From Descriptive Specifications to Operational ones: A Powerful Transformation Rule, its Applications and Variants

Klaus Achatz, Helmuth Partsch
Universität Ulm

Nr. 96-13
Ulmer Informatik-Berichte
Dezember 1996
From Descriptive Specifications to Operational ones:
A Powerful Transformation Rule, its Applications and Variants

K. Achatz, H. Partsch
Fakultät für Informatik
Universität Ulm

Abstract
This paper introduces, discusses and proves a transformation rule to convert specifications of set-valued functions defined by set comprehension into functional implementations. The power of the rule is illustrated by several examples, among them a Prolog interpreter. Also, variants of the rule for specifications involving existential quantification and arbitrary choice are presented and illustrated by representative examples.

0. Introduction

Transformational programming is short for a methodology of constructing programs from formal problem specifications by stepwise application of formal, semantics-preserving transformation rules. In the following the reader is assumed to be basically familiar with the idea of transformational programming and its basic principles. A brief introduction and an overview can be found, e.g., in [Bauer et al. 89] or [Boiten et al. 92]. For a comprehensive treatment of the subject, cf., e.g., [Partsch 90].

In this paper we mainly introduce a powerful transformation rule to convert specifications of set-valued functions defined by set comprehension into executable functional implementations. This rule can be profitably used for all kinds of problems that ask for a set of complex objects that may somehow be built up incrementally, possibly involving backtracking. The rule emerged when dealing with a formal derivation of a Prolog interpreter from a descriptive specification. It turned out that most transformation steps in this concrete exercise were independent of the particular problem and that the remaining ones could be abstracted appropriately. Meanwhile, the rule has been successfully applied to numerous non-trivial examples, some of which are presented in this paper. More examples and a detailed treatment of those presented here can be found in [Achatz, Partsch 96].
0.1 Notational conventions

Within our rules and examples, we use the following notational conventions.

0.1.1 Fonts

Italic is used for all bound variables. Scheme parameters (for expressions in rules) are denoted by upright-uppercase letters (augmented with "arguments" to indicate potential dependencies). And for types (and type variables) we employ boldface-lowercase symbols.

0.1.2 Data types

Type variables (in program schemes) are denoted by single boldface letters (such as $\mathbf{m}$ or $\mathbf{n}$). As basic types, we assume to have available natural numbers (with type $\mathbf{nat}$ and the usual operations) as well as booleans.

Based on basic types, more complex ones may be built by forming supertypes (e.g., $\mathbf{m} \mid \mathbf{n}$ comprising all elements of types $\mathbf{m}$ and $\mathbf{n}$) and record types (e.g. $(a: \mathbf{m}, b: \mathbf{n})$ denoting pairs of elements of types $\mathbf{m}$ and $\mathbf{n}$ respectively, where $a$ and $b$ are selectors for the first (second) argument).

More complex types we will use are sets, sequences, and maps. For respective formal definitions in terms of algebraic types we refer to [Partsch 90]. The intuition of the basic operations of these types (appearing in the examples) is as follows:

sets (type $\text{set of}$) with operations
- $\emptyset$ the empty set
- $\cup$ set union
- $\subseteq$ subset relation
- $\in$ element relation

sequences (type $\text{sequ of}$) with operations
- $\text{[]} \quad$ the empty sequence
- $\text{hd (fst)}$ first (last) element of a (non-empty) sequence
- $\text{tl (fst)}$ (non-empty) sequence without first (last) element
- $\text{++}$ concatenation of sequences (also used to append single elements)
- $s \cdot x$ removal of element $x$ from sequence $s$
- $s[i]$ $i$-th element of a (non-empty) sequence $s$
- $|s|$ length of sequence $s$
- $\in$ element relation

maps (correspondences defined by an algebraic type $\text{EMAP}$, cf. [Partsch 90]) with operations
- $\emptyset$ the empty map
- $\circ$ composition of maps
- \( m[i] \) the "value" of map \( m \) for "argument" \( i \)
- \( \text{dom} \) the domain of the map
- \( \text{ran} \) the range of the map
- \( m[i] \leftarrow a \) map \( m \) with "value" of \( i \)-th "argument" updated to \( a \).

0.1.3 Programs and program schemes

For programs and program schemes a mainly self-explanatory Pascal-like notation is used with a strict, call-by-value semantics. For a complete definition of the language used, we refer to [Bauer et al. 85]. Some of the less conventional notations (and their meaning) are:

\[
\{a : m \parallel P(a)\} \quad \text{set comprehension, denoting the set of all elements } a \text{ (of type } m) \text{ for which } P(a) \text{ holds}
\]

\text{some } a : m \parallel P(a) \quad \text{(non-deterministic) choice, denoting some element } a \text{ (of type } m) \text{ which satisfies } P(a)

\( (a : m \parallel P(a)) \) subtype of \( m \), comprising all elements \( a \) (of type \( m \)) which fulfil \( P(a) \); also used for assertions in the domain of functions

\( |\ldots| \) cardinality of a set

\( \Delta (\forall) \) sequential conjunction (disjunction)

0.1.4 Transformation rules

Transformation rules are denoted as \( \frac{I}{O} C \) resp. \( \frac{I}{O} C \).

\( I \) is the input scheme of the rule, \( O \) its output scheme. The double arrow means semantic equivalence, the single arrow descendence. \( C \) is a set of applicability conditions, all of which are (implicitly) universally quantified clauses (consisting of a premise, + as an implicational symbol, and a semantic relation – equivalence or descendence – as a consequence). Within applicability conditions, \( = \) is used to denote semantic equivalence, and \( \subseteq \) to denote descendence. If there is no premise, also + will be left out. The syntactic constraints (following a rule) give sufficient (but not necessarily minimal) information to infer complete typing of all expressions and functions involved. For details on the logical background of transformation rules, cf. [Partsch 90].

In order to shorten the presentation of the rules we furthermore use the following

**General convention**

In all transformation rules definedness and determinacy are assumed for all expression symbols and auxiliary functions. Also, all program schemes are supposed to be syntactically valid and context-correct.
0.2 Outline of the following sections

Section 1 introduces our central transformation rule, gives some intuition on how to use it, and provides its correctness proof. Section 2 deals with some applications of the rule. In this context, the N-Queens problem will be dealt with in all details (including all instantiations and proof obligations). The problem of all segmentations of a sequence and a Prolog interpreter are treated in a condensed form. In section 3 some variants of the rule from section 1 are discussed. These variants deal with other specification constructs such as existential quantification or non-deterministic choice and are also illustrated by examples (from the area of parsing). As a particular variant we will also show in this section that, under additional constraints, our rule from section 1 indeed simplifies to a form one would intuitively expect. Section 4, finally, deals with related work and draws some conclusions.

1. A powerful transformation rule

In this section, we present our transformation rule, give some intuition and comments, and prove the correctness of the rule.

1.1 The rule

\[ f(X) \text{ where} \]

\[ f(x) = \text{def } \{ r(X, y): n \parallel Q(x, y) \} \]

(0) \( l\{ r(X, y) \parallel Q(x, y) \} | < \infty \equiv \text{true} \)

(1) \( P(X, X, E) \equiv \text{true} \)

(2) \( Q(x, y) \equiv Q(x, E, y) \)

(3) \( P(X, u, v) \Delta T(u, v) \equiv \text{true} \quad H(u, v) \equiv \{ r(X, y) \parallel Q'(u, v, y) \} \)

(4) \( P(X, u, v) \Delta \neg T(u, v) \equiv \text{true} \quad \}

\( \{ r(X, y) \parallel Q'(u, v, y) \} \equiv \)

\( D(u, v) \cup \bigcup_{i=1\ldots n(u, v)} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \} \)

(5) \( 1 \leq i \leq n(u, v) \Delta P(X, u, v) \Delta \neg T(u, v) \Delta B(i, u, v) \equiv \text{true} \quad P(X, K(i, u, v), R(i, u, v)) \equiv \text{true} \)

(6) \( 1 \leq i \leq n(u, v) \Delta B(i, u, v) \equiv \text{true} \quad (R(i, u, v) < v) \equiv \text{true} \text{ where } WF-ORD(r, <) \text{ or }\)

\( (K(i, u, v) < u) \equiv \text{true} \text{ where } WF-ORD(m, <) \)

(7) \( \text{next}(k, u, v) \subseteq \quad \)

\( \text{some } k': \text{nat} \parallel k' \in \{ i: \text{nat} \parallel k+1 \leq i \leq n(u, v)+1 \Delta \} \quad \forall i': \text{nat} \parallel k < i' < i \Rightarrow \neg B(i', u, v) \} \)

\[ g(X) \text{ where} \]
\[ g(x) = \text{def} \ g'(x, E, \emptyset) \]

\[ g'(u, v, s) = \text{def} \ \begin{cases} \text{if } T(u, v) \text{ then } s \cup H(u, v) \text{ else } g''(\text{next}(0, u, v), u, v, s \cup D(u, v)) & \text{fi} \\ \text{else if } k > n(u, v) \text{ then } s \\ \text{elseif } B(k, u, v) \text{ then } g''(\text{next}(k, u, v), u, v, g'(K(k, u, v), R(k, u, v), s)) \\ \text{else } g''(\text{next}(k, u, v), u, v, s) & \text{fi} \end{cases} \]

**Syntactic constraints**

\[ \text{KIND}[r] = m \times p \rightarrow n \]
\[ \text{KIND}[f, g] = m \rightarrow \text{set of } n \]
\[ \text{KIND}[g'] = (u : m \times v \times r \times s : \text{set of } n \parallel p(X, u, v)) \rightarrow \text{set of } n \]
\[ \text{KIND}[g''] = (k : \text{nat} \times :: m \times v : r \times s : \text{set of } n \parallel p(X, u, v)) \rightarrow \text{set of } n \]
\[ \text{KIND}[\text{next}] = (\text{nat} \times m \times r) \rightarrow \text{nat} \]

### 1.2 Some intuition and comments

The input scheme of the rule captures all kinds of functions to compute a set of values specified by a set comprehension. The output scheme gives an algorithmic way to compute the required set.

Condition (0) guarantees the computability of the solution.

In the definition of \(f\), ZF set abstraction is used as an abbreviation for

\[ \{ q : n \parallel \exists y : p \parallel q = r(X, y) \Delta Q(x, y) \}. \]

\(Q\) is a kind of "input-output relation". The additional operation \(r\) allows to modify "output values" \(y\) dependent on the original input value \(X\). In most applications \(r\) will simply be the identity on \(p\) (see examples below).

The purpose of the additional parameters introduced in \(g'\) is as follows:

- The parameter \(v\) of type \(r\) basically serves to incrementally construct elements of the set aimed at with \(E\) being kind of a neutral element to start with. Type \(r\) is supposed to be a "supertype" of \(p\) -- in most applications identical with \(p\) or a product type with a component of type \(p\).

- The parameter \(s\) of type set of \(n\) is used as an accumulator for the set to be constructed, and, obviously, initialized with \(\emptyset\).

\(P\) is an invariant (for \(g'\) and \(g''\)) to couple the additional parameter of type \(r\) to the original one. To this end, condition (1) asserts the validity of the invariant for the start value \(E\), whereas condition (5) is necessary to maintain the invariant during the computation.

\(Q'\) (in the applicability conditions) is a "generalized" form of the original predicate \(Q\) that takes the incremental construction of elements of type \(r\) into account. In most applications, \(Q'\) will simply be of the form

\[ Q'(u, v, y) \equiv \exists y' \parallel c(v, y') = y \Delta Q(u, y') \quad (\text{or } Q(u, y) \text{ if } u = X) \]

with \(c\) being a "constructor" for elements of type \(r\).
T is a termination condition that serves a two-fold purpose: for the very first call of \( g' \) it has to cover the "trivial case" (i.e., the case when \( E \) is already a solution for the initial input value \( X \)); for all other recursive calls of \( g' \) it has to "signal" that the incremental construction of an element of type \( r \) has been completed.

Condition (3) gives the information how to construct the solution upon termination using an expression \( H \). In most applications \( H \) will be of the simple form \( H(u, v) = \{ r(X, v) \} \), i.e. a singleton set.

Condition (4) gives the information on how to split up the computation of the desired set into subcomputations the number of which, \( n(u, v) \), may (dynamically) depend on the actual parameter values. The expressions \( K \) and \( R \) characterize the modification to the parameters within subcomputations. The expression \( D(u, v) \) is relevant for cases where solutions can be computed "directly" from the parameters. In most applications (cf. below), however, it simply will be instantiated to \( \emptyset \).

Condition (6) postulates the existence of a well-founded ordering on \( r \) or a well-founded ordering on \( m \) which is needed for the proof of termination (cf. below). Obviously, for any application of the rule, this condition always may be replaced by a proof of termination of the (instance of the) function \( g \) – which might be simpler in all those cases where a termination ordering different to the one in the proof of the rule is more appropriate.

Condition (7) characterizes an operation \( \text{next} \) for the incrementation of the "control parameter" \( k \). \( \text{next} \) is characterized non-deterministically on purpose in order to be able to choose a descendant that allows pruning the computation by cutting off unsuccessful branches before they are entered. Obviously, the choice \( \text{next}(k, u, v) = \text{def} \ k+1 \) is a valid descendant that satisfies condition (7). Therefore, in all applications where \( \text{next}(k, u, v) = \text{def} \ k+1 \) is used, the respective proof obligation is simply ignored.

Admittedly, the whole bunch of applicability conditions looks complicated at a first glance, due to the many expressions to be "invented" and the lot of resulting proof obligations. As to the first, one should keep in mind that in most applications the simple forms as commented above are sufficient - which substantially reduces the amount of work for finding suitable instantiations. As to the latter, it will be examplified with one application below, that, whenever suitable instantiations have been found, all proofs of the instantiated applicability conditions are straightforward or even trivial.

### 1.3 Proof of correctness

The correctness proof of the above rule is split up into several steps each of which is proved in a calculational style. The single steps of the respective calculations are labelled for later reference. For the details of the transformations used in reasoning, we refer the reader to [Partsch 90].
a) First, we prove the equivalence of $f(X)$ (in the input scheme) with $g(X)$ (from the output scheme).
   To this end we define
   \[
   g : m \rightarrow \text{set of } n
   \]
   \[
   g(x) = g'(x, E, \emptyset)
   \]
   and
   \[
   g' : (u : m \times v : r \times s : \text{set of } n \parallel P(X, u, v)) \rightarrow \text{set of } n
   \]
   \[
   g'(u, v, s) = s \cup \{r(X, y) \parallel Q'(u, v, y)\}.
   \]
   Then we calculate as follows:
   \[
   g(X)
   \]
   \[
   = [(a1) unfold g; cond. (1)]
   \]
   \[
   g'(X, E, \emptyset)
   \]
   \[
   = [(a2) unfold g'; neutrality of \emptyset \text{ w.r.t. } \cup]
   \]
   \[
   \{r(X, y) \parallel Q(X, E, y)\}
   \]
   \[
   = [(a3) cond. (2)]
   \]
   \[
   \{r(X, y) \parallel Q(X, y)\}
   \]
   \[
   = [(a4) fold f]
   \]
   \[
   f(X).
   \]
   Since $f$ has a defined value due to the general convention and condition (0), so do $g$ and $g'$
   according to the above reasoning.

b) Next, we prove the equivalence of $g'$ (as defined in (a)) with $g'$ (as defined in the output scheme
   of the rule). For $g'$ defined as in (a) we calculate as follows:
   \[
   g'(u, v, s)
   \]
   \[
   = [(b1) unfold g']
   \]
   \[
   s \cup \{r(X, y) \parallel Q'(u, v, y)\}
   \]
   \[
   = [(b2) case-introduction; distributivity \cup \text{ over conditional}]
   \]
   \[
   \text{if } T(u, v) \text{ then } s \cup \{r(X, y) \parallel Q'(u, v, y)\} \text{ else } s \cup \{r(X, y) \parallel Q'(u, v, y)\} \text{ fi}
   \]
   \[
   = [(b3) modification using cond. (3), (4)]
   \]
   \[
   \text{if } T(u, v)
   \]
   \[
   \text{ then } s \cup H(u, v)
   \]
   \[
   \text{ else } s \cup D(u, v) \cup \bigcup_{i = 1 \ldots n(u, v)} \{r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y)\} \text{ fi}
   \]
   \[
   = [(b4) simplification according to cond. (7)]
   \]
   \[
   \text{if } T(u, v)
   \]
   \[
   \text{ then } s \cup H(u, v)
   \]
   \[
   \text{ else } s \cup D(u, v) \cup \bigcup_{i = next(0, u, v) \ldots n(u, v)} \{r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y)\} \text{ fi}
   \]
   \[
   = [(b5) abstraction]
if \( T(u, v) \) then \( s \cup H(u, v) \) else \( g''(\text{next}(0, u, v), u, v, s \cup D(u, v)) \) fi where

\[
g'' : (k : \text{nat} \times u : \text{m} \times v : r \times s : \text{set of } n \parallel P(X, u, v)) \rightarrow \text{set of } n
\]

\[
g''(k, u, v, s) = \text{def } s \cup \bigcup_{i = k.n(u, v)} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \}
\]

(c) Thirdly, we have to show that \( g'' \) (as defined in (b)) is equivalent to \( g'' \) from the output scheme of the rule. For \( g'' \) as defined in (b) we reason as follows:

\[
g''(k, u, v, s)
\]

\[
= [\text{(c1) unfold } g'']
\]

\[
s \cup \bigcup_{i = k.n(u, v)} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \}
\]

\[
= [\text{(c2) case-introduction; simplification of then-branch}]
\]

if \( k > n(u, v) \)

then \( s \)

else \( s \cup \bigcup_{i = k.n(u, v)} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \} \) fi.

For the else-branch we reason as follows:

\[
s \cup \bigcup_{i = k.n(u, v)} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \}
\]

\[
= [\text{(c3) \cup-split}]
\]

\[
s \cup \{ r(X, y) \parallel B(k, u, v) \Delta Q'(K(k, u, v), R(k, u, v), y) \} \cup
\]

\[
\bigcup_{i = k+1..\text{next}(k,u,v)-1} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \} \cup
\]

\[
\bigcup_{i = \text{next}(k,u,v)..n(u,v)} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \}
\]

\[
= [\text{(c4) simplification: def. next } \Rightarrow \bigcup_{i = k+1..\text{next}(k,u,v)-1} \{ \ldots \} = \emptyset]
\]

\[
s \cup \{ r(X, y) \parallel B(k, u, v) \Delta Q'(K(k, u, v), R(k, u, v), y) \} \cup
\]

\[
\bigcup_{i = \text{next}(k,u,v)..n(u,v)} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \} \] fi

\[
= [\text{(c5) case introduction; simplification}]
\]

if \( B(k, u, v) \)

then \( s \cup \{ r(X, y) \parallel Q'(K(k, u, v), R(k, u, v), y) \} \)

\[
\bigcup_{i = \text{next}(k,u,v)..n(u,v)} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \}
\]

else \( s \cup \bigcup_{i = \text{next}(k,u,v)..n(u,v)} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \} \) fi

\[
= [\text{(c6) fold } g'' - \text{ cond. (5) guarantees assertion; termination below}]
\]

if \( B(k, u, v) \)

then \( g''(K(k, u, v), R(k, u, v), s) \)

\[
\bigcup_{i = \text{next}(k,u,v)..n(u,v)} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \}
\]

else \( s \cup \bigcup_{i = \text{next}(k,u,v)..n(u,v)} \{ r(X, y) \parallel B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \} \) fi

\[
= [\text{(c7) fold } g'' - \text{ cond. (5) guarantees assertion; termination below}]
\]

if \( B(k, u, v) \) then \( g''(\text{next}(k, u, v), u, v, g''(K(k, u, v), R(k, u, v), s)) \)

else \( g''(\text{next}(k, u, v), u, v, s) \) fi.
Altogether, we have:

\[
g''(k, u, v, s) = \begin{cases} \text{if } k > n(u, v) & \text{then } s \\ \text{elsif } B(k, u, v) & \text{then } g''(\text{next}(k, u, v), u, v, g'(K(k, u, v), R(k, u, v), s)) \\ \text{else} & g''(\text{next}(k, u, v), u, v, s) \end{cases}
\]

d) It remains to prove termination of \(g'\) and \(g''\) which is required for the correctness of the folding steps. By our general convention, all expressions are defined. In the result obtained so far, \(g'\) can be eliminated by unfolding its calls in \(g\) and \(g''\):

\[
g(X) \text{ where}
g(x) = \begin{cases} \text{if } T(x, E) & \text{then } \emptyset \cup H(x, E) \text{ else } g''(\text{next}(0, x, E), x, E, \emptyset \cup D(x, E)) \\ \end{cases}
\]

\[
g''(k, u, v, s) = \begin{cases} \text{if } k > n(u, v) & \text{then } s \\ \text{elsif } B(k, u, v) & \text{then } g''(\text{next}(k, u, v), u, v, \\ & \text{if } T(K(i, u, v), R(i, u, v)) \\ & \text{then } s \cup H(K(i, u, v), R(i, u, v)) \\ & \text{else } g''(\text{next}(0, K(i, u, v), R(i, u, v)), K(i, u, v), R(i, u, v), s) \end{cases}
\]

Simplification and distributivity of function call over conditional (in \(g''\)) then yields

\[
g''(k, u, v, s) = \begin{cases} \text{if } k > n(u, v) & \text{then } s \\ \text{elsif } B(k, u, v) & \text{then if } T(K(i, u, v), R(i, u, v)) \\ & \text{then } g''(\text{next}(k, u, v), u, v, s \cup H(K(i, u, v), R(i, u, v))) \\ & \text{else } g''(\text{next}(k, u, v), u, v, \\ & g''(\text{next}(0, K(i, u, v), R(i, u, v)), K(i, u, v), R(i, u, v), s) \end{cases}
\]

\[
\text{else } g''(\text{next}(k, u, v), u, v, s) \end{cases}
\]

In this last version, termination of \(g\) is obvious provided \(g''\) terminates. To prove termination of \(g''\) we define the following (lexicographic) termination ordering \(\preceq\) on \(\text{nat} \times \text{m} \times \text{r} \times \text{set of n}:

\[
(k, u, v, s) \preceq (k', u', v', s') \iff v < v' \lor (v = v' \land n(u, v) - k < n(u, v) - k') \lor u < u' \lor (u = u' \land n(u, v) - k < n(u, v) - k')
\]

Then, for the recursive calls of \(g''\) we have
\[(\text{next}(k, u, v), u, v, s) \cup H(K(i, u, v), R(i, u, v)) \preceq (k, u, v, s) \] by def. of \(\preceq\) and \text{next}.

\[(\text{next}(0, K(i, u, v), R(i, u, v)), K(i, u, v), R(i, u, v), s) \preceq (k, u, v, s) \] by cond. (6).

\[(\text{next}(k, u, v), u, v, g''(\text{next}(0, K(i, u, v), R(i, u, v)), K(i, u, v), R(i, u, v), s)) \preceq (k, u, v, s) \] by def. of \(\preceq\) and \text{next}.

\[(\text{next}(k, u, v), u, v, s) \preceq (k, u, v, s) \] by def. of \(\preceq\) and \text{next}.

Hence, \(g''\) terminates.

2. Some applications of the rule

In this section we deal with a couple of (non-trivial) problems to illustrate the power of our rule and how to use it. The first one, the N-Queens problem, is given in all details. For the remaining ones, the necessary instantiations and proof obligations can be found in the appendix.

2.1 The N-Queens Problem

The well-known N-Queens Problem asks for finding all ways to place \(N\) mutually nonattacking queens on a \(N \times N\) chessboard.

Using the abbreviation

\[n\text{sequ} =_{\text{def}} \text{sequ of nat}\]

for sequences to represent the positions in the different columns of the chessboard, the formalization of this problem is straightforward:

\[\text{queens}(N) \text{ where}\]

\[\text{queens: nat} \rightarrow \text{set of nsequ} \]

\[\text{queens}(n) =_{\text{def}} \{ sp : \text{nsequ} \parallel lsp = n \land \text{ispsequ}(sp, n) \land \text{nconf}(sp) \}\]

where

\[\text{ispsequ: nsequ \times nat} \rightarrow \text{bool}\]

\[\text{ispsequ}(s, n) =_{\text{def}} \forall i : \text{nat} \parallel 1 \leq i \leq lsl \parallel s[i] \leq n\]

restricts the elements of \text{nsequ} to those less or equal to \(n\), and

\[\text{nconf: nsequ} \rightarrow \text{bool}\]

\[\text{nconf}(s) =_{\text{def}} \forall (i, j : \text{nat} \parallel 1 \leq i, j \leq lsl \land i \neq j) \parallel s[i] \neq s[j] \land \text{abs}(s[i] - s[j]) \neq \text{abs}(i - j)\]

guarantees that all positions in the different columns are "nonattacking", i.e., do not mutually occupy the same row or the same diagonal.

Matching this specification with the input scheme of our rule yields (part of) the instantiations, viz.
\[ f \] by \( \textit{queens} \)
\[ X \] by \( N \)
\[ x \] by \( n \)
\[ y \] by \( sp \)
\[ r(X, y) \] by \( sp \)
\[ Q(x, y) \] by \( lsp = n \land ispossequ(sp, n) \land nconf(sp) \).

If we further instantiate
\[ g \] by \( qu \)
\[ g' \] by \( qu' \)
\[ g'' \] by \( qu'' \)
\[ u \] by \( n \)
\[ v \] by \( p \)
\[ P(X, u, v) \] by \( lpl \leq n \land ispossequ(p, n) \land nconf(p) \)
\[ E \] by \( [] \) (* the empty sequence of columns to start with *)
\[ Q'(u, v, y) \] by \( \exists s' :: \textit{nsequ} \parallel sp = p' + s' \land lsp = n \land ispossequ(sp, n) \land nconf(sp) \)
(* the existence of a possible extension of \( p \) to obtain a complete sequence of positions with the requested properties *)
\[ T(u, v) \] by \( lpl = n \)
\[ H(u, v) \] by \( \{p\} \)
\[ D(u, v) \] by \( \emptyset \)
\[ n(u, v) \] by \( n \)
\[ B(i, u, v) \] by \( nc(p, k) \) where
\[ nc :: \textit{nsequ} \times \textit{nat} \to \textit{bool} \]
\[ nc(p, k) =_{\text{def}} \forall \ (i :: \textit{nat} \parallel 1 \leq i \leq |p|) :: \]
\[ p[i] \neq k \land \textit{abs}(p[i] - k) \neq \textit{abs}(i - (|p| + 1)) \]
(* check whether \( k \) does not "attack" any of the elements of \( p \) *)
\[ K(i, u, v) \] by \( n \)
\[ R(i, u, v) \] by \( p' + k \)
\[ next(k, u, v) \] by \( \textit{next} :: \textit{nat} \times \textit{nat} \times \textit{nsequ} \to \textit{nat} \)
\[ next(k, n, p) =_{\text{def}} \min \ {i :: \textit{nat} \parallel k + 1 \leq i \leq n + 1 \Delta} \]
\[ \forall i' :: \textit{nat} \parallel k < i' < i \Rightarrow i' \in p \}
\[ \text{WF-ORD}(r, <) \] by \( (\textit{nsequ}, a < b \iff n - |a| < n - |b|) \)

we obtain as result of the application of the rule
\[ qu(N) \) where
\[ qu :: \textit{nat} \to \textit{set of nsequ} \]
\[ qu(n) =_{\text{def}} q'(n, [], \emptyset) \]
\[ q : \text{nat} \times \text{nsequ} \times \text{set of nsequ} \rightarrow \text{set of nsequ} \]
\[ q'(n, p, s) = \text{def if } |pl| = n \text{ then } s \cup \{p\} \text{ else } q''(\text{next}(0, n, p), n, p, s) \text{ fi} \]

\[ q'' : \text{nat} \times \text{nat} \times \text{nsequ} \times \text{set of nsequ} \rightarrow \text{set of nsequ} \]
\[ q''(k, n, p, s) = \text{def if } k > n \text{ then } s \]
\[ \text{elsif } \text{nc}(p, k) \text{ then } q''(\text{next}(k, n, p), n, p, q'(n, p+k, s)) \]
\[ \text{else } q''(\text{next}(k, n, p), n, p, s) \text{ fi} \]

It remains to prove the instantiated applicability conditions (which is straightforward or even trivial):

\[(0) \quad (\{sp : \text{nsequ} \parallel lspl = n \land ispossequ(sp, n) \land nconf(sp)\} | < \infty) = \text{true} \]

Trivial, as already \text{nsequ} restricted by \text{ispossequ} has only finitely many elements (for any \(n\)).

\[(1) \quad (\{[]\} \leq n \land ispossequ([], n) \land nconf([], n)) = \text{true} \]

Obviously, \(|[]| = 0 \leq n\), \text{ispossequ}([], n), and \text{nconf}([], n) are all true.

\[(2) \quad \text{lspl} = n \land ispossequ(sp, n) \land nconf(sp) = \exists s': \text{nsequ} \parallel sp = [++]\text{'} \land \text{lspl} = n \land ispossequ(sp, n) \land nconf(sp) \]

Trivial, as for \(s' = sp\) the right-hand side expression simplifies to the one on the left-hand side.

\[(3) \quad (\text{lspl} = n \land ispossequ(sp, n) \land nconf(p)) \Delta \text{lspl} = n = \text{true} \uparrow \]
\[ \{p\} = \{sp : \text{nsequ} \parallel \exists s' : \text{nsequ} \parallel sp = p++]\text{'} \land \text{lspl} = n \land ispossequ(sp, n) \land nconf(sp)\} \]

From \(|sp| = n\), \(|lspl| = n\), and \(sp = p++]\text{'}\), it follows that \(|ls'| = 0\), and, thus, \(s' = [\]\). Hence, the set on the right-hand side of the consequence simplifies to

\[\{sp : \text{nsequ} \parallel sp = p \land \text{lspl} = n \land ispossequ(sp, n) \land nconf(sp)\}.\]

The equality of both sets then follows from the premise on \(p\).

\[(4) \quad (\text{lspl} = n \land ispossequ(sp, n) \land nconf(p)) \Delta \text{lspl} < n = \text{true} \uparrow \]
\[ \{sp : \text{nsequ} \parallel \exists s' : \text{nsequ} \parallel sp = p++]\text{'} \land \text{lspl} = n \land ispossequ(sp, n) \land nconf(sp)\} = \bigcup_{i = 1\ldots n} \{sp : \text{nsequ} \parallel \text{nc}(p, i) \Delta \exists s' : \text{nsequ} \parallel sp = (p++]\text{''} + s' \land \text{lspl} = n \land ispossequ(sp, n) \land nconf(sp)\} \]

Due to the premise \(|sp| < n\) and associativity of ++, the left-hand side of the consequence can be rewritten into

\[\{sp : \text{nsequ} \parallel \exists s' : \text{nsequ}, \text{i: pos}(n) \parallel \quad sp = (p++]\text{''} + s' \land \text{lspl} = n \land ispossequ(sp, n) \land nconf(sp)\}\]
where \( \text{pos}(n) \) abbreviates \((i: \text{nat} \mid 1 \leq i \leq n)\).

Since \( \neg \text{nc}(p, i) \) implies \( \neg \text{nconf}(p++i++) \), the existential quantification on \( i \) can be restricted to those \( i \) that satisfy \( \text{nc}(p, i) \). The requested equality then follows from the general rule

\[
\{x \parallel \exists y, z \parallel \text{P}(z) \land Q(x, y, z)\} = \bigcup_z \{x \parallel \exists y \parallel Q(x, y, z)\}.
\]

\(5\)

\[
1 \leq i \leq n \Delta \{l|p| \leq n \land \text{ispoxeq}(p, n) \land \text{nconf}(p)\} \Delta |p| < n \Delta \text{nc}(p, i) \equiv \text{true} \uplus \text{l}p++i| \leq n \land \text{ispoxeq}(p++i, n) \land \text{nconf}(p++i) \equiv \text{true}
\]

\(|p| < n\) implies \(|p++i| \leq n\), \(\text{ispoxeq}(p++i, n)\) follows immediately from the premise, and \(\text{nconf}(p++i)\) follows from \(\text{nconf}(p)\) and \(\text{nc}(p, i)\).

\(6\)

\[
1 \leq i \leq n \Delta \text{nc}(p, i) \equiv \text{true} \uplus n - |p++i| < n - |p| \equiv \text{true}
\]

The consequence is even true for arbitrary \(n, p\), and \(i\).

\(7\)

\[
\text{min} \{i: \text{nat} \parallel k+1 \leq i \leq n + 1 \Delta \forall i': \text{nat} \parallel k < i' < i \Rightarrow i' \in p\} \subseteq \text{some } k': \text{nat} \parallel k' \in \{i: \text{nat} \parallel k+1 \leq i \leq n + 1 \Delta \forall i': \text{nat} \parallel k < i' < i \Rightarrow \neg \text{nc}(p, i')\}\]

Obviously, \(i' \in p\) implies \(\neg \text{nc}(p, i')\). \(\text{min } m\) is always a descendant of \(\text{some } x \parallel x \in m\).

**Remarks**

- The choice of \(\text{next}\) as defined above (instead of the generally possible choice \(\text{next}(k) =_{\text{def}} k+1\)) is a simple optimization (according to the remark on \(\text{next}\) in section 1).

- As a further improvement, we could have added another parameter to keep all possible candidate positions to be added to the current sequence. This would require modified instantiations, viz.

\[
\text{P}(X, u, v) \quad \text{by} \quad |p| \leq n \land \text{ispoxeq}(p, n) \land \text{nconf}(p) \land \forall (i: \text{pos}(n) \parallel i \in c) \parallel \text{nc}(p, i)
\]

\[
\text{E} \quad \text{by} \quad ([], \{i: \text{pos}(n)\})
\]

\[
\text{B}(i, u, v) \quad \text{by} \quad c \neq \emptyset
\]

\[
\text{R}(i, u, v) \quad \text{by} \quad (p++i, \{k: \text{pos}(n) \parallel \text{nc}(p++i, k)\}).
\]

Together with

\[
\text{next}(k, c) =_{\text{def}} \text{if } c \neq \emptyset \text{ then } \text{min } c \text{ else } n+1 \text{ fi}
\]

these modified instantiations then would have resulted in

\[
\text{qu}(N) \text{ where}
\]

\[
\text{qu}(n) =_{\text{def}} q'(n, [], \{i: \text{pos}(n)\}, \emptyset)
\]

\[
q'(n, p, c, s) =_{\text{def}} \text{if } |p| = n \text{ then } s \cup \{p\} \text{ else } q''(\text{next}(0, c), n, p, c, s) \text{ fi}
\]

- 13 -
\[ q''(k, n, p, c, s) = \begin{cases} 
\text{def} & \text{if } k > n \text{ then } s \\
\text{elsif } c \neq \emptyset \text{ then } q''(\text{next}(k, c), n, p, c, q'(n, p++k, \{i: \text{pos}(n)\} \parallel nc(p++k, i)), s) \\
\text{else} q''(\text{next}(k, c), n, p, c, s) \end{cases} \]

As a further improvement – which is left to the interested reader – one could think of applying finite differencing to efficiently compute the new candidate set in the first recursive call of \( q'' \).

### 2.2 Permutations

The problem asks for finding all permutations of a given sequence (without duplications) of some kind of elements.

Using

\[
\text{ndsequ} = \text{def} \ « \text{sequences without duplicate elements} »
\]

and

\[
\text{same-els: ndsequ} \times \text{ndsequ} \rightarrow \text{bool} \\
\text{same-els}(q, sp) = \text{def} \ « q \text{ and } sp \text{ have exactly the same elements} »
\]

a formalization of the problem is obvious:

\[
\text{perms}(Q) \text{ where} \\
\text{perms:} \text{ndsequ} \rightarrow \text{set of ndsequ} \\
\text{perms}(q) = \text{def} \{ sp: \text{ndsequ} \parallel \text{same-els}(q, sp) \}.
\]

The result of rule application is

\[
p(Q) \text{ where} \\
p: \text{ndsequ} \rightarrow \text{set of ndsequ} \\
p(q) = \text{def} p'(q, [], \emptyset) \\
p': \text{ndsequ} \times \text{ndsequ} \times \text{set of ndsequ} \rightarrow \text{set of ndsequ} \\
p'(q, qp, s) = \text{def} \begin{cases} 
\text{if } q = [] \text{ then } s \cup \{qp\} \text{ else } p''(1, q, qp, s) \end{cases} \fi \\
p'': \text{nät} \times \text{ndsequ} \times \text{ndsequ} \times \text{set of ndsequ} \rightarrow \text{set of ndsequ} \\
p''(k, q, qp, s) = \text{def} \begin{cases} 
\text{if } k > \text{lql} \text{ then } s \\
\text{elsif true then } p''(k+1, q, qp, p'(q-q[k], qp++q[k], s)) \\
\text{else } p''(k+1, q, qp, s) \end{cases} \fi
\]

where \( p'' \) can be further simplified to

\[
p''(k, q, qp, s) = \text{def} \begin{cases} 
\text{if } k > \text{lql} \text{ then } s \\
\text{else } p''(k+1, q, qp, p'(q-q[k], qp++q[k], s)) \end{cases} \fi.
\]
2.3 Segmentations (of a non-empty sequence)

The problem is as follows: Given an arbitrary non-empty sequence $s$ (of a certain kind of elements), it is requested to compute all possible "segmentations" of $s$, i.e., all possible ways of cutting $s$ into non-empty subsequences.

Using

\[
\text{msequ} \equiv \text{def} \ « \text{sequences with elements of type m} » ,
\]
\[
\text{ssequ} \equiv \text{def} \ (s; \text{sequ of msequ} \, \| \, \forall \ 1 \leq i \leq \|s\| \, s[i] \neq [\,]) .
\]

and

\[
\text{flatten: ssequ} \rightarrow \text{msequ}
\]

\[
\text{flatten}(sp) \equiv \text{def} \ \text{if} \ sp = [\,] \ \text{then} \ [\,] \ \text{else} \ \text{hd}(sp) ++ \text{flatten}(\text{tl}(sp)) \ \text{fi}
\]

The problem may be formalized by

\[
\text{all-segs}(Q) \text{ where}
\]

\[
\text{all-segs: msequ} \rightarrow \text{set of ssequ}
\]

\[
\text{all-segs}(q) \equiv \text{def} \ \{ sp; \text{ssequ} \, \| \, \text{flatten}(sp) = q \} .
\]

The "direct" result of rule application (i.e., without subsequent simplifications) is

\[
\text{segs}(Q) \text{ where}
\]

\[
\text{segs: msequ} \rightarrow \text{set of ssequ}
\]

\[
\text{segs}(q) \equiv \text{def} \ \text{segs'(q, [\,], \emptyset)}
\]

\[
\text{segs': msequ} \times \text{ssequ} \times \text{set of ssequ} \rightarrow \text{set of ssequ}
\]

\[
\text{segs'}(q, qp, s) \equiv \text{def} \ \text{if} \ q = [\,] \ \text{then} \ s \cup \{ qp \} \ \text{else} \ \text{segs''(1, q, qp, s) fi}
\]

\[
\text{segs'': nat} \times \text{msequ} \times \text{ssequ} \times \text{set of ssequ} \rightarrow \text{set of ssequ}
\]

\[
\text{segs''(k, q, qp, s) \equiv def}
\]

\[
\text{if} \ k > 2 \ \text{then} \ s
\]

\[
\text{elsif true then segs''(k+1, q, qp, segs'(tl q, if k=1 then q++[hd q] else q++[[hd q]] fi, s))}
\]

\[
\text{else segs''(k+1, q, qp, s) fi}
\]

where

\[
\text{++_lst: ssequ} \times \text{msequ} \rightarrow \text{ssequ}
\]

\[
qp++_{\text{lst}} \equiv \text{def} \ \text{if} \ qp = [\,] \ \text{then} \ [x] \ \text{else} \ \text{fst}(qp)+(\text{lst}(qp++) + x) \ \text{fi}
\]

By simplification of the outer conditional and distributivity of function call over conditional, the definition of segs'' can be further transformed into
segs"(k, q, qp, s) = def
  if k > 2 then s
  elsif k = 1 then segs"(k+1, q, qp, segs'(tl q, qp++lst[hdq], s))
  else segs"(k+1, q, qp, segs'(tl q, qp++[hdq], s)) fi.

By successive unfoldings and simplifications, segs"(1, q, qp, s) can be further transformed into

segs'(tl q, qp++[hdq], segs'(tl q, qp++[hdq], s)).

Hence, segs" can be eliminated and the final version of segs' reads:

segs'(q, qp, s) = def
  if q = [] then s ∪ {qp} else segs'(tl q, qp++[hdq], segs'(tl q, qp++[hdq], s)) fi

2.4 Partitions of a natural number (> 0)

This problem (which is closely related to the previous one) asks for finding all possible ways of representing a positive natural number as a sum.

Again, a formalization is straightforward:

all-parts(Q) where

all-parts: pnat → set of psequ

all-parts(q) = def {sp: psequ || sum(sp) = q}

where

pnat = def (n: nat || n > 0 )

psequ = def sequ of pnat

and

sum: psequ → nat

sum(sp) = def if sp = [] then 0 else hdsp + sum(tlsp) fi.

The result of rule application is

parts(Q) where

parts: pnat → set of psequ

parts(q) = def parts'(q, [], Ø)

parts': pnat × psequ × set of psequ → set of psequ

parts'(q, qp, s) = def if q = 0 then s ∪ {qp} else segs"(1, q, qp, s) fi
\textit{parts} : \textit{nat} \times \textit{pnat} \times \textit{psequ} \times \text{set of} \textit{psequ} \to \text{set of} \textit{psequ}

\textit{parts}''(k, q, qp, s) \equiv_{\text{def}}

\begin{array}{l}
\quad \text{if } k > 2 \quad \text{then } s \\
\quad \text{elsif } true \text{ then } \textit{parts}''(k+1, q, qp, \textit{parts}'(q-1, \text{if } k=1 \text{ then } qp++[1] \text{ else } qp++[1] \text{ fi}, s)) \\
\quad \text{else } \textit{parts}''(k+1, q, qp, s) \text{ fi}
\end{array}

where

\begin{array}{l}
\textit{+}_{\text{lst}} : \textit{psequ} \times \textit{nat} \to \textit{psequ} \\
qp+_{\text{lst}} x \equiv_{\text{def}} \text{if } qp = [] \text{ then } [x] \text{ else } \textit{fst}qp++(\textit{lst}qp+x) \text{ fi}
\end{array}

Again, \textit{parts}'' can be further transformed (using simplification of the outer conditional and distributivity of function call over conditional) into

\textit{parts}''(k, q, qp, s) \equiv_{\text{def}}

\begin{array}{l}
\quad \text{if } k > 2 \text{ then } s \\
\quad \text{elsif } k=1 \text{ then } \textit{parts}''(k+1, q, qp, \textit{parts}'(q-1, qp+_{\text{lst}}1, s)) \\
\quad \text{else } \textit{parts}''(k+1, q, qp, \textit{parts}'(q-1, qp++[1], s)) \text{ fi}
\end{array}

By successive unfoldings and simplifications, \textit{parts}''(1, q, qp, s) can be further transformed into

\textit{parts}'(q-1, qp++[1], \textit{parts}'(q-1, qp+_{\text{lst}}1, s)).

Hence, \textit{parts}'' can be eliminated and the final version of \textit{parts}’ reads:

\textit{parts}'(q, qp, s) \equiv_{\text{def}}

\begin{array}{l}
\quad \text{if } q = 0 \text{ then } s \cup \{qp\} \text{ else } \textit{parts}'(q-1, qp++[1], \textit{parts}'(q-1, qp+_{\text{lst}}1, s)) \text{ fi}
\end{array}

\textbf{2.4 All subsequences}

The problem is as follows: Given an arbitrary sequence \(s\) (of a certain kind of elements), it is requested to compute the set of all subsequences.

Using

\textbf{msequ} \equiv_{\text{def}} \langle \text{sequences with elements of type } m \rangle,

and

\textit{issubsequ} : \textbf{msequ} \times \textbf{msequ} \to \textit{bool}

\textit{issubsequ}([], t) \equiv_{\text{def}} \textit{true}

\textit{issubsequ}(x++s, []) \equiv_{\text{def}} \textit{false}

\textit{issubsequ}(x++s, y++t) \equiv_{\text{def}} (x = y \land \textit{issubsequ}(s, t)) \lor \textit{issubsequ}(x++s, t)

the problem may be formalized by

- 17 -
all-sub\(s(Q)\) where

\[
\text{all-sub}\!: \text{msequ} \to \text{set of msequ}
\]

\[
\text{all-sub}(q) =_{\text{def}} \{ q' : \text{msequ} \sqcup \text{issubsequ}(q', q) \}.
\]

The result of rule application is

\[
\text{subs}(Q)\text{ where}
\]

\[
\text{subs}\!: \text{msequ} \to \text{set of msequ}
\]

\[
\text{subs}(q) =_{\text{def}} \text{subs}'(q, [], \emptyset)
\]

\[
\text{subs}'\!: \text{msequ} \times \text{msequ} \times \text{set of msequ} \to \text{set of msequ}
\]

\[
\text{subs}'(q, p, s) =_{\text{def}} \text{if } q = [] \text{ then } s \cup \{p\} \text{ else } \text{subs}''(1, q, p, s)
\]

\[
\text{subs}''\!: \text{nat} \times \text{msequ} \times \text{msequ} \times \text{set of msequ} \to \text{set of msequ}
\]

\[
\text{subs}''(k, q, p, s) =_{\text{def}}
\]

\[
\begin{align*}
\text{if } k > 2 & \text{ then } s \\
\text{else true then } & \text{subs}''(k+1, q, p, \text{subs}'(\text{tl} q, \text{if } k=1 \text{ then } p \text{ else } p++[\text{hd} q] \text{ fi}, s)) \\
\text{else } & \text{subs}''(k+1, q, p, s) \text{ fi}.
\end{align*}
\]

By simplification of the outer conditional and distributivity of function call over conditional, the definition of \(\text{subs}''\) can be further transformed into

\[
\text{subs}''(k, q, p, s) =_{\text{def}}
\]

\[
\begin{align*}
\text{if } k > 2 & \text{ then } s \\
\text{else } & k=1 \text{ then } \text{subs}''(k+1, q, p, \text{subs}'(\text{tl} q, p, s)) \\
\text{else } & \text{subs}''(k+1, q, p, \text{subs}'(\text{tl} q, p++[\text{hd} q], s)) \text{ fi}.
\end{align*}
\]

By successive unfoldings and simplifications, \(\text{subs}''(1, q, p, s)\) can be further transformed into

\[
\text{subs}'(\text{tl} q, p++[\text{hd} q], \text{subs}'(\text{tl} q, p, s)).
\]

Hence, \(\text{subs}''\) can be eliminated and the final version of \(\text{subs}'\) reads:

\[
\text{subs}'(q, p, s) =_{\text{def}}
\]

\[
\begin{align*}
\text{if } q = [] & \text{ then } s \cup \{p\} \text{ else } \text{subs}'(\text{tl} q, p++[\text{hd} q], \text{subs}'(\text{tl} q, p, s)) \text{ fi}
\end{align*}
\]

Remarks

- Obviously, the above remains correct, if we substitute \text{msequ} by

\[
\text{ndsequ} =_{\text{def}} \text{« sequences without duplicate elements »}.
\]

In this way, the above treatment also covers the problem of all subsets of a set.
Also, the problem of computing all subsequences less than (or greater than) a given size \( n \) is captured, if we instantiate

\[
H(u, v) \quad \text{by} \quad \begin{cases} \text{if } |p| < n \text{ then } \{p\} \text{ else } \emptyset \end{cases} \textit{fi}
\]

and simplify afterwards.

2.6 The Pack-Problem

This problem – which is related to the one dealt with in [Partsch 90], on p. 138 and p. 256 – is a typical representative of a Knapsack problem. Given a collection of blocks (of known size) and collection of boxes (of known size), it is asked for finding all ways of associating blocks to boxes such that all blocks fit into their associated box.

Using

\[
\begin{align*}
\textit{blocks} & \quad \text{def} \quad \textit{sequ of block} \\
\textit{boxes} & \quad \text{def} \quad \textit{sequ of box} \\
\textit{corr} & \quad \text{def} \quad \textit{EMAP(block, box, =)}
\end{align*}
\]

the problem can be formally specified by

\[
able-packs(K, B) \quad \text{where}
\]

\[
able-packs: \quad \textit{blocks} \times \textit{boxes} \rightarrow \text{set of corr}
\]

\[
able-packs(k, b) = \text{def} \quad \{A: \textit{corr} \parallel \textit{legal}(A, k, b)\}
\]

where

\[
\textit{legal}: \quad \textit{corr} \times \textit{blocks} \times \textit{boxes} \rightarrow \text{bool}
\]

\[
\textit{legal}(a, k, b) = \text{def} \quad \text{dom}(a) = k \land b \supseteq \text{ran}(a) \land \textit{stowable}(b, a)
\]

\[
\textit{stowable}: \quad \textit{boxes} \times \textit{corr} \rightarrow \text{bool}
\]

\[
\textit{stowable}(b, a) = \text{def} \quad \forall (i: \text{nat} \parallel 1 \leq i \leq |b|) \parallel (\Sigma_{ke, x: \text{dom}(a) \parallel a[x] = b[i]} \text{ size}(k)) \leq \text{size}(b[i])
\]

guarantees that (a) all blocks are associated to a box, (b) all associated boxes are available, and (c) all blocks fit into their associated box.

The result of rule application is

\[
ap(K, B) \quad \text{where}
\]

\[
ap: \quad \textit{blocks} \times \textit{boxes} \rightarrow \text{set of corr}
\]

\[
ap(k, b) = \text{def} \quad \textit{ap}'(k, b, \emptyset, \emptyset)
\]

\[
ap': \quad \textit{blocks} \times \textit{boxes} \times \textit{corr} \times \text{set of corr} \rightarrow \text{set of corr}
\]

\[
ap'(k, b, a, s) = \text{def} \quad \begin{cases} \text{if } k = [] \text{ then } s \cup \{a\} \text{ else } \textit{ap}''(1, k, b, a, s) \end{cases} \textit{fi}
\]

- 19 -
\[ \text{ap}'' : \text{nat} \times \text{blocks} \times \text{boxes} \times \text{corr} \times \text{set of corr} \rightarrow \text{set of corr} \]

\[ \text{ap}''(i, k, b, a, s) =\text{def} \]

\[ \text{if } i > |b| \text{ then } s \]

\[ \text{else if } \text{fits}(\text{hdlk}, b[i]) \text{ then } \text{ap}''(i+1, k, b, a, \text{ap}')(\text{tlk}, b[i \Theta \text{hdkl}], a[\text{hd}k] \leftarrow b[i], s) \]

\[ \text{else } \text{ap}''(i+1, k, b, a, s) \text{ fi} \]

where

\[ \text{fits} : \text{block} \times \text{box} \rightarrow \text{bool} \]

\[ \text{fits}(x, y) =\text{def} \text{ size}(x) \leq \text{size}(y) \]

and

\[ \Theta : \text{boxes} \times \text{nat} \times \text{block} \rightarrow \text{boxes} \]

\[ b[i \Theta k] =\text{def} \text{ « b updated such that the size of its } i\text{-th component is decreased by } k \text{ »} \]

2.7 Topological Sortings

Given a partial ordering (by a set of pairs of elements), this problem asks for finding all total orderings of the elements (involved in the partial ordering) that "preserve" the partial ordering, i.e., whenever \( a < b \) due to the partial ordering, then also \( a < b \) in the total ordering.

In order to represent partial orderings, we use

\[ \text{pord} =\text{def} (po : \text{set of } (m, m) \parallel \forall (a, b) : (m, m) \parallel (a, b) \in po \Rightarrow (b, a) \notin po). \]

The set of elements (involved in the partial ordering) is represented by

\[ \text{ndsequ} =\text{def} \text{ « sequences without duplicate elements »} \]

and obtained from the originally given partial ordering by

\[ \text{els} : \text{pord} \rightarrow \text{ndsequ} \]

\[ \text{els}(po) =\text{def} \text{ set-to-sequ(}\{a : m \parallel \exists b : m \parallel (a, b) \in po \lor \neg (b, a) \in po\}) \]

(where \text{set-to-sequ} converts a set into a sequence).

With these prerequisites our problem may be formalized as follows:

\[ \text{top-sort}(\text{els}(PO), PO) \text{ where} \]

\[ \text{top-sort} : \text{ndsequ} \times \text{pord} \rightarrow \text{set of ndsequ} \]

\[ \text{top-sort}(m, po) =\text{def} \{t : \text{ndsequ} \parallel \text{same-els}(t, m) \land \text{respects}(t, po)\} \]

where

\[ \text{same-els} : \text{ndsequ} \times \text{ndsequ} \rightarrow \text{bool} \]

\[ \text{same-els}(q, sp) =\text{def} \text{ « } q \text{ and } sp \text{ have exactly the same elements »} \]

- 20 -
and

\[ \text{respects: } \text{ndsequ} \times \text{ndsequ} \rightarrow \text{bool} \]
\[ \text{respects}(i, po) = \text{def} \ \forall 1 \leq i, j \leq |l| \ || i < j \lor (i[i], j[j]) \notin po. \]

The result of rule application is

\[ ts(\text{els}(PO), PO) \ \text{where} \]

\[ ts: \text{ndsequ} \times \text{pord} \rightarrow \text{set of ndsequ} \]
\[ ts(m, po) = \text{def} \ ts'(m, po, [], \emptyset) \]

\[ ts': \text{ndsequ} \times \text{pord} \times \text{ndsequ} \times \text{set of ndsequ} \rightarrow \text{set of ndsequ} \]
\[ ts'(m, po, to, s) = \text{def} \ \begin{cases} & \text{if } m = [] \ \text{then } s \cup \{to\} \ \text{else } ts''(1, m, po, to, s) \ \text{fi} \\ & \text{if } k > |ml| \ \text{then } s \\ & \phantom{\text{if } k > |ml| \ \text{then } s} \ \text{elsif } ismin(m[k], po) \ \text{then } ts''(k+1, m, po, to, ts'(m-m[k], \text{rem}(m[k], po), to++m[k], s)) \\ & \phantom{\text{if } k > |ml| \ \text{then } s} \ \text{elsif } ismin(m[k], po) \ \text{then } ts''(k+1, m, po, to, ts'(m-m[k], \text{rem}(m[k], po), to++m[k], s)) \\ & \phantom{\text{if } k > |ml| \ \text{then } s} \ \text{else } ts''(k+1, m, po, to, s) \ \text{fi} \end{cases} \]

where

\[ ismin: m \times \text{pord} \rightarrow \text{bool} \]
\[ ismin(x, q) = \text{def} \ \exists y \ || (y, x) \in q \]

and

\[ \text{rem: } m \times \text{pord} \rightarrow \text{pord} \]
\[ \text{rem}(x, q) = \text{def} \ q - \{(x, y) || (x, y) \in q\}. \]

2.8 Maximal paths in a graph

Given a finite, directed graph (by a set of nodes and – for each node – a sequence of its successors), this problem asks for finding all paths (without cycles) of maximal length starting from a given node.

In order to represent finite directed graphs, we use

\[ \text{graph} = \text{def} \ « \text{finite, directed graphs as defined above} » , \]
\[ \text{node} = \text{def} \ « \text{nodes of a graph} » , \]

and

\[ \text{nsequ} = \text{def} \ (sn: \text{sequ of node} \ || sn \neq \emptyset) . \]

The successors of a node w.r.t. a given graph can be obtained by
sucses: node × graph → set of node

sucses(n, g) =def « sequence of successors of n in g ».

With these prerequisites our problem may be formalized as follows:

\textit{all-paths}(N, G) \textbf{where}

\textit{all-paths}: node × graph → set of nsequ

\textit{all-paths}(n, g) =def \{ t: nsequ || ismaximal(t, n, g) \}

where

\textit{ismaximal}: nsequ × node × graph → bool

\textit{ismaximal}(t, n, g) =def hdt = n ∧ isend(t, g) ∧ connected(t, g) ∧ acyclic(t)

\textit{isend}: nsequ × graph → bool

\textit{isend}(t, g) =def \textit{sucses}(lstmt, g) = \[] \lor \forall (i: \textit{nat} \parallel 1 \leq i < \textit{|sucses(lstmt, g)|}) \parallel \textit{sucses}(lstmt, g)[i] \in t

\textit{connected}: nsequ × graph → bool

\textit{connected}(t, g) =def \forall (i: \textit{nat} \parallel 1 \leq i < \textit{|t|}) \parallel t[i+1] \in \textit{sucses}(t[i], g)

and

\textit{acyclic}: nsequ → bool

\textit{acyclic}(t) =def \forall (i, j: \textit{nat} \parallel 1 \leq i, j \leq \textit{|t|}) \parallel i \neq j \Rightarrow t[i] \neq t[j].

The result of rule application is

\textit{paths}(N, G) \textbf{where}

\textit{paths}: node × graph → set of nsequ

\textit{paths}(n, g) =def \textit{paths'}(n, g, [n], ∅)

\textit{paths'}: node × graph × nsequ × set of nsequ → set of nsequ

\textit{paths'}(n, g, ns, s) =def if isend(ns, g) then s \cup \{ns\} \textbf{ else } \textit{paths'}(1, n, g, ns, s) \textbf{ fi}

\textit{paths'': nat × node × graph × nsequ × set of nsequ → set of nsequ}

\textit{paths''}(k, n, g, ns, s) =def

if \textit{k} > \textit{|sucses(lstmt, g)|} then \textbf{s}

else sucses(lstmt, g)[k] \notin ns \textbf{ then } \textit{paths''}(k+1, n, g, ns, \textit{paths'}(n, g, ns ++ sucses(lstmt, g)[k], s))

else \textbf{fi}.

2.9 A Prolog Interpreter

A Prolog interpreter takes a Prolog program and a query and yields as a result either the set of all substitutions (for the variables in the query) such that the instantiated query is a logical consequence
from the program, or it does not terminate. If there is no substitution (such that the instantiated query follows from the program) the result is the empty set.

In order to formally specify the problem, we first have to define the notions of program, query, and substitution. A program is a sequence of clauses (or rules) each of which consists of a literal as its left-hand side and a (possibly empty) sequence of literals as its right-hand side. A literal in turn consists of a predicate symbol and a (possibly empty) sequence of terms (as arguments). And a term is either a constant symbol, a variable symbol, or a function symbol together with a sequence of terms as arguments. All this can straightforwardly be formalized as follows:

\[
\begin{align*}
\text{program} & \equiv \text{sequ of clause} \\
\text{clause} & \equiv (\text{lhs: literal, rhs: sequ of literal}) \\
\text{literal} & \equiv (\text{ps: predsymb, st: sequ of term}) \\
\text{term} & \equiv \text{const | var | (functsymb, sequ of term)}.
\end{align*}
\]

A query is a sequence of literals, and a substitution is an association of variable symbols with terms. Again, formalization is straightforward:

\[
\begin{align*}
\text{query} & \equiv \text{sequ of literal} \\
\text{subst} & \equiv \text{EMAP(var, term, =)}.
\end{align*}
\]

In order to construct a substitution and simultaneously to prove that the instantiated query \( gl \) follows from the program \( p \), a Prolog interpreter uses sld-resolution which is captured by the following derivability relation:

\[
\frac{\text{query } \times \text{ subst } \times \text{ query } \times \text{ subst } \to \text{ bool}}{p} \\
\begin{align*}
(gl, \Theta) \Rightarrow (gl', \Theta \cdot \Theta') & \equiv \text{def} \\
\exists \ c \in \ p \ \parallel \ gl' = (\text{rhs}(c')) ++ \text{tlgl})\Theta' \land c' = \text{rn}(c, gl) \land \text{unify(hd}gl, \text{lhs}(c')) \neq \text{fail } \Delta \\
\Theta' & = \text{unify(hd}gl, \text{lhs}(c'))
\end{align*}
\]

where

\( \Theta \cdot \Theta' \) denotes the composition of the substitutions \( \Theta \) and \( \Theta' \).

\( a\Theta \) denotes the application of the substitution \( \Theta \) to \( a \).

\( \text{rn: clause } \times \text{ query } \to \text{ clause} \)

\( \text{rn}(c, gl) \equiv \langle \text{clause } c \text{ with all variables renamed such that they do not occur in } gl \rangle \)

and (with \( c_i \) and \( v_i \) denoting constant and variable symbols, receptively, \( f, g \) denoting function symbols, and \( r_i, s_i \) denoting terms)

\[
\begin{align*}
\text{struct} & \equiv \text{term | literal} \\
\text{unify: struct } \times \text{ struct } \to \text{ subst | fail} \\
\text{unify}(c_1, c_2) & \equiv \text{def if } c_1 = c_2 \text{ then } \emptyset \text{ else fail } \text{fi} \\
\text{unify}(c_1, v_2) & \equiv \text{def } (c_1 \text{ for } v_2)
\end{align*}
\]

- 23 -
unify(c₁, g(r₁, ..., rₖ)) =def fail
unify(v₁, c₂) =def (c₂ for v₁)
unify(v₁, v₂) =def if v₁ ≠ v₂ then (v₂ for v₁) else ⌀ fi
unify(v₁, g(r₁, ..., rₖ)) =def
  if notoccurs(v₁, g(r₁, ..., rₖ)) then (g(r₁, ..., rₖ) for v₁) else fail fi
unify(f(s₁, ..., sₙ), c₂) =def fail
unify(f(s₁, ..., sₙ), v₂) =def
  if notoccurs(v₂, f(s₁, ..., sₙ)) then (f(s₁, ..., sₙ) for v₂) else fail fi
unify(f(s₁, ..., sₙ), g(r₁, ..., rₖ)) =def
  if f = g ∧ n = k then unifylist([s₁, ..., sₙ], [r₁, ..., rₖ], ⌀) else fail fi

notoccurs: var × struct → bool
notoccurs(v, s) =def « v does not occur in s »

unifylist: sequ of struct × sequ of struct × subst → subst | fail
unifylist([], [], ⌀) =def ⌀
unifylist(g++, gl, h++, hl, ⌀) =def
  if ⌀ ≠ fail then unifylist(glΘ', hlΘ', ⌀ · Θ') else fail fi
  where Θ' = unify(g, h)

Equipped with this environment, a formal specification of a Prolog interpreter is as follows:

interpret(P, A) where

interpret: program × query → set of subst
interpret(p, ql) =def \{ Θ | var(A) \parallel (gl, ⌀) \mathcal{P}^{*} ([], Θ) \}

where

Θ | var(A) denotes Θ restricted to those variable symbols occurring in A

and

\mathcal{P}^{*} denotes the reflexive transitive closure of \mathcal{P} .

Correct application of our transformation rule requires to satisfy the applicability condition

(\{ Θ | var(A) \parallel (gl, ⌀) \mathcal{P}^{*} ([], Θ) \}) < ∞) \equiv \text{true}.

Whereas in the previous examples, the corresponding proof obligation was always straightforward to be shown, it is less trivial here. Indeed, it only can be proved if both the program and the query satisfy additional constraints the detailed treatment of which is beyond the scope of this paper. A similar remark holds for condition (6) about the existence of a suitable well-founded ordering. Therefore we simply take these applicability conditions for granted and yield as result of rule application:
\(\text{int}(P, A)\) where

\[
\text{int}: \text{program} \times \text{query} \rightarrow \text{set of subst} \\
\text{int}(p, gl) \equiv \text{int}(p, gl, \emptyset, \emptyset)
\]

\[
\text{int'}: \text{program} \times \text{query} \times \text{subst} \times \text{set of subst} \rightarrow \text{set of subst} \\
\text{int'}(p, gl, \Phi, s) \equiv \text{if } gl = [] \text{ then } s \cup \{ \Phi \mid \text{vars}(A) \} \text{ else } \text{int"}(\text{next}(0, p, \text{hdgl}), p, gl, \Phi, s) \text{ fi}
\]

\[
\text{int"}: \text{nat} \times \text{program} \times \text{query} \times \text{subst} \times \text{set of subst} \rightarrow \text{set of subst} \\
\text{int"}(k, p, gl, \Phi, s) \equiv \text{if } k > |p| \text{ then } s \\
\text{else if } \Phi' \neq \text{fail} \\
\text{then } \text{int"}(\text{next}(k, p, \text{hdgl}), p, gl, \text{int'}(p, (\text{rhs}(\text{rn}(p[k], gl)) + \text{tlgl})\Phi', \Phi' \times \Phi', s)) \\
\text{else } \text{int"}(\text{next}(k, p, \text{hdgl}), p, gl, \Phi, s) \text{ fi where } \Phi' = \text{unify(\text{hdgl}, \text{lhs}(\text{rn}(p[k], gl)))}
\]

\[
\text{next}: \text{nat} \times \text{literal} \rightarrow \text{nat} \\
\text{next}(k, p, l) \equiv \text{min}\{i: \text{nat} \mid (k+1 \leq i \leq |p| \Delta pslhs(p[i])) = psls(l) \} \lor i = |p|+1
\]

2.10 Further examples

Of course, there are many more problems our rule might be applied to. These range e.g. from mathematical problems (e.g., all subsets of a set, all partitions of a set, all subsets of a set up to a fixed size, cf. remark at the end of section 2.5), combinatorial problems (e.g., space-filling puzzles, other known problems on the chessboard), or other graph problems.

3. Variants of the general rule and their applications

In the following we discuss a couple of "variants" of our general rule and illustrate them by respective examples. These "variants" look very similar to our general rule, although they deal with different kinds of specifications and require (slightly) different applicability conditions. Using the word "variant" is simply motivated by the fact that the proofs of these rules only marginally deviate from the proof of our general rule.

3.1 Variant 1: set defined by a union

3.1.1 The rule

\(f(X)\) where

\[
f(x) \equiv \bigcup_{z \in \{y \mid Q(x, y)\}} R(X, z)
\]
\textbf{g(X) where}

\[ g(x) \stackrel{\text{def}}{=} g'(x, E, \emptyset) \]

\[ g'(u, v, s) \stackrel{\text{def}}{=} \begin{cases} T(u, v) & \text{then } s \cup H(u, v) \text{ else } g''(\text{next}(0, u, v), u, v, s) \text{ fi} \\ \text{else} & B(k, u, v) \text{ then } g''(\text{next}(k, u, v), u, v, g'(K(k, u, v), R(k, u, v), s)) \\ & \text{else } g''(\text{next}(k, u, v), u, v, s) \text{ fi} \end{cases} \]

\textbf{Syntactic constraints}

\[ \text{KIND}[r] = m \times p \rightarrow \text{set of } n \]

\[ \text{KIND}[f, g] = m \rightarrow \text{set of } n \]

\[ \text{KIND}[g'] = (u: m \times v: r \times s: \text{set of } n || P(X, u, v)) \rightarrow \text{set of } n \]

\[ \text{KIND}[g''] = (k: \text{nat} \times u: m \times v: r \times s: \text{set of } n || P(X, u, v)) \rightarrow \text{set of } n \]

\[ \text{KIND}[\text{next}] = (\text{nat} \times m \times r) \rightarrow \text{nat} \]

\textbf{Proof}

Exactly as for general rule (with \( g(x) \stackrel{\text{def}}{=} s \cup \bigcup_{z \in \{ y \; \mid \; Q(x, y) \}} r(X, z) \)) in step (a) and \( D(u, v) \)

\textbf{instantiated by } \emptyset). \]

\subsection{3.1.2 An application: The Coding Problem}

This problem is also treated (in a different way) in [Partsch 90] (p. 132, p. 247) and reads as follows: Given a non-empty "clear word" \( N \) (over some character set \( V_1 \)) and a "coding rule" \( P \) (which is a sequence of pairs consisting of a non-empty word over \( V_1 \) and a non-empty "code word", i.e. a non-empty word over a character set \( V_2 \)), it is requested to compute the set of all encodings of \( N \) with respect to \( P \).

Using

\[ \text{clear} \equiv \text{def} \rightarrow \text{clear words} \]

\[ \text{code} \equiv \text{def} \rightarrow \text{code words} \]
rule = \text{def} (l: \text{clear}, r: \text{code})

a possible formalization of the problem is as follows:

encodings(N, P) where

encodings: clear \times sequ of rule \rightarrow set of code

encodings(n, p) = \text{def} \bigcup_{a \in \text{deco}(n, p)} \text{encode } (a, p)

where

deco: clear \rightarrow sequ of clear

deco(n, p) = \text{def} \{ a : \text{sequ of clear} \parallel \text{flatten}(a) = n \land \text{codable}(a, p) \}

encode: sequ of clear \times rule \rightarrow set of code

encode(a, p) = \text{def}

\{ c : \text{code} \parallel \exists b : \text{sequ of code} \parallel
\text{flatten}(b) = c \land \| b \| = \| a \| \land \forall (i: \text{nat} \parallel 1 \leq i \leq \| a \|) \parallel \exists (j: \text{nat} \parallel 1 \leq j \leq \| p \|) \parallel b[i] = r(p[j]) \}

and

codable: sequ of clear \times sequ of rule \rightarrow bool

codable(a, p) = \text{def} \forall (i: \text{nat} \parallel 1 \leq i \leq \| a \|) \parallel \exists (j: \text{nat} \parallel 1 \leq j \leq \| p \|) \parallel a[i] = l(p[j])

flatten: sequ of code \rightarrow code

flatten(b) = \text{def} if b = [] then [] else hd b ++ \text{flatten}(tl b) fi

The result of rule application is

enc(N, P) where

enc: (clear \times sequ of rule) \rightarrow set of code

enc(n, p) = \text{def} enc(n, p, [[]], \emptyset)

enc': (clear \times rule \times set of code \times set of code) \rightarrow set of code

enc'(n, p, e, s) = \text{def} if n = [] then s \cup e else enc''(1, n, p, e, s) fi

enc'': (nat \times clear \times rule \times set of code \times set of code) \rightarrow set of code

enc''(k, n, p, e, s) = \text{def}

if k > \| p \| \text{ then } s

elsif l(p[k]) \cdot n \text{ then } enc''(k+1, n, p, e, enc''(n-\| l(p[k]) \|, p, e \oplus r(p[k]), s))

else enc''(k+1, n, p, e, s) fi

where

\cdot: (clear \times clear) \rightarrow bool

a \cdot b = \text{def} \ « a \text{ is an initial segment of } b »
\[\vdash: \text{(clear } \times \text{ clear)} \rightarrow \text{ clear}\]
\[a \vdash b \overset{\text{def}}{=} b \text{ with initial segment } a \text{ removed}\]

\[\oplus: \text{(set of code } \times \text{ code)} \rightarrow \text{ set of code}\]
\[a \oplus b \overset{\text{def}}{=}[a'++b \parallel a' \in a]\]

3.2 Variant 2: Existential quantification

3.2.1 The rule

\[f(X) \text{ where}\]
\[f(x) \overset{\text{def}}{=} \exists y: p \parallel Q(x, y)\]

\[1), (2), (5), (6), (7) \text{ as in general rule}\]
(3) \[P(X, u, v) \Delta T(u, v) \equiv \text{true} \vdash \text{true} \equiv \exists y: p \parallel Q'(u, v, y)\]
(4) \[P(X, u, v) \Delta \neg T(u, v) \equiv \text{true} \vdash\]
\[\exists y: p \parallel Q'(u, v, y) \equiv \exists_{i=1..n(u, v)} \exists y: p \parallel B(i, u, v) \Delta Q(K(i, u, v), R(i, u, v), y)\]

\[g(X) \text{ where}\]
\[g(x) \overset{\text{def}}{=} g'(x, E, \text{false})\]
\[g'(u, v, s) \overset{\text{def}}{=} \text{if } T(u, v) \text{ then true else } g''(\text{next}(0, u, v), u, v, s) \text{ fi}\]
\[g''(k, u, v, s) \overset{\text{def}}{=} \text{if } s \vee k > n(u, v) \text{ then } s\]
\[\text{elsif } B(k, u, v) \text{ then } g''(\text{next}(k, u, v), u, v, g'(K(k, u, v), R(k, u, v), s))\]
\[\text{else } g''(\text{next}(k, u, v), u, v, s) \text{ fi}\]

Syntactic constraints
\[\text{KIND}[f, g] = m \rightarrow \text{bool}\]
\[\text{KIND}[g] = (u: m \times v: r \times s: \text{bool} \parallel P(X, u, v)) \rightarrow \text{bool}\]
\[\text{KIND}[g''] = (k: \text{nat} \times u: m \times v: r \times s: \text{bool} \parallel P(X, u, v)) \rightarrow \text{bool}\]
\[\text{KIND}[\text{next}] = (\text{nat} \times m \times r) \rightarrow \text{nat}\]

Proof (changes apart from instantiating \(r\) by the identity and \(D(u, v)\) by \(\varnothing\))

a) \[g(x) \overset{\text{def}}{=} s \vee \exists y: p \parallel Q(x, y)\]
(a2'): unfold \( g' \): neutrality of false w.r.t. \( \lor \)
b) (b2'): case-introduction: distributivity \( \lor \) over conditional
c) (c3'): \( \lor \)-split in else-branch

3.2.2 An Application: Top-down recognition (for context-free grammars)

Given a context-free grammar \( G = (N, T, Z, P) \) (where \( N \) is a non-empty set of nonterminal symbols, \( T \) a non-empty set of terminal symbols which is disjoint from \( N \), \( Z \in N \), and \( P \) is a non-empty set of context-free productions consisting of a nonterminal symbol as its left-hand side and a (possibly empty) sequence of (nonterminal or terminal) symbols as its right-hand side) and a terminal word \( w \), it is requested to check whether \( w \) is derivable from \( Z \) using \( P \).

Using

\[
\text{nont} = \text{def} \quad \text{« nonterminal symbols »} \\
\text{term} = \text{def} \quad \text{« terminal symbols »} \\
\text{symb} = \text{def} \quad \text{nont} \mid \text{term} \\
\text{prod} = \text{def} \quad \text{(lhs: nont, rhs: sequ of symb)}
\]

the problem can be formalized by

\[
\text{recognize}(P, W) \quad \text{where}
\]

\[
\text{recognize: sequ of prod} \times \text{sequ of term} \rightarrow \text{bool} \]

\[
\text{recognize}(p, w) = \text{def} \quad \exists y: \text{sequ of term} \parallel [Z] \quad \rightarrow^* \quad y \land y = w
\]

where

\[
\rightarrow: \text{sequ of symb} \times \text{sequ of symb} \rightarrow \text{bool} \\
x \rightarrow y = \text{def} \quad \exists l, r: \text{sequ of symb}, i: \text{nat} \parallel x = l++\text{lhs}(p[i])++r \land y = l++\text{rhs}(p[i])++r
\]

and

\[
\rightarrow^* \quad \text{denotes the reflexive transitive closure of} \quad \rightarrow
\]

The result of rule application is

\[
\text{rec}(P, W) \quad \text{where}
\]

\[
\text{rec: sequ of prod} \times \text{sequ of term} \rightarrow \text{bool} \\
\text{rec}(p, w) = \text{def} \quad \text{rec}(p, w, [Z], \text{false})
\]

\[
\text{rec': sequ of prod} \times \text{sequ of term} \times \text{sequ of symb} \times \text{bool} \rightarrow \text{bool} \\
\text{rec'}(p, w, v, s) = \text{def} \quad \text{if} \quad w = [] \land v = [] \quad \text{then true else} \quad \text{rec'}(\text{next}(1, p, v), p, w, v, s) \quad \text{fi}
\]
rec" : nat \times \text{sequ of prod} \times \text{sequ of term} \times \text{sequ of symb} \times \text{bool} \rightarrow \text{bool}

rec" (k, p, w, v, s) = \text{def}
  \begin{array}{l}
  \text{if } s \lor k > |p|+1 \text{ then } s \\
  \text{else if } (k \leq |p| \land v \neq [] \land \Delta \text{hd}_v = lhs(p[k])) \lor (k = |p|+1 \land w \neq [] \land v \neq [] \land \Delta \text{hd}_w = \text{hd}_v) \\
  \text{then rec"} (\text{next}(k, p, v), p, w, v, \text{rec'}(p, \text{if } k \leq |p| \text{ then w else tl}_w \text{ fi}), \\
  \text{if } k \leq |p| \text{ then } rhs(p[k])++tl_v \text{ else } tl_v \text{ fi}, s)) \\
  \text{else rec"} (\text{next}(k, p, v), p, w, v, s) \text{ fi}
  \end{array}

next : nat \times \text{sequ of prod} \times \text{sequ of symb} \rightarrow \text{nat}

next(k, p, v) = \text{def} \min \{ i : \text{nat} \parallel (k+1 \leq i \leq |p| \land \Delta \text{lhs}(p[i]) = \text{hd}_v) \lor i = |p|+1 \}

where rec" can be further transformed into

rec" (k, p, w, v, s) = \text{def}
  \begin{array}{l}
  \text{if } s \lor k > |p|+1 \text{ then } s \\
  \text{else if } k \leq |p| \land v \neq [] \land \Delta \text{hd}_v = lhs(p[k]) \\
  \text{then rec"} (\text{next}(k, p, v), p, w, v, \text{rec'}(p, w, rhs(p[k])++tl_v, s)) \\
  \text{else if } k = |p|+1 \land w \neq [] \land v \neq [] \land \Delta \text{hd}_w = \text{hd}_v \\
  \text{then rec"} (\text{next}(k, p, v), p, w, v, \text{rec'}(p, tl_w, tl_v, s)) \\
  \text{else rec"} (\text{next}(k, p, v), p, w, v, s) \text{ fi}
  \end{array}

Similar to the Prolog interpreter, proving termination is slightly more difficult than in most of the other examples. In fact, as is known, absence of left-recursion in the grammar is necessary for the proof of termination.

Of course, we could have chosen an even "stronger" definition for next, viz.

next(k, p, v) = \text{def} \min \{ i : \text{nat} \parallel (k+1 \leq i \leq |p| \land \Delta \text{lhs}(p[i]) = \text{hd}_v \land \exists z : \text{symb} \parallel \text{hd}_v \rightarrow \text{hd}_w++z) \lor i = |p|+1 \}

which obviously reflects the idea of LL(1) recognition.

### 3.3 Variant 3: Non-deterministic choice

#### 3.3.1 The rule

f(X) where

f(x) = \text{def} \text{ if } \{ y : p \parallel Q(x, y) \} \neq \emptyset \text{ then some } y : p \parallel Q(x, y) \text{ else dummy fi}
(1), (2), (5), (6), (7) as in general rule
(3) \[ P(X, u, v) \Delta T(u, v) \equiv \text{true} \vdash H(u, v) \in \{y \parallel Q'(u, v, y)\} \equiv \text{true} \]
(4) \[ P(X, u, v) \Delta \neg T(u, v) \equiv \text{true} \vdash \]
\[ \text{some } y: p \parallel Q'(u, v, y) \equiv \text{some } y: p \parallel \exists_{i=1,...,n(u, v)} (B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y)) \]

\[ g(X) \text{ where} \]
\[ g(x) =_{\text{def}} g'(x, E, \text{dummy}) \]
\[ g'(u, v, s) =_{\text{def}} \text{if } T(u, v) \text{ then } H(u, v) \text{ else } g''(\text{next}(0, u, v), u, v, s) \text{ fi} \]
\[ g''(k, u, v, s) =_{\text{def}} \text{if } s = \text{dummy } \lor k > n(u, v) \text{ then } s \]
\[ \text{else } B(k, u, v) \text{ then } g''(\text{next}(k, u, v), u, v, g'(K(k, u, v), R(k, u, v), s)) \]
\[ \text{else } g''(\text{next}(k, u, v), u, v, s) \text{ fi} \]

Syntactic constraints
\[ \text{KIND}[f, g] = m \rightarrow n \mid \text{dummy} \]
\[ \text{KIND}[	ext{g'}] = (u: m \times v: r \times s: n \mid \text{dummy} \parallel P(X, u, v)) \rightarrow n \mid \text{dummy} \]
\[ \text{KIND}[g''] = (k: \text{nat} \times u: m \times v: r \times s: n \mid \text{dummy} \parallel P(X, u, v)) \rightarrow n \mid \text{dummy} \]
\[ \text{KIND}[\text{next}] = (\text{nat} \times m \times r) \rightarrow \text{nat} \]

Proof (changes apart from instantiating \( r \) by the identity and \( D(u, v) \) by \( \emptyset \))

a) \[ g(x) =_{\text{def}} \text{dummy } \odot \text{ some } y: p \parallel Q(x, y) \]

where
\[ a \odot b =_{\text{def}} \text{ if } \text{def}(a) \text{ then } a [] \text{def}(b) \text{ then } b \text{ else } \text{dummy } \text{fi where} \]
\[ \text{def(some } y \parallel Q) =_{\text{def}} \{y \parallel Q\} \neq \emptyset \]
\[ \text{def}(x) =_{\text{def}} x \neq \text{dummy}. \]
(a2'): unfold \( g' \)
(a3'): cond. (2); unfold \( \odot \)

b) (b2'): case-introduction
(b3'): \( \subseteq \) for \( \equiv \)

c) (c2'): \( \subseteq \) for \( \equiv \)
(c3'): \( \lor \)-split in else-branch
(c5'): case introduction in else-branch;

simplification using some \( y \parallel B \lor C \equiv \text{some } y \parallel B \odot \text{some } y \parallel C \)
3.3.2 An Application: Shift-Reduce parsing (of context-free grammars)

Given a context-free grammar $G$ (as defined in the previous example) and a terminal word $w$, it is asked for computing a sequence of shift and reduce actions the application of which allows to reduce $w$ to the axiom $Z$ of $G$ provided such a sequence exists; if not, the result should be error.

For a formal specification of this problem we define nonterm, term, symbol, and product as in the previous example. The sequence of shift and reduce actions is defined by

\[
\text{srseq} \equiv \text{sequences of } S (= \text{shift}) \text{ and } R(X, \alpha) (= \text{reduce with } (X, \alpha))\,.
\]

The application of such a sequence to a terminal word is specified by

\[
\text{apply: } \text{seq of product} \times \text{seq of symbol} \times \text{seq of term} \to \text{seq of symbol} \times \text{seq of term}
\]
\[
\text{apply}([], st, w) \equiv (st, w)
\]
\[
\text{apply}(S++sr, st, t++w) \equiv \text{apply}(sr, st++t, w)
\]
\[
\text{apply}(R(X, \alpha)+sr, st++\alpha, w) \equiv \text{apply}(sr, st++X, w).
\]

With these prerequisites, the original problem may be specified as follows:

\[
\text{parse}(P, [], W) \text{ where}
\]
\[
\text{parse: } \text{seq of product} \times \text{seq of symbol} \times \text{seq of term} \to \text{srseq} \mid \text{error}
\]
\[
\text{parse}(p, st, w) \equiv \text{if } \{ y: \text{srseq} \parallel \text{apply}(y, st, w) = ([Z], []) \} \neq \emptyset
\]
\[
\text{then some } y: \text{srseq} \parallel \text{apply}(y, st, w) = ([Z], []) \text{ else error fi.}
\]

The result of rule application is

\[
\text{par}(P, [], W) \text{ where}
\]
\[
\text{par: } \text{seq of product} \times \text{seq of symbol} \times \text{seq of term} \to \text{srseq} \mid \text{error}
\]
\[
\text{par}(p, st, w) \equiv \text{par'}(p, st, w, [], \text{error})
\]
\[
\text{par': } \text{seq of product} \times \text{seq of symbol} \times \text{seq of term} \times \text{srseq} \times \text{srseq} \mid \text{error} \to \text{srseq} \mid \text{error}
\]
\[
\text{par'}(p, st, w, sr, s) \equiv \text{if } w = [] \land st = [Z] \text{ then } sr \text{ else } \text{par}''(1, p, st, w, sr, s) \text{ fi}
\]
\[ \text{par}^\prime \text{: nat} \times \text{sequ of prod} \times \text{sequ of symb} \times \text{sequ of term} \times \text{srsequ} \times \text{srsequ} \times \text{error} \rightarrow \] 
\[ \text{srsequ} \times \text{error} \]
\[ \text{par}^\prime (k, p, st, w, sr, s) \equiv_{\text{def}} \]
\[ \text{if } s \neq \text{error} \lor k > |p| + 1 \text{ then } s \]
\[ \text{elsif } (k \leq |p| \land \exists s' \parallel st = \text{s'} + \text{rhs}(p[k]) \lor (k = |p| + 1 \land w \neq []) \]
\[ \text{then } \text{par}''(k+1, p, st, w, sr, p, \text{if } k \leq |p| \text{ then } (s' + \text{lhs}(p[k]), w) \text{ else } (sr + \text{hdw}, \text{tlw}) \text{ fi} ,] \]
\[ \text{if } k \leq |p| \text{ then } sr + \text{R}(p[k]) \text{ else } sr + \text{S} \text{ fi, } s) \]
\[ \text{else } \text{par}''(k+1, p, st, w, sr, s) \text{ fi.} \]

where \( \text{par}'' \) can be further transformed into

\[ \text{par}''(k, p, st, w, sr, s) \equiv_{\text{def}} \]
\[ \text{if } s \neq \text{error} \lor k > |p| + 1 \text{ then } s \]
\[ \text{elsif } k \leq |p| \land \exists s' \parallel st = \text{s'} + \text{rhs}(p[k]) \]
\[ \text{then } \text{par}''(k+1, p, st, w, sr, p, s' + \text{lhs}(p[k]), w, sr + \text{R}(p[k]), s) \]
\[ \text{elsif } k = |p| + 1 \land w \neq [] \]
\[ \text{then } \text{par}''(k+1, p, st, w, sr, p, s' + \text{hdw}, \text{tlw}, sr + \text{S}, s) \]
\[ \text{else } \text{par}''(k+1, p, st, w, sr, s) \text{ fi.} \]

Again, as in the case of top-down recognition, additional constraints on the grammar are necessary for the proof of termination.

### 3.4 Variant 4: without incremental construction of the elements of the solution

#### 3.4.1 The rule

\[ f(X) \text{ where} \]
\[ f(x) \equiv_{\text{def}} \{ r(X, y): n \parallel Q(x, y) \} \]

\[
\begin{align*}
(0) & \text{ as in general rule} \\
(3) & \text{T} (u) \equiv \text{true} \uplus H (u) \equiv \{ r(X, y) \parallel Q (u, y) \} \\
(4) & \neg \text{T} (u, v) \equiv \text{true} \uplus \{ r(X, y) \parallel Q (u, y) \} = \text{D} (u) \cup \bigcup_{i = 1..n (u)} \{ r(X, y) \parallel B (i, u) \Delta Q (K (i, u), y) \} \\
(6) & 1 \leq i \leq n (u) \Delta B (i, u) \equiv \text{true} \uplus (K (i, u) < u) \equiv \text{true} \text{ where WF-ORD} (m, <) \\
(7) & \text{next} (k, u) \subseteq \text{some } k': \text{n} \parallel k' \in \{ i: \text{n} \parallel k + 1 \leq i \leq n(u) + 1 \Delta \forall i': \text{n} \parallel k < i' < i \Rightarrow \neg B (i', u) \}
\end{align*}
\]
\( g(X) \) where

\[
g(x) =_{\text{def}} g'(x, \emptyset)
\]

\[
g'(u, s) =_{\text{def}} \begin{cases} T(u) & \text{then } s \cup H(u) \text{ else } g''(\text{next}(0, u), u, s \cup D(u)) \end{cases}
\]

\[
g''(k, u, s) =_{\text{def}} \begin{cases} k > n(u) & \text{then } s \cup D(u) \\
\text{else} & B(k, u) \text{ then } g''(\text{next}(k, u), u, g'(K(k, u), s)) \\
\text{else} & g''(\text{next}(k, u), u, s) \end{cases}
\]

Syntactic constraints

\[
\text{KIND}[r] = m \times p \to n
\]

\[
\text{KIND}[f, g] = m \to \text{set of } n
\]

\[
\text{KIND}[g'] = (m \times \text{set of } n) \to \text{set of } n
\]

\[
\text{KIND}[g''] = (\text{nat} \times m \times \text{set of } n) \to \text{set of } n
\]

\[
\text{KIND}[\text{next}] = (\text{nat} \times m) \to \text{nat}
\]

Proof (changes)

a) \( g(x) =_{\text{def}} g'(x, \emptyset) \);

\[
g'(u, s) =_{\text{def}} s \cup \{ r(X, y) : n \parallel Q(x, y) \};
\]

\( (a1') \): unfold \( g \)

\( (a3') \): −

3.4.2 An Application: Reachable Nodes in graph

Given a finite, directed graph (by a set of nodes and – for each node – a sequence of its successors), this problem asks for finding the set of all nodes reachable from a given node.

In order to represent finite directed graphs, we use (as in section 2.8)

\[
\text{graph} =_{\text{def}} \text{ « finite, directed graphs as defined above »,}
\]

\[
\text{node} =_{\text{def}} \text{ « nodes of a graph »}.
\]

The successors of a node w.r.t. a given graph can be obtained by

\[
\text{succs: node} \times \text{graph} \to \text{sequ of node}
\]

\[
\text{succs}(n, g) =_{\text{def}} \text{ « sequence of successors of } n \text{ in } g ».
\]

With these prerequisites our problem may be formalized as follows:

\[
\text{reachables}(N, G) \text{ where}
\]

\[
\text{reachables: node} \times \text{graph} \to \text{set of node}
\]

\[
\text{reachables}(n, g) =_{\text{def}} \{ m : \text{node} \parallel \text{reachable}(m, n, g) \}
\]
where

\[
\text{reachable: } \text{node} \times \text{node} \times \text{graph} \rightarrow \text{bool}
\]
\[
\text{reachable}(m, n, g) =_{\text{def}} m = n \lor \exists t: \text{sequ of node} \parallel t \neq \emptyset \land (\text{hd}t = n \land \text{lst}t = m \land \text{connected}(t, g))
\]

and

\[
\text{connected: } \text{nsequ} \times \text{graph} \rightarrow \text{bool}
\]
\[
\text{connected}(t, g) =_{\text{def}} \forall (i: \text{nat} \parallel 1 \leq i < |t|) \parallel t[i+1] \in \text{succs}(t[i], g).
\]

The result of rule application is

\[
\text{r}(N, G) \text{ where}
\]
\[
\text{r: } \text{node} \times \text{graph} \rightarrow \text{set of node}
\]
\[
\text{r}(n, g) =_{\text{def}} r(n, g, \emptyset)
\]
\[
\text{r': } \text{node} \times \text{graph} \times \text{set of node} \rightarrow \text{set of node}
\]
\[
\text{r'}(n, g, s) =_{\text{if suc}} \text{suc}(n, g) = \emptyset \text{ then } s \cup \{n\} \text{ else r''(1, n, g, s \cup \{n\}) fi}
\]
\[
\text{r'': } \text{nat} \times \text{node} \times \text{graph} \times \text{set of node} \rightarrow \text{set of node}
\]
\[
\text{paths''}(k, n, g, s) =_{\text{def}}
\]
\[
\text{if } k > |\text{suc}(n, g)| \text{ then } s
\]
\[
\text{else } \neg\text{visited}(g, \text{suc}(n, g)[k]) \text{ then } r''(k+1, n, g, r'(\text{suc}(n, g)[k], \text{vis}(n, g), s))
\]
\[
\text{else r''(k+1, n, g, s) fi}
\]

where

\[
\text{visited: } \text{graph} \times \text{node} \rightarrow \text{bool}
\]
\[
\text{visited}(g, n) =_{\text{def}} \langle n \text{ marked as visited in } g \rangle
\]

and

\[
\text{vis: } \text{node} \times \text{graph} \rightarrow \text{graph}
\]
\[
\text{vis}(n, g) =_{\text{def}} \langle g \text{ with } n \text{ marked as visited} \rangle.
\]

3.5 Variant 5: A special instance of the rule

3.5.1 The rule

Differently to the previous variants, we now consider a variant that is obtained by specializing the general rule by

- defining \(n(u, v) =_{\text{def}} 1\) and \(\text{next}(k, u, v) =_{\text{def}} k+1\) and instantiating these definitions
- simplifying the resulting expressions, and
skipping the dependencies (on $i$) of the expressions $K$, $R$, and $B$.

Thus, we get the following specialization of our general rule:

$$f(X) \text{ where}$$

$$f(x) = \text{def} \{ r(X, y): \text{n} \parallel Q(x, y) \}$$

(0), (1), (2), (3) as in general rule

(4) $P(X, u, v) \Delta \neg T(u, v) = \text{true}$ \[ \{ r(X, y) \parallel Q(u, v, y) \} = \{ r(X, y) \parallel B(u, v) \Delta Q(K(u, v), R(u, v), y) \} \]

(5) $P(X, u, v) \Delta \neg T(u, v) \Delta B(u, v) = \text{true}$ \[ P(X, K(u, v), R(u, v)) = \text{true} \]

(6) $B(u, v) = \text{true}$ \[ (R(u, v) < v) = \text{true} \text{ where } \text{WF-ORD}(r, <) \text{ or } (K(u, v) < u) = \text{true} \text{ where } \text{WF-ORD}(m, <) \]

$g(X) \text{ where}$

$g(x) = \text{def} \ g'(x, E, \emptyset)$

$g'(u, v, s) = \text{def} \ \text{if } T(u, v) \text{ then } s \cup H(u, v) \text{ else } g''(1, u, v, s) \text{ fi}$

$g''(k, u, v, s) = \text{def} \ \text{if } k > 1 \text{ then } s \text{ else } B(u, v) \text{ then } g''(k+1, u, v, g'(K(u, v), R(u, v), s)) \text{ else } g''(k+1, u, v, s) \text{ fi}$

**Syntactic constraints**

as in general rule

A much simpler form of the output scheme may be obtained by further transformation of the output scheme:

(a) $g''(2, u, v, s)$

$$= [\text{instantiation} ] s$$

(b) $g''(1, u, v, s)$

$$= [\text{unfold } g'' ] \text{ if } 1 > 1 \text{ then } s \text{ else } B(u, v) \text{ then } g''(2, u, v, g'(K(u, v), R(u, v), s)) \text{ else } g''(2, u, v, s) \text{ fi}$$

$$= [\text{simplification of conditional} ] \text{ if } B(u, v) \text{ then } g''(2, u, v, g'(K(u, v), R(u, v), s)) \text{ else } g''(2, u, v, s) \text{ fi}$$

$$= [ (a) ]$$
if $B(u, v)$ then $g'(K(u, v), R(u, v), s)$ else $s$ fi

(c) $g'(u, v, s)$

$\equiv [\text{unfold } g']$
if $T(u, v)$ then $s \cup H(u, v)$ else $g''(1, u, v, s)$ fi

$\equiv [\text{(b)}]$
if $T(u, v)$ then $s \cup H(u, v)$
else $B(u, v)$ then $g'(K(u, v), R(u, v), s)$ else $s$ fi

(d) Define $g''(u, v) = \text{def } g'(u, v, \emptyset)$. Then

$g''(u, v)$

$\equiv [\text{unfold } g'']$
$g'(u, v, \emptyset)$

$\equiv [\text{unfold } g' \text{ as obtained in (b)}]$
if $T(u, v)$ then $\emptyset \cup H(u, v)$
else $B(u, v)$ then $g'(K(u, v), R(u, v), \emptyset)$ else $\emptyset$ fi

$\equiv [\text{simplification; fold } g'']$
if $T(u, v)$ then $H(u, v)$
else $B(u, v)$ then $g''(K(u, v), R(u, v))$ else $\emptyset$ fi.

Altogether, we have derived as a new output scheme:

$g(X)$ where

$g: \text{m } \rightarrow \text{ set of n}$

$g(x) = \text{def } g''(x, E)$ where

$g'': (u: \text{m } \times v: r \parallel P(X, u, v)) \rightarrow \text{ set of n}$

$g''(u, v) = \text{def}$
if $T(u, v)$ then $H(u, v)$
else $B(u, v)$ then $g''(K(u, v), R(u, v))$ else $\emptyset$ fi

3.5.2 An application: Prime factors of a natural number (\(\geq 2\))

Given a natural number $n$ (\(\geq 2\)), it is asked to compute the sequence of all prime factors of $n$ in increasing order.

The output type may be formally specified by

$p\text{seuq } = \text{def } (s: \text{nats}equ \parallel \forall \ 1 \leq i \leq |s| \parallel \text{prime}(s[i]))$

where
prime: nat \to bool

\text{prime}(x) \equiv \text{"x is a prime number".}

Using, additionally,

\text{prod: psequ \to nat}

\text{prod}(sp) \equiv \text{if } sp = [] \text{ then } 1 \text{ else hd}(sp) \times \text{prod}(\text{tl}(sp)) \text{ fi}

and

\text{ordered: psequ \to bool}

\text{ordered}(sp) \equiv \text{"sp is increasingly ordered"}

a complete formal specification of the problem is given by

\text{all-primes}(N) \text{ where}

\text{all-primes: nat \to set of psequ}

\text{all-primes}(n) \equiv \{ sp : \text{psequ} \mid \text{prod}(sp) = n \land \text{ordered}(sp) \}.

The result of rule application is

\text{primes}(N) \text{ where}

\text{primes: nat \to set of psequ}

\text{primes}(n) \equiv \text{prs}(n, [], 2)

\text{prs: (nat \times psequ \times nat) \to set of psequ}

\text{prs}(n, s, c) \equiv \text{if } n = 1 \text{ then } \{ s \}

\text{elsif } \text{true then } \text{prs} \text{ if } c \mid n \text{ then } n \div c \text{ else } n \text{ fi, if } c \mid n \text{ then } (s+c, c) \text{ else } (s, \text{next}(c)) \text{ fi}

\text{else } \emptyset \text{ fi}

\text{next: nat \to nat}

\text{next}(c) \equiv \text{if } c=2 \text{ then } 3 \text{ else } c+2 \text{ fi}

where \text{prs} can be further transformed (using simplification of conditional; distributivity of function call and conditional; simplification) into

\text{prs}(n, s, c) \equiv \text{if } n = 1 \text{ then } \{ s \}

\text{elsif } c \mid n \text{ then } \text{prs}(n \div c, s+c, c)

\text{else } \text{prs}(n, s, \text{next}(c)) \text{ fi}
3.5.3 Another application: all primes up to a given natural number

Given a natural number \( n \geq 1 \), it is asked to compute the set of all primes less or equal to \( n \).

Using

\[
\text{prime: } \text{nat} \rightarrow \text{bool} \\
\text{prime}(x) \equiv \text{« } x \text{ is a prime number »}
\]

the problem is obviously formally specified by

\[
\text{all-primes}(N) \text{ where}
\]

\[
\text{all-primes: } \text{nat} \rightarrow \text{set of nat} \\
\text{all-primes}(n) \equiv \{ p: \text{nat} \mid 1 \leq p \leq n \land \text{prime}(p) \}.
\]

The result of rule application is

\[
\text{primes}(N) \text{ where}
\]

\[
\text{primes: } \text{nat} \rightarrow \text{set of nat} \\
\text{primes}(n) \equiv \text{def } \text{prs}(n, 1, \emptyset)
\]

\[
\text{prs: } (\text{nat} \times \text{nat} \times \text{set of nat}) \rightarrow \text{set of nat} \\
\text{prs}(n, q, s) \equiv \text{def}
\]

\[
\begin{cases}
\text{if } q = n \text{ then } s \\
\text{else if } \text{true then } \text{prs}(n, \text{if } \text{prime}(q) \text{ then } (q+1, s \cup \{q\}) \text{ else } (q+1, s) \text{ fi }}
\text{else } \emptyset \text{ fi}
\end{cases}
\]

where \( \text{prs} \) can be further transformed (using simplification of conditional; distributivity of function call and conditional; simplification) into

\[
\text{prs}(n, q, s) \equiv \text{def}
\]

\[
\begin{cases}
\text{if } q = n \text{ then } s \\
\text{else if } \text{prime}(q) \text{ then } \text{prs}(n, q+1, s \cup \{q\})
\text{else } \text{prs}(n, q+1, s) \text{ fi}
\end{cases}
\]

3.6 Further variants

Similar to variants 1 to 3 (where the initial specification asked for some expression over a set rather than the set itself), further variants could be given (and proved in an analogous way) to deal with other set expressions (e.g., \textbf{min}, \textbf{max}, or the like).
So far, we also have only considered variants where the alternatives resulting from the splitting of the computation according to applicability condition (4) are considered in an ordered way. Of course, we can also give analogous variants where the union
\[ \bigcup_{i=1}^{n(u,v)} \{ r(X, y) \upharpoonright B(i, u, v) \Delta Q'(K(i, u, v), R(i, u, v), y) \} \]
in condition 4 is replaced by
\[ \bigcup_{x \in m(u,v)} \{ r(X, y) \upharpoonright B(x, u, v) \Delta Q'(K(x, u, v), R(x, u, v), y) \} \]
and by using
\[
\begin{align*}
m(u,v) \setminus \{ x \} & \quad \text{for} \quad \text{next}(k, u,v) \\
\mu \setminus \{ x \} & \quad \text{for} \quad K(x, u, v) \\
\emptyset & \quad \text{for} \quad k > n(u, v).
\end{align*}
\]
Since, however, efficient computation requires an ordering of these alternatives anyhow, we have refrained from giving these variants explicitly.

4. Related work and concluding remarks

As its main contribution, this paper has presented a powerful transformation rule to convert specifications of set-valued functions defined by set comprehension into executable functional implementations that basically show the behaviour of backtrack algorithms. From this central transformation rule we have furthermore derived several variants to cope with related problems and their solution by backtracking.

4.1 Related work

Early papers that dealt with the problem of backtracking from a methodical point of view are [Gerhart, Yelowitz 76] and [Broy, Wirsing 80]. Both papers give kind of functional program schemes to characterize and discuss the idea of backtracking, but do not investigate the problem of how to transform descriptive specifications into these functional schemes.

A treatment of backtracking as a solution to search problems which is fairly close to our approach can be found in the work of Doug Smith (cf., e.g., [Smith 86], [Smith 88], or [Smith 90] – also for further references). He gives various design strategies which achieve roughly the same effect as the application of our rule. Also the entirety of conditions required to apply these strategies is basically comparable to our applicability conditions. The main difference in the two approaches – apart from having a strategy there and a compact rule here – is in various technical aspects. Also, it is not quite clear how the variants we deal with fit into Smith's approach.
Of course, our rules presented in this paper also show many relationships to the rules presented in [Partsch 90]. E.g., the rule "∃-elimination by backtracking II" may be considered a specialized form of variant 2 presented here, and the rule "computing qualifiers for a finite set" is but a particular special case of our central rule. Moreover, a difference in methodically looking at the problem is worth mentioned here. Whereas in [Partsch 90] we primarily concentrate on solutions for the existence problem and present tactics to extend these solutions to also cover the "some" or the "set-of-all" problem, we started here the other way round which considerably reduces the amount of work for concrete applications.

4.2 Concluding remarks

In some sense, the problems our rule is able to deal with, can be viewed as generate-and-test problems to produce sets the elements of which are characterized by some predicate. Although the only real restriction for the application of the rule is provided by the applicability conditions, best use of the rule can be made for those problems where the elements of the solution are "complex" (i.e., not directly computable from the input) and where the search space is large (maybe infinite) and not easily enumerable. For "simpler" problems (not showing these characteristics) application of the rule is well possible, but usually will require additional effort to simplify the result.

It is obvious, that our rule is primarily aimed at languages with the initially mentioned semantic characteristics. For these, the rule offers a possibility to convert non-operational, descriptive specifications into operational ones – maybe as a first step towards further improving efficiency or switching to an imperative implementation. In the context of a non-strict, lazy functional language (such as Gofer), the value of our rule may be doubted, since in such a language our initial specifications may be directly executed. For all our examples, however, corresponding comparisons (with Gofer implementations of the respective program versions) have shown a substantial increase in efficiency (through application of our rule) also for a non-strict, lazy functional language.

The main problem in using the given transformation rules for concrete applications is in finding the appropriate instantiations required by the applicability conditions. Although some guidance (in finding instantiations) is given by the conditions themselves as well as by the remarks in section 1, three major creative activities remain to be done:

- finding a suitable way of splitting the problem into subproblems (condition (4))
- inventing an appropriate invariant P (conditions (2) and (5)), and
- defining a suitable ordering for the proof of termination (condition (6)).

Once a suitable instantiation has been found, the verification of the resulting proof obligations is straightforward and even gives potential for automation using a theorem prover. Experience (when dealing with all the examples given here and in [Achatz, Partsch 96]) has confirmed the assumption that the reverse of this statement also holds: Whenever complicated proof obligations resulted from
some instantiation, it turned out that the real problem was with the instantiation rather than the proof obligation.

References

[Aehatze, Partsch 96]
Aehatz, K., Partsch, H.: A powerful transformation rule, its applications and variants. Faculty for Informatics, University of Ulm, Technical Report 1996 (to appear)


[Bauer et al. 89]

[Boiten et al. 92]

[Broy, Wirsing 90]

[Gerhart, Yelowitz 76]

[Partsch 90]

[Smith 86]

[Smith 88]

[Smith 90]
Appendix

A.1 Permutations (section 2.2)

Instantiations

\[
\begin{align*}
  f & \quad \text{by} \quad \text{perms} \\
  g & \quad \text{by} \quad p \\
  g' & \quad \text{by} \quad p' \\
  g'' & \quad \text{by} \quad p'' \\
  X & \quad \text{by} \quad Q \\
  x & \quad \text{by} \quad q \\
  y & \quad \text{by} \quad sp \\
  u & \quad \text{by} \quad q \\
  v & \quad \text{by} \quad qp \\
  r(X, y) & \quad \text{by} \quad sp \\
  Q(x, y) & \quad \text{by} \quad \text{same-els}(q, sp) \\
  P(X, u, v) & \quad \text{by} \quad \text{same-els}(Q, q++qp) \\
  E & \quad \text{by} \quad [] \\
  Q'(u, v, y) & \quad \text{by} \quad \exists s' : \text{ndsequ} \parallel sp = qp++s' \land \text{same-els}(q, s') \\
  T(u, v) & \quad \text{by} \quad q = [] \\
  H(u, v) & \quad \text{by} \quad \{qp\} \\
  D(u, v) & \quad \text{by} \quad \emptyset \\
  n(u, v) & \quad \text{by} \quad |q| \\
  B(i, u, v) & \quad \text{by} \quad \text{true} \\
  K(i, u, v) & \quad \text{by} \quad q \cdot q[k] \\
  R(i, u, v) & \quad \text{by} \quad qp++q[k] \\
  \text{next}(k, u, v) & \quad \text{by} \quad k+1 \\
  \text{WF-ORD}(m, <) & \quad \text{by} \quad (\text{ndsequ}, a < b \iff |a| < |b|).
\end{align*}
\]

Proof obligations

(0) \((|\{sp : \text{ndsequ} \parallel \text{same-els}(q, sp)\}| < \infty) \equiv \text{true}\)

(1) \(\text{same-els}(Q, Q++[]) \equiv \text{true}\)

(2) \(\text{same-els}(q, sp) \equiv \exists s' : \text{ndsequ} \parallel sp = []++s' \land \text{same-els}(q, s')\)

(3) \(\text{same-els}(Q, q++qp) \Delta q = [] \equiv \text{true} \uparrow\)

\[
\{qp\} \equiv (sp : \text{ndsequ} \parallel \exists s' : \text{ndsequ} \parallel sp = qp++s' \land \text{same-els}(q, s'))
\]
(4) \( \text{same-els}(Q, q++qp) \Delta q \neq [] \equiv \text{true} \uplus \\{ \text{sp}: \text{ndsequ} \parallel \exists s': \text{ndsequ} \parallel \text{sp} = qp++s' \land \text{same-els}(q, s') \} \equiv \bigcup_{i=1..|q|} \{ \text{sp: ndsequ} \parallel \exists s': \text{ndsequ} \parallel \text{sp} = (qp++)_i++s' \land \text{same-els}(q(q[i], s') \}$

(5) \( 1 \leq i \leq |q| \Delta \text{same-els}(Q, q++qp) \Delta q \neq [] \equiv \text{true} \uplus \text{same-els}(Q, (q-q[i])++(qp++)_i) \equiv \text{true} \)

(6) \( 1 \leq i \leq |q| \equiv \text{true} \uplus |q-q[i]| < |q| \equiv \text{true} \)

#### A.2 Segmentations (section 2.3)

**Instantiations**

- \( f \) by \( \text{all-segs} \)
- \( g \) by \( \text{segs} \)
- \( g' \) by \( \text{segs'} \)
- \( g'' \) by \( \text{segs''} \)
- \( X \) by \( Q \)
- \( x \) by \( q \)
- \( y \) by \( sp \)
- \( u \) by \( q \)
- \( v \) by \( qp \)
- \( r(X, y) \) by \( sp \)
- \( Q(x, y) \) by \( \text{flattens} = q \)
- \( P(X, u, v) \) by \( \text{flattens}(qp++) = Q \)
- \( E \) by \( [] \)
- \( Q'(u, v, y) \) by \( \exists s': \text{ssequ} \parallel \text{flattens}(s') = q \land (sp = qp++s' \lor (s' \neq [] \land sp = qp++\text{lhd} s' ++ \text{tl}s')) \)
- \( T(u, v) \) by \( q = [] \)
- \( H(u, v) \) by \( \{ qp \} \)
- \( D(u, v) \) by \( \emptyset \)
- \( n(u, v) \) by \( 2 \)
- \( B(i, u, v) \) by \( \text{true} \)
- \( K(i, u, v) \) by \( \text{tl} q \)
- \( R(i, u, v) \) by \( \text{if } k=1 \text{ then } qp++[\text{hd} q] \text{ else }qp++[\text{hd} q] \text{ fi} \)
- \( \text{next}(k, u, v) \) by \( k+1 \)
- \( \text{WF-ORD}(m, <) \) by \( \text{msequ}, a < b \iff |a| < |b| \).

**Proof obligations**

(0) \( (|\{ sp: \text{ssequ} \parallel \text{flattens}(sp) = q \}| < \infty) \equiv \text{true} \)
(1) flatten(++) = Q ≡ true

(2) flatten(sp) = q ≡ ∃ s': ssequ || flatten(s') = q ∧ (sp = [] ++ s' ∨ (s' ≠ [] ∆ sp = [] ++ lst hd s' ++ tls'))

(3) flatten(qp++) = Q ∆ q = [] ≡ true ⊩
   \{ qp \} ≡ \{ sp: ssequ || ∃ s': ssequ || flatten(s') = q ∧ 
   (sp = qp++ s' ∨ (s' ≠ [] ∆ sp = qp++ lst hd s' ++ tls')) \}

(4) flatten(qp++) = Q ∆ q ≠ [] ≡ true ⊩
   \{ sp: ssequ || ∃ s': ssequ || flatten(s') = tlq ∧ 
   (sp = if k = 1 then qp++ nil[hdq] else qp++ [[hdq]] fi++ s' ∨ 
   sp = if k = 1 then qp++ nil[hdq] else qp++ [[hdq]] fi++ lst hd s' ++ tls')) \}

(5) 1 ≤ i ≤ 2 ∆ flatten(qp++) = Q ∆ q ≠ [] ≡ true ⊩
    flatten(if k = 1 then qp++ nil[hdq] else qp++ [[hdq]] fi++ tlq) = Q ≡ true

(6) 1 ≤ i ≤ 2 ≡ true ⊩ |tlq| < |ql| ≡ true

A.3 Partitions of a natural number (section 2.4)

Instantiations

f by all-parts

= (all-parts)

g by parts

g' by parts'
g'' by parts''

X by Q

x by q

y by sp

u by q

v by qp

r(X, y) by sp

Q(x, y) by sum(sp) = q

P(X, u, v) by sum(qp)+q = Q

E by [1]

Q'(u, v, y) by ∃ s': psequ || sum(s') = q ∧ 
   (sp = qp++ s' ∨ (s' ≠ [] ∆ sp = qp++ lst hd s' ++ tls'))

T(u, v) by q = 0

H(u, v) by \{ qp \}

D(u, v) by Ø
Proof obligations

(0) \((\{sp: \text{psequ} \mid \text{sum}(sp) = q\} < \infty) \equiv \text{true}\)

(1) \(\text{sum}([]) + q = Q \equiv \text{true}\)

(2) \(\text{sum}(sp) = q \equiv \exists s': \text{psequ} \mid \text{sum}(s') = q \land (sp = [1]+s' \lor (s' \neq [1] \Delta sp = [1]+\text{hds}s'+\text{tl}s'))\)

(3) \(\text{sum}(qp)+q = Q \Delta q = 0 \equiv \text{true} +
\{qp\} = \{sp: \text{psequ} \mid \exists s': \text{psequ} \mid \text{sum}(s') = q \land (sp = qp++s' \lor (s' \neq [1] \Delta sp = qp+\text{hds}s'+\text{tl}s'))\}

(4) \(\text{sum}(qp)+q = Q \Delta q \neq 0 \equiv \text{true} +
\{sp: \text{psequ} \mid \exists s': \text{psequ} \mid \text{sum}(s') = q-1 \land
\quad (sp = \text{if } k=1 \text{ then } qp+11 \text{ else } qp++[1] \text{ fi}++s' \lor
\quad sp = \text{if } k=1 \text{ then } qp+11 \text{ else } qp++[1] \text{ fi}+\text{hds}s'+\text{tl}s')\}

(5) \(1 \leq i \leq 2 \Delta \text{sum}(qp)+q = Q \Delta q = 0 \equiv \text{true} +
\text{sum}(\text{if } k=1 \text{ then } qp+11 \text{ else } qp++[1] \text{ fi})+(q-1) = Q \equiv \text{true}\)

(6) \(1 \leq i \leq 2 \equiv \text{true} + q-1 < q \equiv \text{true}\)

A.4 All subsequences (section 2.4)

Instantiations

\(f\) by \(\text{all-sub}
\(g\) by \(\text{sub}
\(g'\) by \(\text{sub'}
\(g''\) by \(\text{sub''}
\(X\) by \(\text{Q}
\(x\) by \(q
\(y\) by \(q'
\(u\) by \(q
\(v\) by \(p
\(r(X, y)\) by \(q'\)
Q(x, y) by issusbssequ(q', q)
P(X, u, v) by \exists p': msequ \parallel \text{issusbssequ}(p, p') \land p'++q = Q
E by \[
Q'(u, v, y) by \exists s': msequ \parallel q' = p++s' \land \text{issusbssequ}(s', q)
T(u, v) by q = \[
H(u, v) by \{p\}
D(u, v) by \emptyset
n(u, v) by 2
B(i, u, v) by true
K(i, u, v) by tlq
R(i, u, v) by if k=1 then p else p++[hdq] fi
next(k, u, v) by k+1
WF-ORD(m, <) by (msequ, a < b \iff lal < lbl).

Proof obligations

(0) (l\{q': msequ \parallel \text{issusbssequ}(q', q)\}l < \infty) = true

(1) \exists p': msequ \parallel \text{issusbssequ}([], p') \land p'++Q = Q \equiv true

(2) \text{issusbssequ}(q', q) \equiv \exists s': msequ \parallel q' = []++s' \land \text{issusbssequ}(s', q)

(3) (\exists p': msequ \parallel \text{issusbssequ}(p, p') \land p'++q = Q) \Delta q = [] \equiv true \triangleright
   \{p\} \equiv \{q': msequ \parallel \exists s': msequ \parallel q' = p++s' \land \text{issusbssequ}(s', q)\}

(4) (\exists p': msequ \parallel \text{issusbssequ}(p, p') \land p'++q = Q) \Delta q \neq [] \equiv true \triangleright
   \{q': msequ \parallel \exists s': msequ \parallel q' = if k=1 then p else p++[hdq] fi++s' \land
   \text{issusbssequ}(s', tlq)\} \equiv
   \bigcup_{i=1..2} \{q': \text{sequ} \parallel \exists s': msequ \parallel q' = if k=1 then p else p++[hdq] fi++s' \land
   \text{issusbssequ}(s', tlq)\}

(5) 1 \leq i \leq 2 \Delta (\exists p': msequ \parallel \text{issusbssequ}(p, p') \land p'++q = Q) \Delta q \neq [] \equiv true \triangleright
   (\exists p': msequ \parallel \text{issusbssequ}(if k=1 then p else p++[hdq] fi, p') \land p'++tlq = Q) \equiv true

(6) 1 \leq i \leq 2 \equiv true \triangleright |tlq| < |lq| \equiv true

A.5 The Pack-Problem (section 2.6)

Instantiations

f by all-packs
g by ap
g' by ap'
g'' by ap''
\[ X \quad \text{by} \quad (K, B) \]
\[ x \quad \text{by} \quad (k, b) \]
\[ y \quad \text{by} \quad A \]
\[ u \quad \text{by} \quad (k, b) \]
\[ v \quad \text{by} \quad a \]
\[ r(X, y) \quad \text{by} \quad A \]
\[ Q(x, y) \quad \text{by} \quad \text{legal}(A, k, b) \]
\[ P(X, u, v) \quad \text{by} \quad \text{legal}(a, \text{dom}(a), B) \land \text{dom}(a)++k = K \]
\[ E \quad \text{by} \quad \emptyset \]
\[ Q'(u, v, y) \quad \text{by} \quad \exists a': \text{corr} \quad \text{il} A = a \cdot a' \land \text{legal}(a', k, b) \]
\[ T(u, v) \quad \text{by} \quad k = [] \]
\[ H(u, v) \quad \text{by} \quad [a] \]
\[ D(u, v) \quad \text{by} \quad \emptyset \]
\[ n(u, v) \quad \text{by} \quad |b| \]
\[ B(i, u, v) \quad \text{by} \quad \text{fits}(\text{hd}k, b[i]) \]
\[ K(i, u, v) \quad \text{by} \quad (\text{tlk}, b[i \Theta \text{hd}k]) \]
\[ R(i, u, v) \quad \text{by} \quad a[\text{hd}k] \leftarrow b[i] \]
\[ \text{next}(k, u, v) \quad \text{by} \quad k+1 \]
\[ \text{WF-ORD}(r, <) \quad \text{by} \quad (\text{blocks}, a < b \Leftrightarrow |a| < |b|) \]

**Proof obligations**

0. \((|A: \text{corr} \quad \text{il} \text{legal}(A, k, b)| < \infty) \equiv \text{true} \)

1. \((\text{legal}(\emptyset, \text{dom}(\emptyset), B) \land \text{dom}(\emptyset)++K = K) \equiv \text{true} \)

2. \((\text{legal}(A, k, b) \equiv \exists a': \text{corr} \quad \text{il} A = \emptyset \cdot a' \land \text{legal}(a', k, b) \)

3. \((\text{legal}(a, \text{dom}(a), B) \land \text{dom}(a)++k = K) \Delta k = [] \equiv \text{true} \quad (a) \equiv \{A: \text{corr} \quad \exists a': \text{corr} \quad \text{il} A = a \cdot a' \land \text{legal}(a', k, b)\}

4. \((\text{legal}(a, \text{dom}(a), B) \land \text{dom}(a)++k = K) \Delta k \neq [] \equiv \text{true} \quad (A: \text{corr} \quad \exists a': \text{corr} \quad \text{il} A = a \cdot a' \land \text{legal}(a', k, b)) \equiv \bigcup_{i=1..n} \{A: \text{corr} \quad \text{il} \text{.fits}(\text{hd}k, b[i]) \Delta \exists a': \text{corr} \quad \text{il} A = (a[\text{hd}k] \leftarrow b[i]) \cdot a' \land \text{legal}(a', \text{tlk}, b[i \Theta \text{tlk}])\}

5. \(1 \leq i \leq |b| \Delta (\text{legal}(a, \text{dom}(a), B) \land \text{dom}(a)++k = K) \Delta k \neq [] \Delta \text{fits}(\text{hd}k, b[i]) \equiv \text{true} \quad (\text{legal}(a[\text{hd}k] \leftarrow b[i]), \text{dom}(a[\text{hd}k] \leftarrow b[i]), B) \land \text{dom}(a[\text{hd}k] \leftarrow b[i])++\text{tlk} = K) \equiv \text{true} \)

6. \(1 \leq i \leq |b| \Delta \text{fits}(\text{hd}k, b[i]) \equiv \text{true} \quad |\text{tlk}| < |k| \equiv \text{true} \)

- 48 -
A.6 Topological Sortings (section 2.7)

Instantiations

\[
\begin{align*}
f & \quad \text{by} \quad \text{topsorts} \\
g & \quad \text{by} \quad ts \\
g' & \quad \text{by} \quad ts' \\
g'' & \quad \text{by} \quad ts'' \\
X & \quad \text{by} \quad (M, PO) \\
x & \quad \text{by} \quad (m, po) \\
y & \quad \text{by} \quad t \\
u & \quad \text{by} \quad (m, po) \\
v & \quad \text{by} \quad to \\
r(X, y) & \quad \text{by} \quad t \\
Q(x, y) & \quad \text{by} \quad \text{same-els}(t, m) \land \text{respects}(t, po) \\
P(X, u, v) & \quad \text{by} \quad \text{same-els}(M, to++m) \\
E & \quad \text{by} \quad [] \\
Q'(u, v, y) & \quad \exists s': \text{ndsequ} \sqcap t = to++s' \land \text{same-els}(s', m) \land \text{respects}(s', po) \\
T(u, v) & \quad \text{by} \quad m = [] \\
H(u, v) & \quad \text{by} \quad \{to\} \\
D(u, v) & \quad \text{by} \quad \emptyset \\
n(u, v) & \quad \text{by} \quad \text{linl} \\
B(i, u, v) & \quad \text{by} \quad \text{ismin}(m[i], po) \\
K(i, u, v) & \quad \text{by} \quad (m-m[i], \text{rem}(m[i], po)) \\
R(i, u, v) & \quad \text{by} \quad to++m[i] \\
next(k, u, v) & \quad \text{by} \quad k+1 \\
\text{WF-ORD}(m, \prec) & \quad \text{by} \quad (\text{ndsequ}, a < b \iff |a| < |b|)
\end{align*}
\]

Proof obligations

(0) \( \{t: \text{ndsequ} \land \text{same-els}(t, m) \land \text{respects}(t, po)\} \lessdot \infty \equiv \text{true} \)

(1) \( \text{same-els}(M, []) \equiv \text{true} \)

(2) \( \exists s': \text{ndsequ} \land t = [] \land \text{same-els}(s', m) \land \text{respects}(s', po) \equiv \)

(3) \( \text{same-els}(M, to++m) \Delta m = [] \equiv \text{true} \)

\( \{to\} \equiv \{t: \text{ndsequ} \land \exists s': \text{ndsequ} \land t = to++s' \land \text{same-els}(s', m) \land \text{respects}(s', po)\} \)
(4) \( \text{same-els}(M, to++m) \Delta m \neq [] \equiv \text{true} \uparrow \)
\[ \{ t \:: \text{ndsequ} \uparrow \exists s' \cdot \text{ndsequ} \uparrow t = to++s' \land \text{same-els}(s', m) \land \text{respects}(s', po) \} \equiv \bigcup_{i = 1 \ldots |m|} \{ t \:: \text{ndsequ} \uparrow \\ \text{ismin}(m[i], po) \Delta \exists s' \cdot \text{ndsequ} \uparrow t = to++m[i]++s' \land \\ \text{same-els}(s', m-m[i]) \land \text{respects}(s', \text{rem}(m[k], po)) \} \]

(5) \( 1 \leq i \leq b ml \Delta \text{same-els}(M, to++m) \Delta m \neq [] \Delta \text{ismin}(m[i], po) \equiv \text{true} \uparrow \)
\( \text{same-els}(M, (to++m[i])++(m-m[i])) \equiv \text{true} \)

(6) \( 1 \leq i \leq |m| \Delta \text{ismin}(m[i], po) \equiv \text{true} \uparrow |m - m[i]| < b ml \equiv \text{true} \)

A.7 Maximal paths in a graph (section 2.8)

Instantiations

\begin{align*}
f & \quad \text{by} \quad \text{all-paths} \\
g & \quad \text{by} \quad \text{paths} \\
g' & \quad \text{by} \quad \text{paths'} \\
g'' & \quad \text{by} \quad \text{paths"} \\
X & \quad \text{by} \quad (N, G) \\
x & \quad \text{by} \quad (n, g) \\
y & \quad \text{by} \quad t \\
u & \quad \text{by} \quad (n, g) \\
v & \quad \text{by} \quad ns \\
r(X, y) & \quad \text{by} \quad t \\
Q(x, y) & \quad \text{by} \quad \text{ismaximal}(t, n, g) \\
P(X, u, v) & \quad \text{by} \quad \text{hdns} = n \land \text{connected}(ns, g) \land \text{acyclic}(ns) \\
E & \quad \text{by} \quad [n] \\
Q'(u, v, y) & \quad \exists s' \cdot \text{nsequ} \uparrow t = ns++s' \land \text{ismaximal}(t, n, g) \\
T(u, v) & \quad \text{by} \quad \text{isend}(ns, g) \\
H(u, v) & \quad \text{by} \quad \{ ns \} \\
D(u, v) & \quad \text{by} \quad \emptyset \\
n(u, v) & \quad \text{by} \quad |\text{succs}(\text{lstns}, g)| \\
B(i, u, v) & \quad \text{by} \quad \text{succs}(\text{lstns}, g)[i] \notin ns \\
K(i, u, v) & \quad \text{by} \quad (n, g) \\
R(i, u, v) & \quad \text{by} \quad ns++\text{succs}(\text{lstns}, g)[i] \\
next(k, u, v) & \quad \text{by} \quad k+1 \\
WF-ORD(m, <) & \quad \text{by} \quad (\text{nsequ}, a < b \iff \text{nodes}(g)-lal < \text{nodes}(g)-lbl) \)
\end{align*}

Proof obligations

(0) \( (|\{ t \:: \text{nsequ} \uparrow \text{ismaximal}(t, n, g) \}| < \infty) \equiv \text{true} \)
(1) \( \text{hd}[N] = N \land \text{connected}(N, G) \land \text{acyclic}(N) \equiv \text{true} \)

(2) \( \text{ismaximal}(t, n, g) \equiv \exists s'. \text{nsseq} u \parallel t = [n]++s' \land \text{ismaximal}(t, n, g) \)

(3) \( (\text{hdns} = n \land \text{connected}(ns, g) \land \text{acyclic}(ns)) \land \text{isend}(ns, g) \equiv \text{true} \uparrow \{ns\} \equiv \{t: \text{nsseq} u \parallel t = ns++s' \land \text{ismaximal}(t, n, g)\} \)

(4) \( (\text{hdns} = n \land \text{connected}(ns, g) \land \text{acyclic}(ns)) \land \neg \text{isend}(ns, g) \equiv \text{true} \uparrow \{t: \text{nsseq} u \parallel \exists s': \text{nsseq} u \parallel t = ns++s' \land \text{ismaximal}(t, n, g)\} \subseteq \bigcup_{i=1..\text{lml}} \{t: \text{nsseq} u \parallel \text{succs}(\text{lstns}, g)[i] \notin ns \Delta \exists s': \text{nsseq} u \parallel t = ns++\text{succs}(\text{lstns}, g)[i]++s' \land \text{ismaximal}(t, n, g)\} \)

(5) \( 1 \leq i \leq \text{lsuccess}(\text{lstns}, g) \Delta (\text{hdns} = n \land \text{connected}(ns, g) \land \text{acyclic}(ns)) \Delta \neg \text{isend}(ns, g) \Delta \text{success}(\text{lstns}, g)[k] \notin ns \equiv \text{true} \uparrow \text{hdns} = n \land \text{connected}(ns++\text{success}(\text{lstns}, g)[i], g) \land \text{acyclic}(ns++\text{success}(\text{lstns}, g)[i]) \equiv \text{true} \)

(6) \( 1 \leq i \leq \text{lsuccess}(\text{lstns}, g) \Delta \text{success}(\text{lstns}, g)[k] \notin ns \equiv \text{true} \uparrow \text{nodes}(g)\rightarrow \text{lns}++\text{success}(\text{lstns}, g)[i] < \text{nodes}(g)\rightarrow \text{lnsl} \equiv \text{true} \)

A.8 A Prolog Interpreter (section 2.9)

Instantiations

\[
\begin{array}{lll}
\text{f} & \text{by} & \text{interprete} \\
\text{g} & \text{by} & \text{int} \\
\text{g}' & \text{by} & \text{int}' \\
\text{g}'' & \text{by} & \text{int}'' \\
\text{X} & \text{by} & (P, A) \\
x & \text{by} & (p, gl) \\
y & \text{by} & \Theta \\
u & \text{by} & (p, gl) \\
v & \text{by} & \Phi \\
r(X, y) & \text{by} & \Theta \downarrow \text{vars}(A) \\
Q(x, y) & \text{by} & (gl, \Theta) \overset{p}{\Rightarrow} * ([], \Theta) \\
P(X, u, v) & \text{by} & (A, \emptyset) \overset{p}{\Rightarrow} * (gl, \Phi) \\
E & \text{by} & \emptyset \\
Q'(u, v, y) & \text{by} & \exists \Theta': \text{subst} u \parallel \Theta = \Phi \ast \Theta' \land (gl, \emptyset) \overset{p}{\Rightarrow} * ([], \Theta') \\
T(u, v) & \text{by} & gl = [] \\
H(u, v) & \text{by} & \{\Phi \downarrow \text{vars}(A)\} \\
D(u, v) & \text{by} & \emptyset \\
n(u, v) & \text{by} & \text{pl}
\end{array}
\]
B(i, u, v) by $\Phi' \neq \text{fail}$ where $\Phi' = \text{unify}(\text{hdgl}, \text{lhs}(\text{rn}(p[i]), gl))$

K(i, u, v) by $(\text{rhs}(\text{rn}(p[i]), gl)) \rightarrow \text{tlgl}(\Phi')$

R(i, u, v) by $\Phi \cdot \Phi'$

Proof obligations

(0) $(\langle [\Theta]_{\text{vars}(A)} \parallel (gl, \Theta) \rangle \rightarrow^* ([], \Theta)) | \neq \infty = \text{true}$

(1) $((A, \Theta) \rightarrow^* (A, \Theta)) = \text{true}$

(2) $(gl, \Theta) \rightarrow^* ([], \Theta) \equiv \exists \Theta' : \text{subst} \parallel \Theta = \Theta' \land (gl, \Theta) \rightarrow^* ([], \Theta')$

(3) $(A, \Theta) \rightarrow^* (gl, \Theta) \Delta gl = [] \equiv \text{true}$

\[ \{ \Theta | \text{vars}(A) \}\equiv \{ \Theta | \text{vars}(A) \parallel \exists \Theta' : \text{subst} \parallel \Theta = \Theta' \land (gl, \Theta) \rightarrow^* ([], \Theta') \} \]

(4) $(A, \Theta) \rightarrow^* (gl, \Theta) \Delta gl \neq [] = \text{true}$

\[ \{ \Theta | \text{vars}(A) \parallel \exists \Theta' : \text{subst} \parallel \Theta = \Theta' \land (gl, \Theta) \rightarrow^* ([], \Theta') \} =
\]

\[ \bigcup_{i=1}^{\text{lp}} \{ \Theta | \text{vars}(A) \parallel \Phi' \neq \text{fail} \Delta
\]

\[ \exists \Theta' : \text{subst} \parallel \Theta = \Phi \ast \Phi' \ast \Theta' \land ((\text{rhs}(\text{rn}(p[i]), gl)) \rightarrow^* ([], \Theta')) \]

where $\Phi' = \text{unify}(\text{hdgl}, \text{lhs}(\text{rn}(p[i]), gl))$

(5) $1 \leq i \leq \text{lp} \parallel (A, \Theta) \rightarrow^* (gl, \Theta) \Delta gl \neq [] \Delta \Phi' \neq \text{fail}$

where $\Phi' = \text{unify}(\text{hdgl}, \text{lhs}(\text{rn}(p[i]), gl))) \equiv \text{true}$

$(A, \Theta) \rightarrow^* ((\text{rhs}(\text{rn}(p[i]), gl)) \rightarrow^* \text{tlgl}(\Phi') \ast \Phi' \ast \Phi') \equiv \text{true}$

A.9 The Coding Problem (section 3.1.2)

Instantiations

\[ f \] by encodings

\[ g \] by enc

\[ g' \] by enc'

\[ g'' \] by enc''

\[ X \] by $(N, P)$

\[ x \] by $(n, p)$

\[ y \] by a

\[ z \] by a

\[ u \] by $(n, p)$

\[ v \] by e

\[ r(X, y) \] by encode(a, P)

\[ Q(x, y) \] by flatten(a) = n \land \text{codable}(a, p)

\[ P(X, u, v) \] by $\exists z \parallel \text{flatten}(z+n) = N \land e = \text{encode}(z, P)$
Proof obligations

(0) \((\bigcup a \in \text{dec}(n, p) \, \text{encode}(a, p)) < \infty) = \text{true}\)

(1) \(\exists z \parallel \text{flatten}(z++) = N \land \{[]\} = \text{encode}(z, P) \equiv \text{true}\)

(2) \(\text{flatten}(a) = n \land \text{codable}(a, p) \equiv \exists a' \parallel a = [++]a' \land \text{flatten}(a') = n \land \text{codable}(a', p)\)

(3) \(\exists z \parallel \text{flatten}(z++) = N \land e = \text{encode}(z, P) \Delta n = [] \equiv \text{true} \uplus e \equiv \bigcup a \in (a \parallel \forall e' \in e \parallel \exists a' \parallel a = e'++a' \land \text{flatten}(a') = n \land \text{codable}(a', p)) \parallel \text{encode}(a, P)\)

(4) \(\exists z \parallel \text{flatten}(z++) = N \land e = \text{encode}(z, P) \Delta n \neq [] \equiv \text{true} \uplus \bigcup a \in (a \parallel \forall e' \in e \parallel \exists a' \parallel a = e'++a' \land \text{flatten}(a') = n \land \text{codable}(a', p)) \parallel \text{encode}(a, P) \equiv \bigcup i = 1..|p| \bigcup a \in (a \parallel |p[i]|) \cdot n \land \forall e' \in e \supset \text{r}(p[i]) \parallel \exists a' \parallel a = e'++a' \land \text{flatten}(a') = n \cdot |p[i]| \land \text{codable}(a', p)) \parallel \text{encode}(a, P)\)

(5) \(1 \leq i \leq |p| \Delta (\exists z \parallel \text{flatten}(z++) = N \land e = \text{encode}(z, P)) \Delta n \neq [] \Delta l(p[i]) \cdot n \equiv \text{true} \uplus (\exists z \parallel \text{flatten}(z++) = N \land e \supset \text{r}(p[i]) = \text{encode}(z, P)) \equiv \text{true}\)

(6) \(1 \leq i \leq |p| \Delta l(p[i]) \cdot n \equiv \text{true} \uplus |n| \cdot l(p[i]) < |nl| \equiv \text{true}\)

A.10 Top-down recognition (section 3.2.2)

Instantiations

\[f\] by \(\text{recognize}\)

\[g\] by \(\text{rec}\)

\[g'\] by \(\text{rec}'\)

\[g''\] by \(\text{rec}''\)

\[X\] by \((P, W)\)

\[x\] by \((p, w)\)

\[y\] by \(y\)

\[u\] by \((p, w)\)
Proof obligations

(1) \( \exists w' \parallel w'++W = W \wedge [Z] \rightarrow \ast w'++[Z] \equiv true \)

(2) \([Z] \rightarrow \ast y \wedge y = w \equiv [Z] \rightarrow \ast y \wedge y = w\)

(3) \(\exists w' \parallel w'++w = W \wedge [Z] \rightarrow \ast w'++v \Delta (w = [] \wedge v = []) \equiv true \)

(4) \(\exists w' \parallel w'++w = W \wedge [Z] \rightarrow \ast w'++v \Delta \neg(w = [] \wedge v = []) \equiv true \)

(5) \(1 \leq i \leq |p|+1 \Delta (\exists w' \parallel w'++w = W \wedge [Z] \rightarrow \ast w'++v) \Delta \neg(w = [] \wedge v = []) \Delta \neg(h \neq [\ast]) \Delta \neg(h \neq [\ast]) \Delta hdw = hdv \equiv true \)

A.11 Shift-Reduce parsing (section 3.3.2)

Instantiations

\[ f \quad \text{by} \quad \text{parse} \]
\[ g \quad \text{by} \quad \text{par} \]
\[ g' \quad \text{by} \quad \text{par}' \]
\[ g'' \quad \text{by} \quad \text{par}'' \]
\( X \) by \((P, [], W)\)
\( x \) by \((p, st, w)\)
\( y \) by \(y\)
\( u \) by \((p, st, w)\)
\( v \) by \(sr\)
\( Q(x, y) \) by \(apply(y, st, w) = ([Z], [])\)
\( P(X, u, v) \) by \(apply(sr, [], W) = (st, w)\)
\( E \) by \([]\)
\( Q'(u, v, y) \) by \(\exists y'. \forall s' \forall st = s'++y' = y \land apply(y', st, w) = ([Z], [])\)
\( T(u, v) \) by \(w = [] \land st = [Z]\)
\( H(u, v) \) by \(sr\)
\( n(u, v) \) by \(|pl| + 1\)
\( B(i, u, v) \) by \((i \leq |pl| \land \exists s' \forall st = s'++rhs(p[i]) \lor (i = |pl| + 1 \land w \neq [])\)
\( K(i, u, v) \) by \(if i \leq |pl| then (s'++lhs(p[i]), w) else (st++hdw, tlw) fi\)
\( R(i, u, v) \) by \(if i \leq |pl| then sr++R(p) else sr++S fi\)
\( \text{next}(k, u, v) \) by \(k+1\)

Proof obligations

1. \(apply([], [], W) = ([], W) \equiv true\)

2. \(apply(y, st, w) = ([Z], []) \equiv \exists y'. \forall s' \forall y'++y' = y \land apply(y', st, w) = ([Z], [])\)

3. \(apply(sr, [], W) = (st, w)) \Delta \neg (w = [] \land st = [Z]) \equiv true \uparrow
\text{sr} \in \{ y : srseq \land \exists y' \forall sr++y' = y \land apply(y', st, w) = ([Z], []) \} \equiv true\)

4. \(apply(sr, [], W) = (st, w))) \Delta (w = [] \land st = [Z]) \equiv true \uparrow
\text{some} y : srseq \land \exists y' \forall sr++y' = y \land apply(y', st, w) = ([Z], []) \equiv true \uparrow
\text{some} y : srseq \land \exists i = 1..|pl| + 1 (i \leq |pl| \land \exists s' \forall st = s'++rhw(p[i]) \lor (i = |pl| + 1 \land w \neq []) \Delta
\exists y' \forall if i \leq |pl| then sr++R(p) else sr++S fi++y' = y \land
apply(y', if i \leq |pl| then (s'++lhs(p[i]), w) else (st++hdw, tlw) fi) = ([Z], [])\)

5. \(1 \leq i \leq |pl| + 1 \Delta (apply(sr, [], W) = (st, w))) \Delta (w = [] \land st = [Z]) \Delta
((i \leq |pl| \land \exists s' \forall st = s'++rhw(p[i]) \lor (i = |pl| + 1 \land w \neq [])) \equiv true \uparrow
apply(if i \leq |pl| then sr++R(p) else sr++S fi, [], W) = if i \leq |pl| then (s'++lhs(p[i]), w) else (st++hdw, tlw) fi \equiv true\)
A.12 Reachable nodes in graph (section 3.4.2)

Instantiations

\[ f \] by \( \text{reachables} \)
\[ g \] by \( r \)
\[ g' \] by \( r' \)
\[ g'' \] by \( r'' \)
\[ X \] by \( (N, G) \)
\[ x \] by \( (n, g) \)
\[ y \] by \( m \)
\[ u \] by \( (n, g) \)
\[ r(X, y) \] by \( m \)
\[ Q(x, y) \] by \( \text{reachable}(m, n, g) \)
\[ T(u) \] by \( \text{succs}(n, g) = [] \)
\[ H(u) \] by \( \{n\} \)
\[ D(u) \] by \( \{n\} \)
\[ n(u) \] by \( |\text{succs}(n, g)| \)
\[ B(i, u) \] by \( \neg \text{visited}(g, \text{succs}(n, g)[k]) \)
\[ K(i, u) \] by \( \{\text{succs}(n, g)[k], \text{vis}(n, g)\} \)
\[ \text{next}(k, u) \] by \( k+1 \)
\[ \text{WF-ORD}(m, <) \] by \( (\text{graph}, g < h \iff \{a: \text{node} \parallel a \in \text{nodes}(g) \land \text{visited}(a, g)\} > \{a: \text{node} \parallel a \in \text{nodes}(h) \land \text{visited}(a, h)\} \)

Proof obligations

(0) \( (\{m: \text{node} \parallel \text{reachable}(m, n, g)\} < \infty) \equiv \text{true} \)
(3) \( \text{succs}(n, g) = [] \equiv \text{true} \)
(4) \( \text{succs}(n, g) \neq [] \equiv \text{true} \)
(6) \( 1 \leq i \leq |\text{succs}(n, g)| \Delta \neg \text{visited}(g, \text{succs}(n, g)[i]) \equiv \text{true} \)

\[ |\text{vis}(n, g)| > |g| \equiv \text{true} \]
A.13 Prime factors of a natural number (section 3.5.2)

Instantiations

\[
\begin{align*}
f & \quad \text{by} \quad \text{all-primes} \\
g & \quad \text{by} \quad \text{primes} \\
g'' & \quad \text{by} \quad \text{prs} \\
X & \quad \text{by} \quad N \\
x & \quad \text{by} \quad n \\
y & \quad \text{by} \quad sp \\
u & \quad \text{by} \quad n \\
v & \quad \text{by} \quad (s, c) \\
r(X, y) & \quad \text{by} \quad sp \\
Q(x, y) & \quad \text{by} \quad \text{prod}(sp) = n \land \text{ordered}(sp) \\
P(X, u, v) & \quad \text{by} \quad \forall (c' :: \text{nat} \mid 2 \leq c' < c) \parallel \neg (c' \mid n) \land \text{prod}(s)^* n = N \land \text{ordered}(s) \\
E & \quad \text{by} \quad ([1], 1) \\
Q'(u, v, y) & \quad \text{by} \quad \exists s' :: \text{psequ} \parallel sp = s++s' \land \text{prod}(s') = n \land \text{ordered}(sp) \\
T(u, v) & \quad \text{by} \quad n = 1 \\
H(u, v) & \quad \text{by} \quad s \\
B(u, v) & \quad \text{by} \quad \text{true} \\
K(u, v) & \quad \text{by} \quad \text{if } c \mid n \text{ then } n \text{ div } c \text{ else } n \text{ fi} \\
R(u, v) & \quad \text{by} \quad \text{if } c \mid n \text{ then } (s++c, c) \text{ else } (s, \text{next}(c)) \text{ fi} \\
\text{WF-ORD}(m, <) & \quad \text{by} \quad ((\text{psequ}, \text{nat}), (a, b) < (c, d) \iff (|a| > |b|) \lor (|a| = |b| \land c > d))
\end{align*}
\]

Proof obligations

(0) \(([[sp :: \text{psequ} \parallel \text{prod}(sp) = n \land \text{ordered}(sp)] < \infty) \equiv \text{true}\)

(1) \((\forall (c' :: \text{nat} \mid 2 \leq c' < 2) \parallel \neg (c' \mid n) \land \text{prod}([])^* N = N \land \text{ordered}([])) = \text{true}\)

(2) \(\text{prod}(sp) = n \land \text{ordered}(sp) \equiv \exists s' :: \text{psequ} \parallel sp = [[]+s' \land \text{prod}(s') = n \land \text{ordered}(sp)]\)

(3) \((\forall (c' :: \text{nat} \mid 2 \leq c' < c) \parallel \neg (c' \mid n) \land \text{prod}(s)^* n = N \land \text{ordered}(s)) \Delta n = 1 \equiv \text{true} \uplus \exists s' :: \text{psequ} \parallel sp = s++s' \land \text{prod}(s') = n \land \text{ordered}(sp)\)

(4) \((\forall (c' :: \text{nat} \mid 2 \leq c' < c) \parallel \neg (c' \mid n) \land \text{prod}(s)^* n = N \land \text{ordered}(s)) \Delta n \neq 1 \equiv \text{true} \uplus \left\{\begin{array}{l}
\{sp :: \text{psequ} \parallel \exists s' :: \text{psequ} \parallel sp = s++s' \land \text{prod}(s') = n \land \text{ordered}(sp)\}
\end{array}\right\}
\]

\[\text{prod}(s') = \text{if } c \mid n \text{ then } n \text{ div } c \text{ else } n \text{ fi \land ordered(sp)}\]

- 57 -
(5) \((\forall \ (c' : \text{nat}) \ | 2 \leq c' < c) \ | \neg(c' \ | n) \land \text{prod}(s)^\ast n = N \land \text{ordered}(s)) \Delta n \neq 1 \equiv \text{true} \uparrow \)

\((\forall \ (c' : \text{nat}) \ | 2 \leq c' < c \ | \begin{cases} \text{if } c \mid n \ \text{then } c \ \text{else } \text{next}(c) \ \text{fi} \ \end{cases} \ | \neg(c' \ | n) \land \text{prod}(\text{if } c \mid n \ \text{then } s++c \ \text{else } s \ \text{fi}) = \text{if } c \mid n \ \text{then } n \ \text{div } c \ \text{else } n \ \text{fi} = N \land \text{ordered}(\text{if } c \mid n \ \text{then } s++c \ \text{else } s \ \text{fi})) \equiv \text{true} \)

(6) \(\begin{cases} \text{if } c \mid n \ \text{then } (s++c, c) \ \text{else } (s, \text{next}(c)) \ \text{fi} < (s, c) \equiv \text{true} \end{cases} \)

A.14 All primes up to a given natural number (section 3.5.3)

Instantiations

\(f \ \text{by } \text{all-primes} \)

\(g \ \text{by } \text{primes} \)

\(g'' \ \text{by } \text{prs} \)

\(X \ \text{by } N \)

\(x \ \text{by } n \)

\(y \ \text{by } p \)

\(u \ \text{by } n \)

\(v \ \text{by } (q, s) \)

\(r(X, y) \ \text{by } p \)

\(Q(x, y) \ \text{by } 1 \leq p \leq n \land \text{prime}(p) \)

\(P(X, u, v) \ \text{by } q \leq n \land s = \{c : \text{nat} \ | 1 \leq c \leq q \land \text{prime}(c)\} \)

\(E \ \text{by } (1, \emptyset) \)

\(Q'(u, v, y) \ \text{by } p \in s \lor (q \leq p \leq n \land \text{prime}(p)) \)

\(T(u, v) \ \text{by } q = n \)

\(H(u, v) \ \text{by } s \)

\(B(u, v) \ \text{by } \text{true} \)

\(K(u, v) \ \text{by } n \)

\(R(u, v) \ \text{by } \begin{cases} \text{if } \text{prime}(q) \ \text{then } (q+1, s \cup \{q\}) \ \text{else } (q+1, s) \ \text{fi} \end{cases} \)

\(\text{WF-ORD}(r, <) \ \text{by } (\text{nat}, a < b \iff b > a) \)

Proof obligations

(0) \(\begin{cases} \text{if } p : \text{nat} \ | 1 \leq p \leq n \land \text{prime}(p)) \ | < \infty \equiv \text{true} \end{cases} \)

(1) \(1 \leq N \land \emptyset = \{c : \text{nat} \ | 1 \leq c \leq 1 \land \text{prime}(c)\} = \text{true} \)

(2) \(1 \leq p \leq n \land \text{prime}(p) \equiv p \in \emptyset \lor (1 \leq p \leq n \land \text{prime}(p)) \)

(3) \(\begin{cases} \text{if } q \leq n \land s = \{c : \text{nat} \ | 1 \leq c \leq q \land \text{prime}(c)\} \ \Delta q = n = \text{true} \uparrow \end{cases} \)

\(s = \{p : \text{nat} \ | p \in s \lor (q \leq p \leq n \land \text{prime}(p))\} \)
(4) \( (q \leq n \land s = \{ c : \text{nat} \parallel 1 \leq c \leq q \land \text{prime}(c) \}) \Delta q \neq n \equiv \text{true} \)
\( (p : \text{nat} \parallel p \in s \lor (q \leq p \leq n \land \text{prime}(p))) \equiv \)
\( (p : \text{nat} \parallel \text{if prime}(q) \text{ then } p \in s \cup \{ q \} \text{ else } p \in s \text{ fi } \lor (q+1 \leq p \leq n \land \text{prime}(p))) \)

(5) \( (q \leq n \land s = \{ c : \text{nat} \parallel 1 \leq c \leq q \land \text{prime}(c) \}) \Delta q \neq n \equiv \text{true} \)
\( (q+1 \leq n \land \text{if prime}(q) \text{ then } s \cup \{ q \} \text{ else } s \text{ fi } = \{ c : \text{nat} \parallel 1 \leq c \leq q+1 \land \text{prime}(c) \}) \equiv \text{true} \)

(6) \( q+1 > q \equiv \text{true} \)
Liste der bisher erschienenen Ulmer Informatik-Berichte
Einige davon sind per FTP von ftp.informatik.uni-ulm.de erhältlich
Die mit * markierten Berichte sind vergriffen

List of technical reports published by the University of Ulm
Some of them are available by FTP from ftp.informatik.uni-ulm.de
Reports marked with * are out of print

91-01 Ker-I Ko, P. Orponen, U. Schöning, O. Watanabe
Instance Complexity

91-02* K. Gluditz, H. Fassbender, H. Vogler
Compiler-Based Implementation of Syntax-Directed Functional Programming

91-03* Alfons Geser
Relative Termination

91-04* J. Köbler, U. Schöning, J. Toran
Graph Isomorphism is low for PP

91-05 Johannes Köbler, Thomas Thierauf
Complexity Restricted Advice Functions

91-06* Uwe Schöning
Recent Highlights in Structural Complexity Theory

91-07* F. Green, J. Köbler, J. Toran
The Power of Middle Bit

91-08* V. Arvind, Y. Han, L. Hemachandra, J. Köbler, A. Lozano,
M. Mundhenk, A. Ogihara, U. Schöning, R. Silvestri, T. Thierauf
Reductions for Sets of Low Information Content

92-01* Vikraman Arvind, Johannes Köbler, Martin Mundhenk
On Bounded Truth-Table and Conjunctive Reductions to Sparse and Tally Sets

92-02* Thomas Noll, Heiko Vogler
Top-down Parsing with Simultaneous Evaluation of Noncircular Attribute Grammars

92-03 Fakultät für Informatik
17. Workshop über Komplexitätstheorie, effiziente Algorithmen und Datenstrukturen

92-04* V. Arvind, J. Köbler, M. Mundhenk
Lowness and the Complexity of Sparse and Tally Descriptions

92-05* Johannes Köbler
Locating P/poly Optimally in the Extended Low Hierarchy

92-06* Armin Kühnemann, Heiko Vogler
Synthesized and inherited functions - a new computational model for syntax-directed semantics

92-07* Heinz Fassbender, Heiko Vogler
A Universal Unification Algorithm Based on Unification-Driven Leftmost Outermost Narrowing
92-08* Uwe Schöning
On Random Reductions from Sparse Sets to Tally Sets

92-09* Hermann von Hassel, Laura Martignon
Consistency in Stochastic Network

92-10 Michael Schmitt
A Slightly Improved Upper Bound on the Size of Weights Sufficient to Represent Any Linearly Separable Boolean Function

92-11 Johannes Köbler, Seinosuke Toda
On the Power of Generalized MOD-Classes

92-12 V. Arvind, J. Köbler, M. Mundhenk
Reliable Reductions, High Sets and Low Sets

92-13 Alfons Geser
On a monotonic semantic path ordering

92-14* Joost Engelfriet, Heiko Vogler
The Translation Power of Top-Down Tree-To-Graph Transducers

93-01 Alfred Lupper, Konrad Froitzheim
AppleTalk Link Access Protocol basierend auf dem Abstract Personal Communications Manager

The COCOON Object Model

93-03 Thomas Thierauf, Seinosuke Toda, Osamu Watanabe
On Sets Bounded Truth-Table Reducible to P-selective Sets

93-04 Jin-Yi Cai, Frederic Green, Thomas Thierauf
On the Correlation of Symmetric Functions

93-05 K. Kuhn, M. Reichert, M. Nathe, T. Beuter, C. Heinlein, P. Dadam
A Conceptual Approach to an Open Hospital Information System

93-06 Klaus Gaßner
Rechnerunterstützung für die konzeptuelle Modellierung

93-07 Ulrich Keßler, Peter Dadam
Towards Customizable, Flexible Storage Structures for Complex Objects

94-01 Michael Schmitt
On the Complexity of Consistency Problems for Neurons with Binary Weights

94-02 Armin Kühnemann, Heiko Vogler
A Pumping Lemma for Output Languages of Attributed Tree Transducers

94-03 Harry Buhrman, Jim Kadin, Thomas Thierauf
On Functions Computable with Nonadaptive Queries to NP

94-04 Heinz Faßbender, Heiko Vogler, Andrea Wedel
Implementation of a Deterministic Partial E-Unification Algorithm for Macro Tree Transducers
94-05  V. Arendt, J. Köhler, R. Schuler
On Helping and Interactive Proof Systems

94-06  Christian Kalus, Peter Dadam
Incorporating record subtyping into a relational data model

94-07  Markus Tresch, Marc H. Scholl
A Classification of Multi-DATABASE Languages

94-08  Friedrich von Henke, Harald Rueß
Arbeitstreffen Typtheorie: Zusammenfassung der Beiträge

Construction and Deduction Methods for the Formal Development of Software

94-10  Axel Dold
Formalisierung schematischer Algorithmen

94-11  Johannes Köhler, Osamu Watanabe
Xeu Collapse Consequences of NP Having Small Circuits

94-12  Rainer Schuler
On Average Polynomial Time

94-13  Rainer Schuler, Osamu Watanabe
Towards Average-Case Complexity Analysis of NP Optimization Problems

94-14  Wolfram Schulte, Ton Vullinghs
Linking Reactive Software to the X-Window System

94-15  Alfred Lupper
Namensverwaltung und Adressierung in Distributed Shared Memory-Systemen

94-16  Robert Regn
Verteilte Unix-Betriebssysteme

94-17  Helmut Partsch
Again on Recognition and Parsing of Context-Free Grammars:
Two Exercises in Transformational Programming

94-18  Helmut Partsch
Transformational Development of Data-Parallel Algorithms: an Example

95-01  Oleg Verbitsky
On the Largest Common Subgraph Problem

95-02  Uwe Schöning
Complexity of Presburger Arithmetic with Fixed Quantifier Dimension

95-03  Harry Buhrman, Thomas Thierauf
The Complexity of Generating and Checking Proofs of Membership

95-04  Rainer Schuler, Tomoyuki Yamakami
Structural Average Case Complexity

95-05  Klaus Achaz, Wolfram Schulte
Architecture Independent Massive Parallelization of Divide-And-Conquer Algorithms
95-06  Christoph Karg, Rainer Schuler  
Structure in Average Case Complexity

95-07  P. Dadam, K. Kuhn, M. Reichert, T. Beuter, M. Nathe  
ADEPT: Ein integrierender Ansatz zur Entwicklung flexibler, zuverlässiger  
kooperierender Assistenzsysteme in klinischen Anwendungsumgebungen

95-08  Jürgen Kehrer, Peter Schulthess  
Aufbereitung von gescannten Röntgenbildern zur filmlosen Diagnostik

95-09  Hans-Jörg Burtschick, Wolfgang Lindner  
On Sets Turing Reducible to P-Selective Sets

95-10  Boris Hartmann  
Berücksichtigung lokaler Randbedingung bei globaler Zieloptimierung mit  
eurowalen Netzen am Beispiel Truck Backer-Upper

95-11  Thomas Beuter, Peter Dadam  
Prinzipien der Replikationskontrolle in verteilten Systemen

95-12  Klaus Achatz, Wolfram Schulte  
Massive Parallelization of Divide-and-Conquer Algorithms over Powerlists

95-13  Andrea Mößle, Heiko Vogler  
Efficient Call-by-value Evaluation Strategy of Primitive Recursive Program  
Schemes

95-14  Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß  
A Generic Specification for Verifying Peephole Optimizations

96-01  Errüment Canver, Jan-Tecker Gayen, Adam Moik  
Formale Entwicklung der Steuerungssoftware für eine elektrisch ortsbetriebene  
Weiche mit VSE

96-02  Bernhard Nebel  
Solving Hard Qualitative Temporal Reasoning Problems: Evaluating the  
Efficiency of Using the ORD-Horn Class

96-03  Ton Vollinghs, Wolfram Schulte, Thilo Schwinn  
An Introduction to TkGofer

96-04  Thomas Beuter, Peter Dadam  
Anwendungsspezifische Anforderungen an  
Workflow-Mangement-Systeme am Beispiel  
der Domäne Concurrent-Engineering

96-05  Gerhard Schellhorn, Wolfgang Ahrendt  
Verification of a Prolog Compiler - First Steps with KIV

96-06  Manindra Agrawal, Thomas Thierauf  
Satisfiability Problems

96-07  Vikraman Arvind, Jacobo Torán  
A nonadaptive NC Checker for Permutation Group Intersection

96-08  David Cyrlak, Oliver Möller, Harald Rueß  
An Efficient Decision Procedure for a Theory of Fix-Sized Bitvectors with  
Composition and Extraction
96-09  Bernd Biechele, Dietmar Ernst, Frank Houdek, Joachim Schmid, Wolfram Schulte
Erfahrungen bei der Modellierung eingebetteter Systeme mit verschiedenen
SA/RT-Ansätzen

96-10  Falk Bartels, Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß
Formalizing Fixed Point Theory in PVS

96-11  Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß
Mechanized Semantics of Simple Imperative Programming Constructs

96-12  Axel Dold, Friedrich W. von Henke, Holger Pfeifer, Harald Rueß
Generic Compilation Schemes for Programming Constructs

96-13  Klaus Achatz, Helmut Partsch
From Descriptive Specifications to Operational ones: A Powerful Transformation
Rule, its Applications and Variants