, and − are shorthand notations for the respective "parameterized" commands They are supposed to stand for tuples consisting of the respective command letter and the types of the "arguments" of the command. Thus, e g , m abbreviates (m, **string**, **string**), s is short for (s, **string**), + is short for (+, **nat**), etc,
- i′ and ec are commands produced by *parse* that are not available to the user; ec denotes an additional error command in case of syntactically illegal input (cf. below) and i′ is used for the decomposition of an insertion command into linewise insertions (cf also below).

For sequences of commands we simply use the definition

mode commsequ = ESEQU(**command**).

2.3.2. The problem of backspaces

According to the informal description a {bs} character may erase certain preceding characters. In case of certain "parameterized" commands it even may be used to erase the respective command letter

One might be tempted to try a specification where first all {bs} characters are removed from the input string before parsing takes place. However, a closer look at the informal description tells us immediately that the erasing effect of a {bs} character depends on the respective state of the parsing process when the {bs} is detected.

Thus, e.g., an input string of the form

(*) b{bs}k{bs}{bs}c...

has the same effect as

bkc...

provided a new command is currently being recognized. If, however, the string (*) is the beginning of a text to be inserted, then it has the same effect as

c...

(if we assume that an i command cannot be undone by {bs}, cf below). And if the above string (*) is the beginning of the argument for a search, then even the preceding command letter is to be erased.

It is exactly for this interrelation between parsing and backspace removal why it is appropriate to specify the effect of backspaces jointly with parsing rather than specifying both tasks independent of each other

For modelling the effect of {bs} we will use the auxiliary operations

funct *mbl* = (**string** *t*) **nstring** ("move backspaces left"),

and

funct *dlb* = (**nstring** *n*) **nbsstring** ("delete leading backspaces"),

where

mode nbsstring = (**string** s. {bs} $\notin s$),

defines strings containing no backspaces,

mode bsstring = ESEQU({bs}),

defines strings consisting of backspaces only, and

mode nstring = (**string** s: $\exists$ **nbsstring** n, **bsstring** b: $s = b + n$)

defines ("normalized") strings having backspaces only at the beginning

The operation *mbl* deletes pairs of characters consisting of a character (different from {bs}) and an immediately following {bs} character. Remaining {bs} characters accumulate to the left of the resulting string, as can be seen from the property

$mbl(\{bs\}+s)=\{bs\}+mbl(s)$. Formally this operation is defined by

$$\forall\ \textbf{string}\ s, s_1, s_2, \textbf{char}\ c.\ c \notin \{cr, esc\}:$$

$$mbl(s)=mbl(s_1+s_2)\ \textbf{provided}\ s=s_1+c+\{bs\}+s_2,$$

$$mbl(s)=s\ \textbf{provided}\ s\neq s_1+c+\{bs\}+s_2.$$

The operation *dlb* simply skips all leading {bs} characters (from a string where all backspaces are at the beginning) until a character different from {bs} is encountered. It is defined by

$$dlb(\varepsilon)=\varepsilon,$$

$$dlb(\{bs\}+s)=dlb(s),$$

$$dlb(c+s)=c+s\ \textbf{provided}\ c\neq\{bs\}.$$

For both operations completeness of the definition is obvious.

For the formal specification of parsing it is convenient to have furthermore an auxiliary predicate that checks for a given string whether all backspaces can be removed:

$$\textbf{funct}\ nbs=(\textbf{string}\ t)\ \textbf{bool}$$
$$mbl(t)=\varepsilon\ \nabla\ \textbf{first}(mbl(t))\neq\{bs\}$$

(where ∇ denotes sequential disjunction)

2.3.3. Parsing correct input

Now we have all prerequisites to give a formal specification for the operation *parse*. We first concentrate on those cases that are obvious from the verbal problem description

Assuming the general quantification

$$\forall\ \textbf{input}\ sc, \textbf{char}\ c, \textbf{string}\ t_1, t_2, \textbf{dstring}\ t_d:$$

we define *parse* as follows:

Empty input yields an empty command sequence, i.e.,

$$parse(\varepsilon)=\varepsilon.$$

A single b, e, k, u, f, w character is recognized as the respective command, i.e.,

$$parse(c+sc)=c+parse(sc)\ \textbf{provided}\ c\in\{b, e, k, u, f, w\}.$$

With respect to the m, s, and a command we have to differentiate according to the form of their first "argument". If all backspaces in this argument can be removed, i e. if $nbs(t_1)$ holds, we follow the verbal description and define

$$parse(m+t_1+\{cr\}+t_2+\{cr\}+sc)=(m, mbl(t_1), dlb(mbl(t_2)))+parse(sc)$$
$$\textbf{provided}\ nbs(t_1),$$

$$parse(\mathrm{s}+t_1+\{\mathrm{cr}\}+sc)=(\mathrm{s},\ mbl(t_1))+parse(sc)\ \textbf{provided}\ nbs(t_1),$$

$$parse(\mathrm{a}+t_1+\{\mathrm{cr}\}+sc)=(\mathrm{a},\ mbl(t_1))+parse(sc)\ \textbf{provided}\ nbs(t_1).$$

In case $\neg nbs(t_1)$ holds, $mbl(t_1)$ starts with a backspace which erases the preceding command letter. Therefore we specify

$$parse(c+t_1+sc)=parse(\mathbf{rest}(mbl(t_1))+sc)$$
$$\textbf{provided}\ \neg nbs(t_1)\wedge c\in\{\mathrm{m},\mathrm{s},\mathrm{a}\}.$$

Note that this axiom implies the axiom

$$parse(c+\{\mathrm{bs}\}+sc)=parse(sc)\ \textbf{provided}\ c\in\{\mathrm{m},\mathrm{s},\mathrm{a}\}$$

which formalizes the informal requirement that the backspace key may be used to erase an m or s character which starts a command.

With respect to the i command we have to make a decision. Obviously, the i command could be specified analogous to the m, s, or a command above. However, taking into account the particular "insertion mode" mentioned in the informal description, it also seems reasonable to assume that an i character starting an insertion command cannot be erased by a following {bs}. Deciding in favour of the latter possibility we specify:

$$parse(\mathrm{i}+t_1+\{\mathrm{cr}\}+sc)=(\mathrm{i},\ dlb(mbl(t_1))+\{\mathrm{cr}\})+parse(\mathrm{i}+sc),$$

$$parse(\mathrm{i}+t_1+\{\mathrm{esc}\}+sc)=(\mathrm{i}',\ dlb(mbl(t_1))+\{\mathrm{cr}\})+parse(sc)$$

Here the pseudo-command i′ is used to signal that the last line of an insertion command has been recognized.

A correct move command requires a non-empty sequence of digits followed by a + or − sign. Hence, parsing a move command can be specified by

$$parse(t_\mathrm{d}+sig+sc)=(sig,\ conv(mbl(t_\mathrm{d}),0))+parse(sc)$$
$$\textbf{provided}\ (mbl(t_\mathrm{d})\neq\varepsilon\ \Delta\ \mathbf{first}(mbl(t_\mathrm{d}))\neq\{\mathrm{bs}\})\wedge sig\in\{+,-\}$$

(where Δ denotes sequential conjunction).

Here we have used another auxiliary operation, viz.

funct *conv* = (**nbsdstring** *d*, **nat** *n*) **nat**
("convert" a string of digits into a number),

where

mode nbsdstring = ESEQU(**digit**),

characterizes strings consisting of digits only.

conv describes the translation of a sequence of digits into a decimal number. It is formally defined by

$$conv(\varepsilon, in)=in,$$

$$conv(n+sd, in)=conv(sd, in\times 10+n).$$

2.3.4 Parsing incorrect input

One of the major benefits of using the algebraic specification technique is the availability of the theoretical notion of "sufficient completeness" (cf. [17]) that considerably helps in detecting incompleteness, even in a seemingly complete verbal specification as ours In our particular example, for instance, we immediately find out that the informal requirements contain no information on how to react to erroneous situations which may come up in connection with syntactically wrong or incomplete input

Obviously, incomplete input has no effect, i.e., yields an empty command sequence:

$$parse(c+t_1)=\varepsilon \textbf{ provided } c\in\{\mathrm{m},\mathrm{a},\mathrm{s},\mathrm{i}\}\wedge nbs(t_1),$$

$$parse(\mathrm{m}+t_1+\{\mathrm{cr}\}+t_2)=\varepsilon \textbf{ provided } nbs(t_1),$$

$$parse(t_\mathrm{d})=\varepsilon \textbf{ provided } mbl(t_\mathrm{d})\neq\varepsilon$$

Contrary to this, syntax errors result in an "error command" ec. Syntax errors occur, whenever, in a situation where a command is expected, a character appears that does not start a legal command:

$$parse(c+sc)=\mathrm{ec}+parse(sc)$$
$$\textbf{provided } c\notin(\{\mathrm{b},\mathrm{e},\mathrm{m},\mathrm{i},\mathrm{k},\mathrm{u},\mathrm{s},\mathrm{a},\mathrm{f},\mathrm{w}\}\cup\textbf{digit}).$$

Another erroneous situation is the illegal use of an {esc} character in connection with an m, s, or a command

$$parse(c+t_1+\{\mathrm{esc}\}+sc)=\mathrm{ec}+parse(sc) \textbf{ provided } nbs(t_1)\wedge c\in\{\mathrm{m},\mathrm{a},\mathrm{s}\},$$

$$parse(\mathrm{m}+t_1+\{\mathrm{cr}\}+t_2+\{\mathrm{esc}\}+sc)=\mathrm{ec}+parse(sc) \textbf{ provided } nbs(t_1).$$

Finally, an error situation occurs, if a sequence of digits is not followed by a + or − sign:

$$parse(t_\mathrm{d}+c+sc\}=\mathrm{ec}+parse(sc)$$
$$\textbf{provided } (mbl(t_\mathrm{d})\neq\varepsilon\ \Delta\ \textbf{first}(mbl(t_\mathrm{d}))\neq\{\mathrm{bs}\})\wedge c\notin(\{+,-\}\cup\textbf{digit}).$$

Here we have specified a very simplistic view of error handling. We simply indicate the presence of an error or ignore incomplete input. Of course, we also could have specified a much better error diagnosis by informing the user on the kind of error that has occurred With little additional effort, we also could have made a specification in such a way that in case of an incomplete command the system attempts to process the available part of the input and requests the missing part from the user With respect to the length of our presentation, however, we have abandoned this possibility.

Above, we have treated leading {bs}, {esc}, or {cr} characters as an erroneous situation One also could imagine simply ignoring them, but only at the expense of an additional axiom for *parse.*

2.3.5. Summary

In order to provide for easier checks of consistency and formal completeness, we now simply summarize the definition of *parse* by collecting all axioms given so far As a kind of first step towards an implementation, in this summary we rearrange and rephrase the above axioms in such a way that the left-hand sides of the axioms just differentiate between empty and non-empty input sequences and that, furthermore, in the case of non-empty input sequences, axioms concerning the same leading character are grouped together:

∀ input in, sc, **char** c, **digit** d, **string** t_1, t_2, **dstring** t_d:

$parse(\varepsilon) = \varepsilon$,

$parse(c + in) = c + parse(in)$ **provided** $c \in \{\mathrm{b, e, k, u, f, w}\}$,

$parse(c + in) = \mathrm{ec} + parse(in)$
  provided $c \notin (\{\mathrm{b, e, m, i, k, u, s, a, f, w}\} \cup$ **digit**),

$parse(\mathrm{m} + in) = (\mathrm{m}, mbl(t_1), dlb(mbl(t_2))) + parse(sc)$
  provided $in = t_1 + \{\mathrm{cr}\} + t_2 + \{\mathrm{cr}\} + sc \wedge nbs(t_1)$,
$parse(\mathrm{m} + in) = parse(\mathbf{rest}(mbl(t_1)) + sc)$ **provided** $in = t_1 + sc \wedge \neg nbs(t_1)$,
$parse(\mathrm{m} + in) = \varepsilon$
  provided ($in \in$ **string** Δ $nbs(in)) \vee (in = t_1 + \{\mathrm{cr}\} + t_2 \wedge nbs(t_1))$,
$parse(\mathrm{m} + in) = \mathrm{ec} + parse(sc)$
  provided $(in = t_1 + \{\mathrm{cr}\} + t_2 + \{\mathrm{esc}\} + sc \wedge nbs(t_1))$
  $\vee\ (in = t_1 + \{\mathrm{esc}\} + sc \wedge nbs(t_1))$,

$parse(\mathrm{i} + in) = (\mathrm{i}, dlb(mbl(t_1)) + \{\mathrm{cr}\}) + parse(\mathrm{i} + sc)$
  provided $in = t_1 + \{\mathrm{cr}\} + sc$,
$parse(\mathrm{i} + in) = (\mathrm{i}', dlb(mbl(t_1)) + \{\mathrm{cr}\}) + parse(sc)$
  provided $in = t_1 + \{\mathrm{esc}\} + sc$,
$parse(\mathrm{i} + in) = \varepsilon$ **provided** $in \in$ **string**,

$parse(\mathrm{s} + in) = (\mathrm{s}, mbl(t_1)) + parse(sc)$ **provided** $in = t_1 + \{\mathrm{cr}\} + sc \wedge nbs(t_1)$,
$parse(\mathrm{s} + in) = parse(\mathbf{rest}(mbl(t_1)) + sc)$ **provided** $in = t_1 + sc \wedge \neg nbs(t_1)$,
$parse(\mathrm{s} + in) = \varepsilon$ **provided** $in \in$ **string** Δ $nbs(in)$,
$parse(\mathrm{s} + in) = \mathrm{ec} + parse(sc)$ **provided** $in = t_1 + \{\mathrm{esc}\} + sc \wedge nbs(t_1)$,

$parse(\mathrm{a} + in) = (\mathrm{a}, mbl(t_1)) + parse(sc)$ **provided** $in = t_1 + \{\mathrm{cr}\} + sc \wedge nbs(t_1)$,
$parse(\mathrm{a} + in) = parse(\mathbf{rest}(mbl(t_1)) + sc)$ **provided** $in = t_1 + sc \wedge \neg nbs(t_1)$,
$parse(\mathrm{a} + in) = \varepsilon$ **provided** $in \in$ **string** Δ $nbs(in)$,
$parse(\mathrm{a} + in) = \mathrm{ec} + parse(sc)$ **provided** $in = t_1 + \{\mathrm{esc}\} + sc \wedge nbs(t_1)$,

$$
\begin{aligned}
& parse(d+in) = (sig, conv(mbl(t_d), 0)) + parse(sc) \\
& \quad \textbf{provided } in = t_d + sig + sc \\
& \qquad\qquad \wedge (mbl(t_d) \neq \varepsilon \,\Delta\, \textbf{first}(mbl(t_d)) \neq \{\text{bs}\}) \wedge sig \in \{+, -\}, \\
& parse(d+in) = parse(\textbf{rest}(mbl(t_1)) + sc) \textbf{ provided } in = t_1 + sc \wedge \neg nbs(t_1), \\
& parse(d+in) = \varepsilon \textbf{ provided } in \in \textbf{dstring} \,\Delta\, mbl(d+in) \neq \varepsilon, \\
& parse(d+in) = \text{ec} + parse(sc) \\
& \quad \textbf{provided } d + in = t_d + c + sc \\
& \qquad\qquad \wedge (mbl(t_d) \neq \varepsilon \,\Delta\, \textbf{first}(mbl(t_d)) \neq \{\text{bs}\}) \wedge c \notin (\{+, -\} \cup \textbf{digit}).
\end{aligned}
$$

Now the proofs of sufficient completeness and soundness (i.e , non-overlapping left-hand sides) of the specification are straightforward, although not trivial. These are left as an exercise to the interested reader.

2 4. *The kernel of the editor*

The "kernel" of the editor (or its "abstract", "internal" behaviour) is captured by the operation *effect* that maps a sequence of commands and a state to a new state The state transition affected by a single command is described by an auxiliary operation *apply*

2.4 1. *The effect of sequences of commands*

A sequence is either empty, or it can be decomposed into an individual element and a sequence. Thus an unreflected formalization would immediately lead to

$$effect(\varepsilon, s) = \cdot\cdot$$

$$effect(c + cs, s) = \cdots$$

where s denotes an arbitrary state, cs a sequence of commands, and c an individual command.

However, at a closer inspection of the verbal requirements for an individual command we will soon find out that there are commands of different quality, namely commands having effects on states, and others (such as a or f), having effects on other commands. In particular this latter observation suggests that in our formalization below always two leading elements of a sequence should be considered simultaneously

Using the definitions of **command** and **commsequ** as introduced in the previous section, the overall behaviour of the editor may be specified, according to the verbal description, by

$$\forall \textbf{ command } c, c', \textbf{commsequ } cs, \textbf{state } s:$$

(1) $effect(\varepsilon, s) = s$,
(2a) $effect(c + \varepsilon, s) = s$ **provided** $c \in \{\text{a}, \text{f}\}$,
(2b) $effect(c + \varepsilon, s) = apply(c, s)$ **provided** $c \notin \{\text{a}, \text{f}\}$,
(3a) $effect(c + c' + cs, s) = effect(c' + cs, s)$ **provided** $c \in \{\text{a}, \text{f}\}$,
(3b) $effect(c + c' + cs, s) = effect(c' + cs, apply(c, s))$
provided $c \notin \{\text{a}, \text{f}\} \wedge c' \notin \{\text{a}, \text{f}\}$,
(3c) $effect(c + \text{f} + cs, s) = effect((\text{s}, key) + cs, s)$ **provided** $c = (\text{m}, key, rep)$,
(3d) $effect(c + c' + cs, s) = effect((\text{m}, key, rep) + cs, apply((\text{s}, key), s))$
provided $c' = (\text{a}, rep) \wedge c = (\text{s}, key)$,
(3e) $effect(c + \text{f} + cs, s) = effect(c + cs, s)$ **provided** $c \notin \{\text{a}, \text{f}, \text{m}\}$,
(3f) $effect(c + \text{a} + cs, s) = effect(c + cs, s)$ **provided** $c \notin \{\text{a}, \text{f}, \text{s}\}$.

Here we deliberately have used shorthand notations such as e g. $c \notin \{\text{a}, \text{f}\}$ for $\neg\exists$ **string** x: $c = (\text{a}, x) \wedge c \neq \text{f}$.

Of course, we could have combined, e.g , axioms (2a) and (3a) into one axiom. However, the form used above has the advantage that the formal completeness of this specification, as well as its soundness, are obvious: Axiom (1) covers empty command sequences; axioms (2a) and (2b) deal with the case of singleton command sequences; and the axioms (3a)-(3f) deal with all possibilities of command sequences containing at least two elements.

In fact, the above specification is essentially equivalent to the one given in [11], the only differences being

- an "evaluation" of command sequences from "left-to-right" rather than "right-to-left";
- a different way of processing consecutive b or e commands.

As before, the above specification also had to take care of problems not mentioned in the informal requirements, such as, e.g.,

- "what is the effect of an f command applied after a command that is not an a or m command?"
- "what is the effect of an a command applied to a command different from an s command?"

both of which were simply decided to have no effect.

2.4.2. *Definition of state*

In order to be able to specify the function *apply* used in the above specification, we first have to define what a "state" is.

As already outlined, the abstract notion of state served for collectively referring to the "internal" objects of the editor, i.e. the text file, the delete stack (see commands k and u) and the output sequence. Thus, an obvious definition for state is a triple consisting of these entities, i.e.,

mode state = TRIPLE(**file**, **stack**, **output**).

Consequently, the constant *initstate* denoting the initial state can be defined by

state *initstate* $= \langle \varepsilon_f, \varepsilon_d, \varepsilon_o + ? \rangle$

where ε_f denotes the empty file, ε_d the empty stack, and ε_o the empty output. The "?" in the output sequence reflects the verbal requirement that "the editor outputs a prompt character "?" to invite the entry of commands".

Of course, one also could imagine other definitions of *initstate* such as

state *initstate* $= \langle init(tf), \varepsilon_d, \varepsilon_o + ? \rangle$

where *tf* is a function for retrieving a (non-empty) text from some secondary store and *init* initializes the file for text processing. The specification of the editor itself, however, is not influenced by this decision.

From the informal requirements we know that a "text is a sequence of lines". Obviously, a (proper) line is a (possibly empty) string composed of letters, digits, and blanks, that is furthermore terminated by a {cr}, i e.,

mode pline = (**input** l: $\exists$ **nbsstring** $n \cdot l = n + \{cr\}$),

and, hence, a text can be specified by

mode text = ESEQU(**pline**)

(where ε_t is used to denote the empty text).

Phrasings like "first line in the file", "next line", "current line", etc., suggest considering the "current text file" as a triple consisting of some piece of text (that might be empty), an actual line, followed by another piece of (possibly empty) text. However, there is also the case of the empty file, i.e , the file containing nothing at all In order not to be forced to always distinguish between empty and non-empty files, we introduce a "pseudo-line" (denoted by ε_l), extend the above definition of a line to be either a proper line (as defined above) or a pseudo-line, i.e.,

mode line = (**pline** | ε_l),

and define files uniformly by triples consisting of text, (extended) line, and text, i e. by

mode file = TRIPLE(**text, line, text**)

(where ε_f denotes the triple $\langle \varepsilon_t, \varepsilon_l, \varepsilon_t \rangle$ which models the empty file).

Furthermore we assume

mode action = (d(.), p( ), t(.), ..)

(i e a collection of output actions where d(.), p( ),... are abstract representations for the actions "display", "print", "type", etc , that are used in the informal description)

2.4.3. *Application of single commands*

Based on the above decomposition of **state**, we are now in a position to attempt a formal definition of the function *apply* used above.

Obviously (inherited by the above level of our specification) *apply* only has to deal with commands that are not an a or f command. Therefore it suffices to consider

funct *apply* = (**command'** *c*, **state** *s*) **state**,

where

mode command' = (**command** x: $x \notin \{\mathrm{a}, \mathrm{f}\}$),

since, according to the axioms (2b), (3b), and (3d), f or a commands may not occur as arguments of *apply*.

For defining the semantics of the individual commands we have to give appropriate axioms, at least one for each of them. From the aspect of presentation, it seems reasonable to point out similarities between different commands whenever there are any.

Thus, obviously all commands (except for the e, i, i', and the u command) display the {bel}, if the text file is empty, i.e.

$\forall$ **command'** *c*, **stack** *d*, **output** *o*: $c \notin \{\mathrm{e}, \mathrm{i}, \mathrm{i}', \mathrm{u}\}$:

$$apply(c, \langle \varepsilon_f, d, o\rangle) = \langle \varepsilon_f, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle.$$

This means that, in the sequel, for all commands (except for e, i, i', and u) a non-empty text file may be assumed. Therefore, for the remaining axioms we assume the general quantification

$\forall$ **text** t_1, t_2, t_3, **pline** l, l_1, **stack** *d*, **output** *o*: $t_3 \neq \varepsilon_t$:

which will be supplemented by additional restrictions when dealing with individual commands.

From the informal requirements the specification of the "regular" behaviour of the editor (i.e., in cases where t_3 denotes a non-empty text and l, l_1 denote proper lines) is straightforward.

(a) The b command:

$$\begin{aligned}&apply(\mathrm{b}, \langle\langle t_1, l, t_2\rangle, d, o\rangle)\\ &\quad = \langle\langle \varepsilon_t, \mathbf{first}(t_1 + l + t_2), \mathbf{rest}(t_1 + l + t_2)\rangle, d, o + \mathrm{p}(\mathbf{first}(t_1 + l + t_2)) + ?\rangle.\end{aligned}$$

The prerequisite for the definedness of the partial operations **first** and **rest** is fulfilled here, as $t_1 + l + t_2 \neq \varepsilon_t$ is guaranteed by the definition of **pline**.

(b) The e command:

$$apply(\mathrm{e}, \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\langle t_1 + l + t_2, \varepsilon_l, \varepsilon_t\rangle, d, o + \mathrm{d}(\{\mathrm{eof}\}) + ?\rangle,$$

$$apply(\mathrm{e}, \langle \varepsilon_f, d, o\rangle) = \langle \varepsilon_f, d, o + \mathrm{d}(\{\mathrm{eof}\}) + ?\rangle.$$

(c) The m command:

$\forall$ **nbsstring** x, y:

$$apply((\mathrm{m}, x, y), \langle\!\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\!\langle t_1, repl(l, x, y), t_2\rangle, d, o + re(l, x, y) + ?\rangle.$$

Here *repl* means the replacement of x by y in l and *re* means *repl* in case of a successful search and {bel} otherwise. Due to our strict top-down design philosophy *repl* and *re* can be defined on the next level of refinement, e.g., in connection with lines. These definitions then have to deal also with the question which occurrence is to be replaced, if there are several. Note also, that according to the definition of **nbsstring** both x and y may be empty.

(d) The i command:

$$apply((\mathrm{i}, l), \langle\varepsilon_\mathrm{t}, d, o\rangle) = \langle\!\langle\varepsilon_\mathrm{t}, l, \varepsilon_\mathrm{t}\rangle, d, o + \mathrm{d}(l)\rangle,$$

$$apply((\mathrm{i}, l), \langle\!\langle t_1, l_1, t_2\rangle, d, o\rangle) = \langle\!\langle t_1 + l_1, l, t_2\rangle, d, o + \mathrm{d}(l)\rangle,$$

$$apply((\mathrm{i}', l), \langle\varepsilon_\mathrm{t}, d, o\rangle) = \langle\!\langle\varepsilon_\mathrm{t}, l, \varepsilon_\mathrm{t}\rangle, d, o + \mathrm{d}(l) + ?\rangle,$$

$$apply((\mathrm{i}', l), \langle\!\langle t_1, l_1, t_2\rangle, d, o\rangle) = \langle\!\langle t_1 + l_1, l, t_2\rangle, d, o + \mathrm{d}(l) + ?\rangle,$$

For the i command we used the fact that the syntactic analysis converts a "complex" i command (i.e. insertion of several lines of text) into a sequence of insertions of lines. Only after the last line to be inserted (signalled by the pseudo-command i′) a prompt has to be output Note also, that by the definition of *parse* the argument l of an i command is always guaranteed to be a (proper) line

(e) The k command.

$$apply(\mathrm{k}, \langle\!\langle t_1, l, t_3\rangle, d, o\rangle) = \langle\!\langle t_1, \textbf{first } t_3, \textbf{rest } t_3\rangle, \textbf{push}(d, l), o + d(\textbf{first } t_3) + ?\rangle.$$

(f) The u command.

$$apply(\mathrm{u}, \langle\varepsilon_\mathrm{t}, \textbf{push}(d, l), o\rangle) = \langle\!\langle\varepsilon_\mathrm{t}, l, \varepsilon_\mathrm{t}\rangle, d, o + \mathrm{d}(l) + ?\rangle,$$

$$apply(\mathrm{u}, \langle\!\langle t_1, l, t_2\rangle, \textbf{push}(d, l_1), o\rangle) = \langle\!\langle t_1, l_1, l + t_2\rangle, d, o + \mathrm{d}(l_1) + ?\rangle.$$

(g) The move commands:

$\forall$ **nat** i:

$$apply((+, 0), \langle\!\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\!\langle t_1, l, t_2\rangle, d, o + \mathrm{t}(l) + ?\rangle,$$

$$apply((+, i+1), \langle\!\langle t_1, l, t_3\rangle, d, o\rangle)$$
$$= apply((+, i), \langle\!\langle t_1 + l, \textbf{first } t_3, \textbf{rest } t_3\rangle, d, o\rangle),$$

and similarly

$$apply((-, 0), \langle\!\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\!\langle t_1, l, t_2\rangle, d, o + \mathrm{t}(l) + ?\rangle,$$

$$apply((-, i+1), \langle\!\langle t_3, l, t_2\rangle, d, o\rangle)$$
$$= apply((-, i), \langle\!\langle\textbf{lead } t_3, \textbf{last } t_3, l + t_2\rangle, d, o\rangle).$$

(h) The s command:

$\forall$ **nbsstring** *key*:

$apply((\mathrm{s}, key), \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\langle t_1, l, t_2\rangle, d, o + \mathrm{d}(l) + ?\rangle$ **provided** *key* **isin** *l*,

$apply((\mathrm{s}, key), \langle\langle t_1, l, t_3\rangle, d, o\rangle)$
$= apply((\mathrm{s}, key), \langle\langle t_1 + l, \mathbf{first}\ t_3, \mathbf{rest}\ t_3\rangle, d, o\rangle)$**provided** $\neg$(*key* **isin** *l*).

Here **isin** denotes a predicate checking whether the string *key* occurs in *l*.

(i) The w command

$apply(\mathrm{w}, \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\langle t_1, l, t_2\rangle, d, o + \mathrm{d}(wind(t_1, l, t_2)) + ?\rangle.$

Again, *wind* denotes an auxiliary operation that generates a window of the desired size.

2.4.4. *Erroneous situations*

So far we have specified what has been required from the commands in the verbal specification. A simple formal examination, however, will again yield that the specification of *apply* is not complete, since borderline cases (that do not appear in the verbal specification) still have to be dealt with.

Of course, there are several possibilities to get rid of these marginal cases. One way is to use partial functions. However, in contrast to other applications, using partial functions does not make sense in our particular example, since it is hard to imagine what "undefined" should mean (should it mean that the screen bursts, or what else?). Another (simpler) way in our particular example would be to transmit the {bel} character as a kind of "universal" error message. This would be similar (and, hence, similarly unsatisfactory) to what in error handling in compilers is known as "*panic mode*". What is really tacitly expected (at least from the customer's side) is a "friendly" behaviour, i.e. whenever in such a borderline case there is a chance still to do something reasonable, the system should do so. It also seems reasonable to additionally output {bel} in order to signal a "warning" in these cases.

Aiming at such a friendly behaviour, we define:

ad (e)

$apply(\mathrm{k}, \langle\langle t_1, l, \varepsilon_t\rangle, d, o\rangle)$
$= \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, \mathbf{push}(d, l), o + \mathrm{d}(\{\mathrm{eof}\}) + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle.$

In the verbal specification it is required to display the line after the one deleted. However, if there is none, it cannot be displayed

ad (f)

$apply(\mathrm{u}, \langle\langle t_1, l, t_2\rangle, \varepsilon_d, o\rangle) = \langle\langle t_1, l, t_2\rangle, \varepsilon_d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle.$

Here we have assumed that the u command has no effect on the state, if the delete stack is empty—again a case that is not considered in the verbal requirements.

ad (g)

$$apply((+, i+1), \langle\langle t_1, l, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, l, \varepsilon_t\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle$$

(and, of course, analogously for $(-, i+1)$).

The informal requirements might suggest that + and − have converse effects, e g

$$(-, m) + (+, n) = \begin{cases} (-, (m-n)) & \text{if } m > n, \\ (+, (n-m)) & \text{if } n > m, \\ (+, 0) & \text{if } n = m \end{cases}$$

(and analogously for $(+, m) + (-, n)$).

If we had specified in this way, we not only have forgotten that each move command also generates output even if the file is not changed. We also would have prevented any "user-friendly" solution to the above-mentioned borderline case.

ad (h)

$\forall$ **nbsstring** *key*

$$apply((\mathrm{s}, key), \langle\langle t_1, l, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, l, \varepsilon_t\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle$$
provided $\neg(key$ **isin** $l)$

Still two more questions have to be tackled.

First, we have not yet specified what happens, if the current line of the file is the pseudo-line ε_1 As we assume that files only can be created using the editor operations, the case $\langle t_1, \varepsilon_1, t_2\rangle$ always implies $t_2 = \varepsilon_t$, since (for a non-empty file) $\langle t_1, \varepsilon_1, t_2\rangle$ can only result from applying an e or k command Hence, the case $\langle t_1, \varepsilon_1, t_2\rangle$ with $t_2 \neq \varepsilon_t$ needs not to be considered explicitly, and an appropriate invariant assertion can be added to *apply*

```
funct apply
   = (command' $c$, state $s$
      $\exists$ stack $d$, output $o$, text $t_1, t_2$  $s = \langle\langle t_1, \varepsilon_1, t_2\rangle, d, o\rangle \Rightarrow t_2 = \varepsilon_t$) state
```

For the current line being the pseudo-line we have

$\forall$ **command'** $c \notin \{\mathrm{i}, \mathrm{i}', \mathrm{u}, \mathrm{b}, \mathrm{e}, -\}$

$$apply(c, \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle$$

and

$$apply((\mathrm{i}, l), \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, l, \varepsilon_t\rangle, d, o + \mathrm{d}(l)\rangle,$$

$$apply((\mathrm{i}', l), \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, l, \varepsilon_t\rangle, d, o + \mathrm{d}(l) + ?\rangle,$$

$$apply(\mathrm{u}, \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, \mathbf{push}(d, l), o\rangle) = \langle\langle t_1, l, \varepsilon_t\rangle, d, o + \mathrm{d}(l) + ?\rangle,$$

$apply(\mathrm{b}, \langle\langle t_3, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle \varepsilon_t, \mathbf{first}\ t_3, \mathbf{rest}\ t_3\rangle, d, o + \mathrm{p}(\mathbf{first}\ t_3) + ?\rangle,$

$apply(\mathrm{e}, \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o + \mathrm{d}(\{\mathrm{eof}\}) + ?\rangle,$

$apply((-, 0), \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,$

$apply((-, \iota+1), \langle\langle t_3, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = apply((-, \iota), \langle\langle \mathbf{lead}\ t_3, \mathbf{last}\ t_3, \varepsilon_t\rangle, d, o\rangle).$

Secondly we have to deal with the fact that any syntactically incorrect input is mapped to an (abstract) error command ec. This error command simply can be specified by

$\forall$ **file** f:

$apply(\mathrm{ec}, \langle f, d, o\rangle) = \langle f, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle.$

2.4.5. Summary

As for the function *parse* we now summarize the axioms of *apply* and rearrange and rephrase them in order to ease the proofs of soundness and formal completeness. Thus we get altogether:

funct *apply*
  = (**command′** c, **state** s:
    $\exists$ **stack** d, **output** o, **text** t_1, t_2: $s = \langle\langle t_1, \varepsilon_1, t_2\rangle, d, o\rangle \Rightarrow t_2 = \varepsilon_t$) **state**,

$\forall$ **file** f, **stack** d, **output** o, **nat** ι, **text** t_1, t_2, **pline** l, l_1, **nbsstring** x, y:

$apply(\mathrm{b}, \langle \varepsilon_f, d, o\rangle) = \langle \varepsilon_f, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,$
$apply(\mathrm{b}, \langle\langle t_1, l, t_2\rangle, d, o\rangle)$
  $= \langle\langle \varepsilon_t, \mathbf{first}(t_1 + l + t_2), \mathbf{rest}(t_1 + l + t_2)\rangle, d, o + \mathrm{p}(\mathbf{first}(t_1 + l + t_2)) + ?\rangle,$
$apply(\mathrm{b}, \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle \varepsilon_t, \mathbf{first}\ t_1, \mathbf{rest}\ t_1\rangle, d, o + \mathrm{p}(\mathbf{first}\ t_1) + ?\rangle$
  provided $t_1 \neq \varepsilon_t$,

$apply(\mathrm{e}, \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\langle t_1 + l + t_2, \varepsilon_1, \varepsilon_t\rangle, d, o + \mathrm{d}(\{\mathrm{eof}\}) + ?\rangle,$
$apply(\mathrm{e}, \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o + \mathrm{d}(\{\mathrm{eof}\}) + ?\rangle,$

$apply((\mathrm{m}, x, y), \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\langle t_1, repl(l, x, y), t_2\rangle, d, o + re(l, x, y) + ?\rangle,$
$apply((\mathrm{m}, x, y), \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,$

$apply((\mathrm{i}, l), \langle\langle t_1, l_1, t_2\rangle, d, o\rangle) = \langle\langle t_1 + l_1, l, t_2\rangle, d, o + \mathrm{d}(l)\rangle,$
$apply((\mathrm{i}, l), \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, l, \varepsilon_t\rangle, d, o + \mathrm{d}(l)\rangle,$
$apply((\mathrm{i}', l), \langle\langle t_1, l_1, t_2\rangle, d, o\rangle) = \langle\langle t_1 + l_1, l, t_2\rangle, d, o + \mathrm{d}(l) + ?\rangle,$
$apply((\mathrm{i}', l), \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, l, \varepsilon_t\rangle, d, o + \mathrm{d}(l) + ?\rangle,$

$apply(\mathrm{k}, \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\langle t_1, \mathbf{first}\ t_2, \mathbf{rest}\ t_2\rangle, \mathbf{push}(d, l), o + d(\mathbf{first}\ t_2) + ?\rangle$
  provided $t_2 \neq \varepsilon_t$,
$apply(\mathrm{k}, \langle\langle t_1, l, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, \mathbf{push}(d, l), o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,$
$apply(\mathrm{k}, \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o\rangle) = \langle\langle t_1, \varepsilon_1, \varepsilon_t\rangle, d, o + \mathrm{d}(\{\mathrm{eof}\}) + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,$

$$\begin{aligned}
&\mathit{apply}(\mathrm{u}, \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\langle t_1, \mathbf{top}\ d, l + t_2\rangle, \mathbf{pop}\ d, o + \mathrm{d}(\mathbf{top}\ d) + ?\rangle\\
&\quad \mathbf{provided}\ d \neq \varepsilon_{\mathrm{d}},\\
&\mathit{apply}(\mathrm{u}, \langle f, \varepsilon_{\mathrm{d}}, o\rangle) = \langle f, \varepsilon_{\mathrm{d}}, \mathrm{o} + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,\\
&\mathit{apply}(\mathrm{u}, \langle\langle t_1, \varepsilon_{\mathrm{l}}, \varepsilon_{\mathrm{t}}\rangle, d, o\rangle) = \langle\langle t_1, \mathbf{top}\ d, \varepsilon_{\mathrm{t}}\rangle, \mathbf{pop}\ d, o + \mathrm{d}(\mathbf{top}\ d) + ?\rangle\\
&\quad \mathbf{provided}\ d \neq \varepsilon_{\mathrm{d}},
\end{aligned}$$

$$\begin{aligned}
&\mathit{apply}((+, 0), \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\langle t_1, l, t_2\rangle, d, o + t(l) + ?\rangle,\\
&\mathit{apply}((+, i), \langle\langle t_1, \varepsilon_{\mathrm{l}}, \varepsilon_{\mathrm{t}}\rangle, d, o\rangle) = \langle\langle t_1, \varepsilon_{\mathrm{l}}, \varepsilon_{\mathrm{t}}\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,\\
&\mathit{apply}((+, i+1), \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \mathit{apply}((+, i), \langle\langle t_1 + l, \mathbf{first}\ t_2, \mathbf{rest}\ t_2\rangle, d, o\rangle)\\
&\quad \mathbf{provided}\ t_2 \neq \varepsilon_{\mathrm{t}},\\
&\mathit{apply}((+, i+1), \langle\langle t_1, l, \varepsilon_{\mathrm{t}}\rangle, d, o\rangle) = \langle\langle t_1, l, \varepsilon_{\mathrm{t}}\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,
\end{aligned}$$

$$\begin{aligned}
&\mathit{apply}((-, 0), \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\langle t_1, l, t_2\rangle, d, o + \mathrm{t}(l) + ?\rangle,\\
&\mathit{apply}((-, 0), \langle\langle t_1, \varepsilon_{\mathrm{l}}, \varepsilon_{\mathrm{t}}\rangle, d, o\rangle) = \langle\langle t_1, \varepsilon_{\mathrm{l}}, \varepsilon_{\mathrm{t}}\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,\\
&\mathit{apply}((-, i+1), \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \mathit{apply}((-, i), \langle\langle \mathbf{lead}\ t_1, \mathbf{last}\ t_1, l + t_2\rangle, d, o\rangle)\\
&\quad \mathbf{provided}\ t_1 \neq \varepsilon_{\mathrm{t}},\\
&\mathit{apply}((-, i+1), \langle\langle \varepsilon_{\mathrm{t}}, l, t_2\rangle, d, o\rangle) = \langle\langle \varepsilon_{\mathrm{t}}, l, t_2\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,\\
&\mathit{apply}((-, i+1), \langle\langle t_1, \varepsilon_{\mathrm{l}}, \varepsilon_{\mathrm{t}}\rangle, d, o\rangle) = \mathit{apply}((-, i), \langle\langle \mathbf{lead}\ t_1, \mathbf{last}\ t_1, \varepsilon_{\mathrm{t}}\rangle, d, o\rangle)\\
&\quad \mathbf{provided}\ t_1 \neq \varepsilon_{\mathrm{t}},\\
&\mathit{apply}((-, i+1), \langle \varepsilon_{\mathrm{l}}, d, o\rangle) = \langle \varepsilon_{\mathrm{l}}, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,
\end{aligned}$$

$$\begin{aligned}
&\mathit{apply}((\mathrm{s}, x), \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\langle t_1, l, t_2\rangle, d, o + \mathrm{d}(l) + ?\rangle\ \mathbf{provided}\ x\ \mathbf{isin}\ l,\\
&\mathit{apply}((\mathrm{s}, x), \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \mathit{apply}((\mathrm{s}, x), \langle\langle t_1 + l, \mathbf{first}\ t_2, \mathbf{rest}\ t_2\rangle, d, o\rangle)\\
&\quad \mathbf{provided}\ t_2 \neq \varepsilon_{\mathrm{t}} \wedge \neg(x\ \mathbf{isin}\ l),\\
&\mathit{apply}((\mathrm{s}, x), \langle\langle t_1, l, \varepsilon_{\mathrm{t}}\rangle, d, o\rangle) = \langle\langle t_1, l, \varepsilon_{\mathrm{t}}\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle\\
&\quad \mathbf{provided}\ \neg(x\ \mathbf{isin}\ l),\\
&\mathit{apply}((\mathrm{s}, x), \langle\langle t_1, \varepsilon_{\mathrm{l}}, \varepsilon_{\mathrm{t}}\rangle, d, o\rangle) = \langle\langle t_1, \varepsilon_{\mathrm{l}}, \varepsilon_{\mathrm{t}}\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,
\end{aligned}$$

$$\begin{aligned}
&\mathit{apply}(\mathrm{w}, \langle\langle t_1, l, t_2\rangle, d, o\rangle) = \langle\langle t_1, l, t_2\rangle, d, o + \mathrm{d}(\mathit{wind}(t_1, l, t_2)) + ?\rangle\\
&\quad \mathbf{provided}\ l \neq \varepsilon_{\mathrm{l}},\\
&\mathit{apply}(\mathrm{w}, \langle\langle t_1, \varepsilon_{\mathrm{l}}, \varepsilon_{\mathrm{t}}\rangle, d, o\rangle) = \langle\langle t_1, \varepsilon_{\mathrm{l}}, \varepsilon_{\mathrm{t}}\rangle, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle,
\end{aligned}$$

$$\mathit{apply}(\mathrm{ec}, \langle f, d, o\rangle) = \langle f, d, o + \mathrm{d}(\{\mathrm{bel}\}) + ?\rangle.$$

3. Transformational development

The subsequent transformational development will deal with the functions *effect*, *apply*, and *parse* Its emphasis is on the transition from the respective algebraic specifications to equivalent (tail recursive) applicative programs working in an "on-line" fashion, i e , processing one unit after the other (without knowing the full history)

The overall strategy we are going to follow is fairly simple· first, we transform the algebraic specification into an applicative program, second, we apply suitable transformations to improve this program The main techniques to be used are rephrasing of axioms on the level of algebraic specification, and abstraction, embedding, case introduction, and unfold/fold for the operational improvement.

3.1. Development of an operational version of effect

The operation *effect*, describing the overall behaviour of the editor, was defined on sequences of commands rather than on individual commands to be processed one at a time. In this way we were able to specify the effects of the commands a and f, which depend on "previous" commands, in a very elegant way. In the sequel we are going to transform this abstract specification into an equivalent one where commands are individually processed.

3.1.1. Condensing the specification of effect

The formulation of the axioms for *effect* in Section 2.4.1 was primarily guided by the intention to ease a completeness proof. This led to a fairly large number of axioms. Therefore our first development step aims at reducing this number while simultaneously providing somewhat more uniformity for the right-hand sides of the axioms in order to keep the presentation reasonably short.

To this end we introduce

funct $apply' = ($**command** $c,$ **state** $s)$ **state**:

if $c \in \{\mathrm{a}, \mathrm{f}\}$ **then** s **else** $apply(c, s)$ **fi**

(which allows us to combine the axioms (2a) and (2b) as well as (3a) and (3b)) and furthermore combine (3e) and (3f) into one axiom. Thus we get a new (equivalent) specification for *effect*:

$\forall$ **command** c, c', **commsequ** cs, **state** s:

(1) $effect(\varepsilon, s) = s,$

(2ab) $effect(c + \varepsilon, s) = apply'(c, s),$

(3ab) $effect(c + c' + cs, s) = effect(c' + cs, apply'(c, s))$
provided $c \in \{\mathrm{a}, \mathrm{f}\} \vee c' \notin \{\mathrm{a}, \mathrm{f}\},$

(3c) $effect(c + c' + cs, s) = effect((\mathrm{s}, key) + cs, apply'(c', s))$
provided $c' = \mathrm{f} \wedge c = (\mathrm{m}, key, rep),$

(3d) $effect(c + c' + cs, s) = effect((\mathrm{m}, key, rep) + cs, apply'(c, s))$
provided $c' = (\mathrm{a}, rep) \wedge c = (\mathrm{s}, key),$

(3ef) $effect(c + c' + cs, s) = effect(c + cs, apply'(c', s))$
provided $c \notin \{\mathrm{a}, \mathrm{f}\} \wedge ((c' = \mathrm{f} \wedge c \neq \mathrm{m}) \vee (c' = \mathrm{a} \wedge c \neq \mathrm{s}))$

3 1 2 Introduction of delimiter symbols

We still have to differentiate between empty sequences, singleton sequences, and sequences with more than one element. Hence our next efforts aim at making this distinction vanish. Intuitively, we use a technique that is well-known in the areas of compiler construction or string processing. There, frequently, strings are supplemented by (unique) delimiter symbols such that recognizing an empty string is reduced to recognize the respective delimiter symbol, and therefore "normal" cases and "borderline" cases can be treated alike

Formally, we perform a simple data type transformation changing a command sequence into one which is delimited at both ends by the (new) delimiter symbols # (acting as dummy commands). To this end we introduce

- sequences consisting of commands and delimiter symbols, defined by

  mode commsequ' = ESEQU(**command**$^{\#}$)

  where

  mode command$^{\#}$ = **command** | {#},

- sequences of commands that have a delimiter symbol at their right end,

  mode commsequ$^{\#}$ = (**commsequ'** s': $\exists$ **commsequ** s: $s' = s + \#$),

  and

- sequences of commands that have delimiter symbols at both ends,

  mode commsequ$^{\#\#}$ = (**commsequ'** s' $\exists$ **commsequ** s $s' = \# + s + \#$).

Next we adjust *apply'* and *effect* to these new data types, i.e we transform them into functions *apply*$^{\#}$ and *effect*$^{\#}$ that work on delimited command sequences, but otherwise are the same (Note, that by definition, for all objects *cs* of types **commsequ**$^{\#}$ and **commsequ**$^{\#\#}$ respectively, $cs \neq \varepsilon$ holds)

We get

funct *apply*$^{\#}$ = (**command**$^{\#}$ c, **state** s) **state**.

  if $c \in \{\mathrm{a}, \mathrm{f}, \#\}$ **then** s **else** *apply*(c, s) **fi**,

and

funct (**commsequ**$^{\#\#}$, **state**) **state** *effect*$^{\#}$,

defined by

$\forall$ **command**$^{\#}$ c, **command** c', **commsequ**$^{\#}$ cs, **state** s.

(1/2) $effect^{\#}(c + \#, s) = apply^{\#}(c, s)$,

(3ab) $effect^{\#}(c + c' + cs, s) = effect^{\#}(c' + cs, apply^{\#}(c, s))$
  provided $c \in \{\mathrm{a}, \mathrm{f}, \#\} \vee c' \notin \{\mathrm{a}, \mathrm{f}\}$,

(3c) $effect^{\#}(c + c' + cs, s) = effect^{\#}((\mathrm{s}, key) + cs, apply^{\#}(c', s))$
  provided $c' = \mathrm{f} \wedge c = (\mathrm{m}, key, rep)$,

(3d) $effect^{\#}(c + c' + cs, s) = effect^{\#}((\mathrm{m}, key, rep) + cs, apply^{\#}(c, s))$
  provided $c' = (\mathrm{a}, rep) \wedge c = (\mathrm{s}, key)$,

(3ef) $effect^{\#}(c+c'+cs, s) = effect^{\#}(c+cs, apply^{\#}(c', s))$
provided $c \notin \{\mathrm{a}, \mathrm{f}, \#\} \wedge ((c' = \mathrm{f} \wedge c \neq \mathrm{m}) \vee (c' = \mathrm{a} \wedge c \neq \mathrm{s}))$.

The right-hand sides of (3c) and (3ef) can be further simplified using the respective premises and the definition of *apply*$^{\#}$. We get

(3c) $\ldots = effect^{\#}((\mathrm{s}, key) + cs, s)$ **provided** ...,

(3ef) $\ldots = effect^{\#}(c+cs, s)$ **provided** ....

Of course, the definition of *edit* has also to be adapted to these new data types

$$edit(in) = unparse(effect^{\#}(\# + parse(in) + \#, initstate))$$

such that the previous transformation step technically amounts to an embedding (cf. [12]).

3.1.3. *Shifting the focus of the computation*

Due to the fact that commands may have effects on other commands the definition of *effect* has to take into account always two commands at a time. In the previous definition of *effect* this fact was reflected by looking one command ahead. Of course, the same situation also can be handled by remembering the command that has been considered before

In order to achieve this shift of the focus of the computation we add two arguments to *effect* which are to remember the previous command and the previous state. Technically, this is achieved by another embedding. We introduce

funct (**commsequ**$^{\#}$ cs, **state** s', **command**$^{\#}$ c, **state** s:
$s' = apply^{\#}(c, s)$) **state** *eff*,

defined by

$$eff(cs, apply^{\#}(c, s), c, s) = effect^{\#}(c + cs, s),$$

and transform the definition of *edit* into

$$edit(in) = unparse(eff(parse(in) + \#, initstate, \#, initstate)).$$

Obviously, this definition is equivalent to the previous one, as (according to the definition of *apply*$^{\#}$)

$$apply^{\#}(\#, initstate) = initstate.$$

The goal now is to derive a definition of *eff* which is independent of *effect*$^{\#}$. Using the above definition of *eff*, the axioms for *effect*$^{\#}$ directly translate to

$\forall$ **command**$^{\#}$ c, **command** c', **commsequ**$^{\#}$ cs, **state** s, s': $s' = apply^{\#}(c, s)$.

(1/2) $eff(\#, s', c, s) = s'$,
(3ab) $eff(c' + cs, s', c, s) = eff(cs, apply^{\#}(c', s'), c', s')$
provided $c \in \{\mathrm{a}, \mathrm{f}, \#\} \vee c' \notin \{\mathrm{a}, \mathrm{f}\}$,

(3c) $\mathit{eff}(c'+cs, s', c, s) = \mathit{eff}(cs, \mathit{apply}^{\#}((\mathrm{s}, \mathit{key}), s), (\mathrm{s}, \mathit{key}), s)$
provided $c' = \mathrm{f} \wedge c = (\mathrm{m}, \mathit{key}, \mathit{rep})$,

(3d) $\mathit{eff}(c'+cs, s', c, s) = \mathit{eff}(cs, \mathit{apply}^{\#}((\mathrm{m}, \mathit{key}, \mathit{rep}), s'), (\mathrm{m}, \mathit{key}, \mathit{rep}), s')$
provided $c' = (\mathrm{a}, \mathit{rep}) \wedge c = (\mathrm{s}, \mathit{key})$,

(3ef) $\mathit{eff}(c'+cs, s', c, s) = \mathit{eff}(cs, s', c, s)$
provided $c \notin \{\mathrm{a}, \mathrm{f}, \#\} \wedge ((c' = f \wedge c \neq \mathrm{m}) \vee (c' = \mathrm{a} \wedge c \neq \mathrm{s}))$.

Finally, *apply*$^{\#}$ can be eliminated·
By simple unfolding the assertion translates to

$$s' = \textbf{if } c \in \{\mathrm{a}, \mathrm{f}, \#\} \textbf{ then } s \textbf{ else } \mathit{apply}(c, s) \textbf{ fi}$$

which can be further simplified to

$$(c \in \{\mathrm{a}, \mathrm{f}, \#\} \mathbin{\Delta} s' = s) \mathbin{\nabla} s' = \mathit{apply}(c, s).$$

Unfolding *apply*$^{\#}$ in the axioms is trivial for (3c) and (3d) and leads to a case distinction for 3(ab).
Altogether we get

$\forall$ **command**$^{\#}$ c, **command** c', **commsequ**$^{\#}$ cs, **state** s, s':
$(c \in \{\mathrm{a}, \mathrm{f}, \#\} \mathbin{\Delta} s' = s) \mathbin{\nabla} s' = \mathit{apply}(c, s)$.

(1/2) $\mathit{eff}(\#, s', c, s) = s'$,

(3a) $\mathit{eff}(c'+cs, s', c, s) = \mathit{eff}(cs, s', c', s')$ **provided** $c \in \{\mathrm{a}, \mathrm{f}, \#\} \wedge c' \in \{\mathrm{a}, \mathrm{f}\}$,

(3b) $\mathit{eff}(c'+cs, s', c, s) = \mathit{eff}(cs, \mathit{apply}(c', s'), c', s')$ **provided** $c' \notin \{\mathrm{a}, \mathrm{f}\}$,

(3c) $\mathit{eff}(c'+cs, s', c, s) = \mathit{eff}(cs, \mathit{apply}((\mathrm{s}, \mathit{key}), s), (\mathrm{s}, \mathit{key}), s)$
provided $c' = \mathrm{f} \wedge c = (\mathrm{m}, \mathit{key}, \mathit{ren})$,

(3d) $\mathit{eff}(c'+cs, s', c, s) = \mathit{eff}(cs, \mathit{apply}((\mathrm{m}, \mathit{key}, \mathit{rep}), s'), (\mathrm{m}, \mathit{key}, \mathit{rep}), s')$
provided $c' = (\mathrm{a}, \mathit{rep}) \wedge c = (\mathrm{s}, \mathit{key})$,

(3ef) $\mathit{eff}(c'+cs, s', c, s) = \mathit{eff}(cs, s', c, s)$
provided $c \notin \{\mathrm{a}, \mathrm{f}, \#\} \wedge ((c' = \mathrm{f} \wedge c \neq \mathrm{m}) \vee (c' = \mathrm{a} \wedge c \neq \mathrm{s}))$.

Note that this version also could have been derived without introducing the auxiliary operations *apply'* and *apply*$^{\#}$ respectively, however, at the expense of considerably more effort in keeping the number of axioms as small as possible

3.1 4 An equivalent applicative program

According to the rules given in [3], our last version of *eff* can immediately be converted into the definition of an applicative function

```
funct eff' = (commsequ# cs, state s', command# c, state s:
              (c ∈ {a, f, #} Δ s' = s) ∇ s' = apply(c, s)) state.
  begin command# c' = first cs,
  if c' = # then s'
    else commsequ# cs' = rest cs,
      if c ∈ {a, f, #} ∧ c' ∈ {a, f} then eff'(cs', s', c', s')
      □ c' ∉ {a, f} then eff'(cs', apply(c', s'), c', s')
□ $c' = \mathrm{f} \wedge c = (\mathrm{m}, key, rep)$ then $eff'(cs', apply((\mathrm{s}, key), s), (\mathrm{s}, key), s)$
□ $c' = (\mathrm{a}, rep) \wedge c = (\mathrm{s}, key)$
   then $eff'(cs', apply(\mathrm{m}, key, rep), s'), (\mathrm{m}, key, rep), s')$
□ $c \notin \{\mathrm{a}, \mathrm{f}, \#\} \wedge ((c' = \mathrm{f} \wedge c \neq \mathrm{m}) \vee (c' = \mathrm{a} \wedge c \neq \mathrm{s}))$
   then $eff'(cs', s', c, s)$   fi fi end
```

where, by definition, the last branch of the guarded expression simply can be abbreviated to **else**.

3.2. *Introduction of on-line processing*

The previous definition of *eff'* still assumes that the complete input sequence is already parsed into a sequence of commands. Our next intermediate goal is the introduction of "on-line" behaviour, i.e. the parsing and processing of single commands one after the other rather than considering (pre-parsed) command sequences as done so far.

3.2.1. *Combining parsing and processing of single commands*

Technically, the introduction of "on-line" behaviour is achieved by simple function composition, applied to *eff'* and *parse* in the right-hand side of the definition of *edit*, viz.

$$edit(in) = unparse(eff'(parse(in) + \#, initstate, \#, initstate)).$$

This function composition simply can be done by using the unfold/fold strategy. First, we redefine *edit* into

$$edit(in) = unparse(eff''(in, initstate, \#, initstate)),$$

where

```
funct $eff'' = $ (input $in$, state $s'$, command$^{\#}$ $c$, state $s$:
            $(c \in \{\mathrm{a}, \mathrm{f}, \#\} \;\Delta\; s' = s) \;\nabla\; s' = apply(c, s)$) state:
  $eff'(parse(in) + \#, s', c, s)$.
```

By unfolding (the simplified version of) *eff'* we get

```
funct $eff'' = $ (input $in$, state $s'$, command$^{\#}$ $c$, state $s$:
            $(c \in \{\mathrm{a}, \mathrm{f}, \#\} \;\Delta\; s' = s) \;\nabla\; s' = apply(c, s)$) state.
  begin command$^{\#}$ $c' = $ first$(parse(in) + \#)$;
  if $c' = \#$ then $s'$
    else commsequ$^{\#}$ $cs' = $ rest$(parse(in) + \#)$;
      if $c \in \{\mathrm{a}, \mathrm{f}, \#\} \wedge c' \in \{\mathrm{a}, \mathrm{f}\}$ then $eff'(cs', s', c', s')$
      □ $c' \notin \{\mathrm{a}, \mathrm{f}\}$ then $eff'(cs', apply(c', s'), c', s')$
      □ $c' = \mathrm{f} \wedge c = (\mathrm{m}, key, rep)$ then $eff'(cs', apply((\mathrm{s}, key), s), (\mathrm{s}, key), s)$
      □ $c' = (\mathrm{a}, rep) \wedge c = (\mathrm{s}, key)$
        then $eff'(cs', apply((\mathrm{m}, key, rep), s'), (\mathrm{m}, key, rep), s')$
        else $eff'(cs', s', c, s)$   fi fi end
```

This last version still contains sequences of commands that have to be eliminated. We do this in two steps. First, by abstraction, we introduce auxiliary operations

```
funct $nextcommand$ = (input $in$) command$^\#$.
  first($parse(in) + \#$)
```

and

```
funct $inputrest$ = (input $in$  $parse(in) \neq \varepsilon$) input:
  some input $z$  $parse(z) + \# = $ rest($parse(in) + \#$).
```

Obviously, $parse(inputrest(in)) + \# =$ **rest**$(parse(in) + \#)$, and, hence, our next intermediate version reads

```
funct $eff''$ = (input $in$, state $s'$, command$^\#$ $c$, state $s$
            $(c \in \{a, f, \#\} \Delta s' = s) \nabla s' = apply(c, s))$ state
  begin command$^\#$ $c' = nextcommand(in)$,
  if $c' = \#$ then $s'$
    else input $in' = inputrest(in)$; commsequ$^\#$ $cs' = parse(in') + \#$,
      if $c \in \{a, f, \#\} \wedge c' \in \{a, f\}$ then $eff'(cs', s', c', s')$
      $\square$ $c' \notin \{a, f\}$ then $eff'(cs', apply(c', s'), c', s')$
      $\square$ $c' = f \wedge c = (m, key, rep)$ then $eff'(cs', apply((s, key), s), (s, key), s)$
      $\square$ $c' = (a, rep) \wedge c = (s, key)$
        then $eff'(cs', apply((m, key, rep), s'), (m, key, rep), s')$
        else $eff'(cs', s', c, s)$   fi fi end
```

Next, by simply unfolding the declaration of cs', we get

```
funct $eff''$ = (input $in$, state $s'$, command$^\#$ $c$, state $s$:
            $(c \in \{a, f, \#\} \Delta s' = s) \nabla s' = apply(c, s))$ state.
  begin command$^\#$ $c' = nextcommand(in)$,
  if $c' = \#$ then $s'$
    else input $in' = inputrest(in)$,
      if $c \in \{a, f, \#\} \wedge c' \in \{a, f\}$ then $eff'(parse(in') + \#, s', c', s')$
        $\square$ $c' \notin \{a, f\}$ then $eff'(parse(in') + \#, apply(c', s'), c', s')$
        $\square$ $c' = f \wedge c = (m, key, rep)$
          then $eff'(parse(in') + \#, apply((s, key), s), (s, key), s)$
        $\square$ $c' = (a, rep) \wedge c = (s, key)$
          then $eff'(parse(in') + \#, apply((m, key, rep), s'),$
                      $(m, key, rep), s')$
          else $eff'(parse(in') + \#, s', c, s)$   fi fi end
```

Now folding (with assertion) of eff'' is possible and our final result is

```
funct $eff''$ = (input $in$, state $s'$, command$^\#$ $c$, state $s$
            $(c \in \{a, f, \#\} \Delta s' = s) \nabla s' = apply(c, s))$ state
  begin command$^\#$ $c' = nextcommand(in)$,
if $c' = \#$ then $s'$
  else input $in' = inputrest(in)$;
    if $c \in \{a, f, \#\} \wedge c' \in \{a, f\}$ then $eff''(in', s', c, s')$
    □ $c' \notin \{a, f\}$ then $eff''(in', apply(c', s'), c', s')$
    □ $c' = f \wedge c = (m, key, rep)$
      then $eff''(in', apply((s, key), s), (s, key), s)$
    □ $c' = (a, rep) \wedge c = (s, key)$
      then $eff''(in', apply(m, key, rep), s'), (m, key, rep), s')$
      else $eff''(in', s', c, s)$   fi fi end
```

where commands are recognized (by *nextcommand*) and processed one at a time.

3.2.2. *A more detailed version*

Obviously, within the given context,

$$\begin{aligned} c' \notin \{a, f\} \Leftrightarrow\ & c' = ec \vee c' = b \vee c' = e \vee c' = (m, key, rep) \vee \\ & c' = (i, l) \vee c' = (i', l) \vee \\ & c' = k \vee c' = u \vee c' = (+, n) \vee c' = (-, n) \vee c' = s(key) \vee c' = w \end{aligned}$$

Hence the respective branch in the above program can be further detailed. Additionally, we combine the auxiliary operations *nextcommand* and *inputrest* into a single new auxiliary operation

```
funct $nextcomm$ = (input $in$) (command$^\#$, input):
  ($nextcommand(in)$, if $parse(in) = \varepsilon$ then $\varepsilon$ else $inputrest(in)$ fi)
```

This leads to

```
funct $eff''$ = (input $in$, state $s'$, command$^\#$ $c$, state $s$:
            $(c \in \{a, f, \#\}\ \Delta\ s' = s)\ \nabla\ s' = apply(c, s)$) state.
  begin (command$^\#$ $c'$, input $in'$) = $nextcomm(in)$,
  if $c' = \#$ then $s'$
    else if $c \in \{a, f, \#\} \wedge c' \in \{a, f\}$ then $eff''(in', s', c', s')$
      □ $c' = ec$ then $eff''(cs', apply(ec, s'), ec, s')$
      □ $c' = b$ then $eff''(cs', apply(b, s'), b, s')$
      □ $c' = e$ then $eff''(cs', apply(e, s'), e, s')$
      □ $c' = (m, key, rep)$
        then $eff''(cs', apply((m, key, rep), s'), (m, key, rep), s')$
      □ $c' = (i, l)$ then $eff''(cs', apply((i, l), s'), (i, l), s')$
      □ $c' = (i', l)$ then $eff''(cs', apply((i', l), s'), (i', l), s')$
      □ $c' = k$ then $eff''(cs', apply(k, s'), k, s')$
      □ $c' = u$ then $eff''(cs', apply(u, s'), u, s')$
      □ $c' = (+, n)$ then $eff''(cs', apply((+, n), s'), (+, n), s')$
      □ $c' = (-, n)$ then $eff''(cs', apply((-, n), s'), (-, n), s')$
      □ $c' = (s, key)$ then $eff''(cs', apply((s, key), s'), (s, key), s')$
    □ $c' = \mathrm{w}$ **then** $\mathit{eff}''(cs', \mathit{apply}(\mathrm{w}, s'), \mathrm{w}, s')$
    □ $c' = \mathrm{f} \wedge c = (\mathrm{m}, \mathit{key}, \mathit{rep})$
      **then** $\mathit{eff}''(in', \mathit{apply}((\mathrm{s}, \mathit{key}), s), (\mathrm{s}, \mathit{key}), s)$
    □ $c' = (\mathrm{a}, \mathit{rep}) \wedge c = (\mathrm{s}, \mathit{key})$
      **then** $\mathit{eff}''(in', \mathit{apply}((\mathrm{m}, \mathit{key}, \mathit{rep}), s), (\mathrm{m}, \mathit{key}, \mathit{rep}), s')$
      **else** $\mathit{eff}''(in', s', c, s)$   **fi fi end**
```

3.2.3. An explicit, axiomatic definition of nextcomm

So far, the auxiliary operation *nextcomm* is defined using the operation *parse.* Our next intermediate goal is to derive a definition of *nextcomm* which is independent of *parse.*

First, by unfolding the respective definitions of *nextcommand* and *inputrest*, we get

```
funct nextcomm = (input in)(command#, input):
  (first(parse(in) + #),
  if parse(in) = ε then ε else some input z:
    parse(z) + # = rest(parse(in) + #) fi).
```

An axiomatic definition of *nextcomm* can be obtained from the axioms of *parse* in a straightforward way (again by using unfold/fold steps and the definition of *nextcomm*):

∀ input *in*, *sc*, **char** *c*, **digit** *d*, **string** t_1, t_2, **dstring** t_d:

$\mathit{nextcomm}(\varepsilon) = (\#, \varepsilon)$,

$\mathit{nextcomm}\ (c + in) = (c, in)$ **provided** $c \in \{\mathrm{b, e, k, u, f, w}\}$,

$\mathit{nextcomm}(c + in) = (\mathrm{ec}, in)$
  provided $c \notin (\{\mathrm{b, e, m, i, i', k, u, s, a, f, w}\} \cup \textbf{digit})$,

$\mathit{nextcomm}(\mathrm{m} + in) = ((\mathrm{m}, \mathit{mbl}(t_1), \mathit{dlb}(\mathit{mbl}(t_2))), sc)$
  provided $in = t_1 + \{\mathrm{cr}\} + t_2 + \{\mathrm{cr}\} + sc \wedge \mathit{nbs}(t_1)$,
$\mathit{nextcomm}(\mathrm{m} + in) = \mathit{nextcomm}(\mathbf{rest}(\mathit{mbl}(t_1)) + sc)$
  provided $in = t_1 + sc \wedge \neg \mathit{nbs}(t_1)$,
$\mathit{nextcomm}(\mathrm{m} + in) = (\#, \varepsilon)$
  provided $(in \in \textbf{string}\ \Delta\ \mathit{nbs}(in)) \vee (in = t_1 + \{\mathrm{cr}\} + t_2 \wedge \mathit{nbs}(t_1))$,
$\mathit{nextcomm}(\mathrm{m} + in) = (\mathrm{ec}, sc)$
  provided $(in = t_1 + \{\mathrm{cr}\} + t_2 + \{\mathrm{esc}\} + sc \wedge \mathit{nbs}(t_1))$
      $\vee (in = t_1 + \{\mathrm{esc}\} + sc \wedge \mathit{nbs}(t_1))$,

$\mathit{nextcomm}(\mathrm{i} + in) = ((\mathrm{i}, \mathit{dlb}(\mathit{mbl}(t_1)) + \{\mathrm{cr}\}), \mathrm{i} + sc)$
  provided $in = t_1 + \{\mathrm{cr}\} + sc$,
$\mathit{nextcomm}(\mathrm{i} + in) = ((\mathrm{i'}, \mathit{dlb}(\mathit{mbl}(t_1)) + \{\mathrm{cr}\}), sc)$
  provided $in = t_1 + \{\mathrm{esc}\} + sc$,
$\mathit{nextcomm}(\mathrm{i} + in) = (\#, \varepsilon)$ **provided** $in \in$ **string**,

$\mathit{nextcomm}(\mathrm{s} + in) = ((\mathrm{s}, \mathit{mbl}(t_1)), sc)$ **provided** $in = t_1 + \{\mathrm{cr}\} + sc \wedge \mathit{nbs}(t_1)$,

$nextcomm(\mathrm{s}+in) = nextcomm(\mathbf{rest}(mbl(t_1))+sc)$
  provided $in = t_1+sc \wedge \neg nbs(t_1)$,
$nextcomm(\mathrm{s}+in) = (\#, \varepsilon)$ **provided** $in \in$ **string** $\Delta\ nbs(in)$,
$nextcomm(\mathrm{s}+in) = (\mathrm{ec}, sc)$ **provided** $in = t_1+\{\mathrm{esc}\}+sc \wedge nbs(t_1)$,

$nextcomm(\mathrm{a}+in) = ((\mathrm{a}, mbl(t_1)), sc)$ **provided** $in = t_1+\{\mathrm{cr}\}+sc \wedge nbs(t_1)$,
$nextcomm(\mathrm{a}+in) = nextcomm(\mathbf{rest}(mbl(t_1))+sc)$
  provided $in = t_1+sc \wedge \neg nbs(t_1)$,
$nextcomm(\mathrm{a}+in) = (\#, \varepsilon)$ **provided** $in \in$ **string** $\Delta\ nbs(in)$,
$nextcomm(\mathrm{a}+in) = (\mathrm{ec}, sc)$ **provided** $in = t_1+\{\mathrm{esc}\}+sc \wedge nbs(t_1)$,

$nextcomm(d+in) = ((sig, conv(mbl(t_d), 0)), sc)$
  provided $in = t_d+sig+sc \wedge (mbl(d_d) \neq \varepsilon\ \Delta\ \mathbf{first}(mbl(t_d)) \neq \{\mathrm{bs}\})$
    $\wedge\ sig \in \{+, -\}$,
$nextcomm(d+in) = nextcomm(\mathbf{rest}(mbl(t_1))+sc)$
  provided $in = t_1+sc \wedge \neg nbs(t_1)$,
$nextcomm(d+in) = (\#, \varepsilon)$ **provided** $in \in$ **dstring** $\Delta\ mbl(d+in) \neq \varepsilon$,
$nextcomm(d+in) = (\mathrm{ec}, sc)$
  provided $d+in = t_d+c+sc \wedge (mbl(t_d) \neq \varepsilon\ \Delta\ \mathbf{first}(mbl(t_d)) \neq \{\mathrm{bs}\})$
    $\wedge\ c \notin \{+, -\} \cup$ **digit**.

3.2.4. *An operational definition for nextcomm*

It remains to derive operational versions for *nextcomm* starting with the axiomatic definition given above As a representative we will deal with the s command in detail.

First, we redefine (by abstraction) the respective axioms, viz.

(1) $nextcomm(\mathrm{s}+in) = ((\mathrm{s}, mbl(t_1)), sc)$ **provided** $in = t_1+\{\mathrm{cr}\}+sc \wedge nbs(t_1)$,
(2) $nextcomm(\mathrm{s}+in) = nextcomm(\mathbf{rest}(mbl(t_1))+sc)$
  provided $in = t_1+sc \wedge \neg nbs(t_1)$,
(3) $nextcomm(\mathrm{s}+in) = (\#, \varepsilon)$ **provided** $in \in$ **string** $\Delta\ nbs(in)$,
(4) $nextcomm(\mathrm{s}+in) = (\mathrm{ec}, sc)$ **provided** $in = t_1+\{\mathrm{esc}\}+sc \wedge nbs(t_1)$,

into

```
nextcomm(s + in) =
  if $in = \varepsilon$ then $(\#, \varepsilon)$
  else if first $in = \{\mathrm{bs}\}$ then nextcomm(rest in)
    else ps(first in, first in, rest in) fi fi
```

where the auxiliary operation *ps* parses an s command and is defined by

```
funct ps = (nstring p, string o, input in:
            $p = mbl(o) \wedge (o \neq \varepsilon\ \Delta$ first $o \neq \{\mathrm{bs}\})$) (command$^{\#}$, input):
  nextcomm(s + o + in).
```

Next, we derive axioms for *ps* using its definition, the assertion on the parameters, unfold and folding with assertion. As we specified *nextcomm* in such a way that all axioms are disjoint, no special care with respect to "overlapping" cases has to be taken. We get

> $\forall$ **nstring** p, **string** o, **input** in, **char** c, c': $c' \notin \{\mathrm{bs}, \mathrm{cr}, \mathrm{esc}\} \wedge c \notin \{\mathrm{bs}, \mathrm{cr}, \mathrm{esc}\}$:
> $ps(p, o, \varepsilon) = nextcomm(\mathrm{s} + o + \varepsilon) = (\#, \varepsilon)$ [according to (3)]
> $ps(p + c, o, \{\mathrm{bs}\} + in) = nextcomm(\mathrm{s} + o + \{\mathrm{bs}\} + in) = ps(p, o + \{\mathrm{bs}\}, in)$
> [by folding, as $p + c = mbl(o) \wedge (o \neq \varepsilon \,\Delta\, \textbf{first}\; o \neq \{\mathrm{bs}\}) \Rightarrow$
> $p = mbl(o + \{\mathrm{bs}\}) \wedge (o + \{\mathrm{bs}\} \neq \varepsilon \,\Delta\, \textbf{first}\,(o + \{\mathrm{bs}\}) \neq \{\mathrm{bs}\})$]
> $ps(\varepsilon, o, \{\mathrm{bs}\} + in) = nextcomm(\mathrm{s} + o + \{\mathrm{bs}\} + in) = nextcomm(in)$
> [according to (2), as $\varepsilon = mbl(o) \Rightarrow mbl(o + \{\mathrm{bs}\}) = \{\mathrm{bs}\}$
> $\Rightarrow \neg nbs(o + \{\mathrm{bs}\})$]
> $ps(p, o, \{\mathrm{cr}\} + in) = nextcomm(\mathrm{s} + o + \{\mathrm{cr}\} + in) = ((\mathrm{s}, p), in)$
> [according to (1), as $p \in$ **nstring** $\wedge\, p = mbl(o) \wedge nbs(o)$]
> $ps(p, o, \{\mathrm{esc}\} + in) = nextcomm(\mathrm{s} + o + \{\mathrm{esc}\} + in) = (\mathrm{ec}, in)$
> [according to (4), as $p \in$ **nstring** $\wedge\, p = mbl(o) \wedge nbs(o)$]
> $ps(p, o, c' + in) = nextcomm(\mathrm{s} + o + c' + in) = ps(p + c', o + c', in)$
> [by folding, as $c' \notin \{\mathrm{bs}, \mathrm{cr}, \mathrm{esc}\} \wedge p = mbl(o) \wedge (o \neq \varepsilon \,\Delta\, \textbf{first}\; o \neq \{\mathrm{bs}\}) \Rightarrow$
> $p + c' = mbl(o + c') \wedge (o + c' \neq \varepsilon \,\Delta\, \textbf{first}\,(o + \{\mathrm{bs}\}) \neq \{\mathrm{bs}\})$]

This definition of *ps* can be transformed into a recursive function in a straightforward way·

```
funct ps = (pstring p, string o, input in:
                $p = mbl(o) \wedge (o \neq \varepsilon \,\Delta\, \textbf{first}\; o \neq \{bs\})$) (command#, input):
    if $in = \varepsilon$ then $(\#, \varepsilon)$
       else if first $in = \{bs\}$
                then if $p = \varepsilon$ then nextcomm(rest in)
                                 else ps(p, $o + \{bs\}$, rest in) fi
             □ first $in = \{cr\}$ then ((s, p), rest in)
             □ first $in = \{esc\}$ then (ec, rest in)
                                  else ps($p +$ first i, $o +$ first i, rest in) fi fi.
```

In an analogous way also the remaining axioms of *nextcomm* for a, s, m, i, or move commands can be treated to finally obtain a fully operational version of *nextcomm.*

The result obtained so far does not yet exhibit true "on-line" behaviour, since *unparse* (in the definition of *edit*) is applied just to the "final" state rather than successively to all intermediate states thus producing output in an "on-line" way

Obviously, another function composition (here of *unparse* and *eff''*) is necessary which, although straightforward analogous to the treatment of *parse,* cannot be demonstrated in detail due to the lacking specification of *unparse*

3.3 Remarks on the further development

We have stopped our derivation at the level of applicative programs, since we are convinced that the still missing steps towards a conventional programming language such as, e.g., Pascal are fairly obvious. Nevertheless we would like to add a few more comments on these final optimization steps.

First, the definition of *apply* has to be transformed such that for each command there is exactly one axiom. Starting with the definition of *apply* from above this is straightforward and left as an exercise to the reader.

Next, we can get rid of the auxiliary structures **state** and **file** by simply unfolding the respective definitions. Of course this also requires unfolding of all calls of *apply* in *eff″* (where the recursively defined axioms have to be made into appropriate auxiliary operations). This also leads to a redefinition of *edit* into

$$edit(in) = unparse(eff^*(in, \varepsilon_t, \varepsilon_l, \varepsilon_t, \varepsilon_d, \varepsilon_o, \varepsilon^{\#}, \varepsilon_t, \varepsilon_l, \varepsilon_t, \varepsilon_d, \varepsilon_o))$$

where the explicit definition of

funct *eff** = (**input** *in*, **text** t_1, **line** *l*, **text** t_2, **stack** *d*,
output *o*, **command**$^{\#}$ *c*, **text** ot_1,
line *ol*, **text** ot_2, **stack** *od*, **output** *oo*)
(**text, line, text, stack, output**),

which is characterized by

$$\begin{aligned} &eff^*(in, t_1, l, t_2, d, o, c, ot_1, ol, ot_2, od, oo) \\ &\quad = eff''(in, \langle\!\langle t_1, l, t_2\rangle, d, o\rangle, c, \langle\!\langle ot_1, ol, ot_2\rangle, od, oo\rangle), \end{aligned}$$

again is straightforward.

The definition of *eff** also allows a possible further optimization according to the general idea of saving storage at the expense of additional computational effort Rather than keeping the "old state" $\langle\!\langle ot_1, ol, ot_2\rangle, od, oo\rangle$ explicitly, it is maybe more economic to just have the "current state" $\langle\!\langle t_1, l, t_2\rangle, d, o\rangle$, some information *inf*, and an operation *restore*, such that

$$restore(\langle\!\langle t_1, l, t_2\rangle, d, o\rangle, inf) = \langle\!\langle ot_1, ol, ot_2\rangle, od, oo\rangle,$$

i.e. to recompute the information contained in $\langle\!\langle ot_1, ol, ot_2\rangle, od, oo\rangle$ in case it is needed. Whether this last transformation, which in some sense is a counterpart to the technique of "finite differencing" (cf. [12]), is really an improvement, depends on further facts that go beyond the scope of this paper.

The function *eff** and its further optimized versions are all tail-recursive such that the well-known transformation from tail recursion to iteration (including final polish-up transformations for imperative programs) could be added as a final optimization step.

4. Conclusion

By means of a realistic case study we have demonstrated how the paradigms of algebraic specification and transformational programming can be used to bridge the gap between informally stated requirements and a running program. In particular, we have demonstrated how an abstract view taken in an algebraic specification is to be transformed into a practically reasonable operational version.

A reader unexperienced in using formal techniques in program development might be intimidated by the amount of formulas that were needed for a detailed formal problem description and discouraged or even bored by persistence necessary to follow its development. This very reader should be simply reminded of the fact that writing a program (without formal development) would require at least the same amount of formulae (called "statements") and of the well-known difficulty to make sure that the program really does what it should do.

Likewise, the ratio between specification and development, i.e., more effort for the specification than for the development, might appear strange to some readers. We think that this phenomenon is a simple consequence of the fact that formal specifications require a precise and complete statement of the problem and thus leave no room for "handwaving" or for hiding aspects of the problem in the development of an algorithm. Thus, program development starting from a formal specification can exclusively concentrate on making constructs operational and efficient without being bothered by aspects of problem analysis, which obviously reduces the effort compared to the traditional approach to software development.

Although still some more effort has to be invested in order to complete this sample derivation, e.g , by adding a suitable treatment of cursor positions (which we have deliberately left out due to the lack of respective information in the informal requirements), the general way how to deal with such kinds of problems should have become obvious. In particular, an attentive reader with basic knowledge in algebraic specification and transformational programming should be able to do similar developments, e.g for existing, commercially available editors, himself.

Acknowledgement

I would like to thank the referees of SCP, Niek van Diepen, Norbert Völker, and Frank Stomp for their constructive criticism and a lot of valuable remarks on earlier versions of this paper. In particular, I enjoyed several highly technical discussions with Frank, Norbert, and Niek which provided the basis for further essential improvements.

References

[1] F L Bauer and H Wossner, *Algorithmic Language and Program Development* (Springer, Berlin/Heidelberg/New York, 1982)

[2] F L Bauer, R Berghammer, M Broy, W Dosch, F Geiselbrechtinger, R Gnatz, E Hangel, W Hesse, B Krieg-Bruckner, A Laut, T Matzner, B Moller, F Nickl, H Partsch, P Pepper, K Samelson, M Wirsing and H Wossner, *The Munich Project CIP Volume I The Wide Spectrum Language CIP-L*, Lecture Notes in Computer Science **183** (Springer, Berlin/Heidelberg/New York, 1985)

[3] F L Bauer, H Ehler, A Horsch, B Moller, H Partsch, O Paukner and P Pepper, *The Munich Project CIP Volume II The Transformation System CIP-S*, Lecture Notes in Computer Science **292** (Springer, Berlin/Heidelberg/New York, 1987)

[4] F L Bauer, B Moller, H Partsch and P Pepper, Formal program construction by transformations—computer-aided, intuition-guided programming, *IEEE Trans Software Engrg* (1988)

[5] M Broy, Fixed point theory for communication and concurrency, in D Bjørner, Ed , *IFIP TC2 Working Conference on Formal Description of Programming Concepts II, Garmisch-Partenkirchen, June 1982* (North-Holland, Amsterdam, 1983)

[6] M S Feather, A survey and classification of some program transformation approaches and techniques, in L G L T Meertens, Ed , *Program specification and transformation, Proc IFIP TC2 Working Conference, Bad Tolz, April 15-17, 1986* (North-Holland, Amsterdam, 1987)

[7] C A R Hoare Communicating sequential processes, *Comm ACM* **21**(8) (1978) 666-677

[8] I Kupka and N Wilsing, Functions describing interactive programming, in Gunther et al , Eds , *International Computing Symposium 1973* (North-Holland, Amsterdam, 1974)

[9] B Kutzler and F Lichtenberger, *Bibliography on Abstract Data Types*, Informatik-Fachberichte **68** (Springer, Berlin/Heidelberg/New York, 1983)

[10] R Milner, A calculus for communicating systems, Lecture Notes Computer Science **92** (Springer, Berlin/Heidelberg/New York, 1980)

[11] H Partsch, Algebraic requirements definition a case study, *Techn Sci Info* **5**(1) (1986) 21-36

[12] H Partsch, Specification and transformation of programs—a formal approach to software development, Lecture Notes (1988) to appear

[13] H Partsch and B Moller, Konstruktion korrekter Programme durch Transformation *Informatik-Spektrum* **10**(6) (1987) 309-323

[14] H Partsch and P Pepper, Abstract data types as a tool for requirements engineering in D Kronig and G Hommel, Eds, Requirements engineering, *Informatik-Fachberichte* **74** (Springer, Berlin/Heidelberg/New York, 1983) 42-55

[15] H Partsch and R Steinbruggen, Program transformation systems, *ACM Comput Surveys* **15** (1983) 199-236

[16] Working material of the IFIP WG 2 1 meeting, Brussels, Belgium, December 17-21, 1979

[17] M Wirsing, P Pepper, H Partsch, W Dosch and M Broy, On hierarchies of abstract data types, *Acta Informatica* **20** (1983) 1-33