Formal correctness proofs of functional programs: Dijkstra's algorithm, a case study

Holger Benl and Helmut Schwichtenberg

Draft of March 8, 1999

One can argue that writing proofs might be better than writing programs, for the following simple reason: There is no algorithm that can check whether a program meets its specification, but it is easy to check whether a given proof is correct.

Let us first review some well-known paradigms for connecting proofs and programs (see Bates/Constable [1] for more background).

1. Write the (functional) program as term in an approriate higher type language, and prove its correctness, i.e. that it satisfies its specification.
2. From a *constructive* proof of a formula $\forall x \exists^* y\, B(x, y)$ extract a program.
3. For a *non-constructive* proof of a formula $\forall x \exists y\, B(x, y)$ (with $\exists$ defined by $\neg\forall\neg$ and $B(x, y)$ quantifier-free):
   a. Use the instanciated proof as a program.
   b. Transform the non-constructive proof into a constructive proof ('A-translation') and extract a program.

Note that a constructive proof of $\forall x \exists^* y\, B(x, y)$ is usually based on an idea for an algorithm, and hence it is not surprising that one can extract a corresponding program from the proof. Therefore paradigms (1) and (2) are not too different from each other. However, there are advantages of (2) over (1):

- One need not write a program explicitely; the machine will extract it automatically from the formal proof (which can be machine-checked to be correct).
- More importantly, when dealing with proofs instead of mere programs it is possible to adapt proofs to special cases (Goad's *pruning* [4]), and then extract a better program (cf. e.g. [5]).

Concerning program extraction from classical proofs (3), it is known that (3a) and (3b) yield the same algorithm (cf. [2]). Moreover, for (3b) refinements are possible (and necessary), in order to obtain reasonable programs (cf. [3]).

In the present note we will only deal with (1), and to a lesser extent with (2). Rather than developing much theory we shall treat a case study of some practical interest: the algorithm of DIJKSTRA, computing the minimal distances from a fixed node in a weighted graph. The aim is to demonstrate that it is possible to interactively construct a formal proof, and extract a reasonable program from it. The theory behind this extraction process is the well-known (modified) *realizability*.

The plan of this note is as follows. We first recall the DIJKSTRA algorithm and prove its correctness. Then we comment on some issues coming up in a formalization of this proof, which has been carried out in the MINLOG system under development in Munich.

1 The Dijkstra algorithm: correctness proof

DIJKSTRA's algorithm deals with a weighted graph; it determines the minimal distances from a fixed node. The point is that it runs in time $O(n^2)$, whereas the naive algorithm needs $O(n^3)$ steps.

Let $\Gamma = (V, E, w)$ be a weighted graph, where V is the (finite) set of *nodes*, $E \subseteq \{\, \{a,b\} \mid a, b \in V \,\}$ is the set of *edges* and $w\colon \{\, \{a,b\} \mid a,b \in V \,\} \to \mathbb{N} \cup \{\infty\}$ is the *weight* function. We write $w(a,b)$ for $w(\{a,b\})$; hence $w(a,b) = w(b,a)$ by definition. We also require that

1. $w(a,b) = 0$ iff $a = b$.
2. For $a \neq b$, $w(a,b) = \infty$ iff there is no edge between a and b, i.e. $\{a,b\} \notin E$.

We need to find a map $d\colon V \to \mathbb{N} \cup \{\infty\}$ such that $d(b)$ is the length of the shortest path from the distinguished node b_0 to b. For simplicity let us assume that the weighted graph has the set $V = \{0, 1, \dots, N-1\}$ of nodes, and that 0 is the distinguished node. Here is a well-known formulation of DIJKSTRA's algorithm:

$$
\begin{array}{l}
d(a) := w(a,0);\\
U := V \setminus \{0\} \quad (= \text{set of undone nodes});\\
\text{while } U \neq \emptyset \text{ do}\\
\quad \langle \text{pick } c \in U \text{ such that } d(c) = \min_{a \in U} d(a) \rangle;\\
\quad \text{for } a \in U \text{ do}\\
\qquad d(a) := \min\{d(a), d(c) + w(c,a)\};\\
\quad \text{end};\\
\quad U := U \setminus \{c\};\\
\text{end};
\end{array}
$$

Clearly the complexity is $O(n^2)$. In the table 1 below we give an example for the working of the DIJKSTRA algorithm.

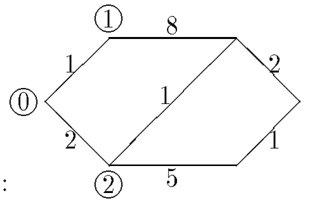
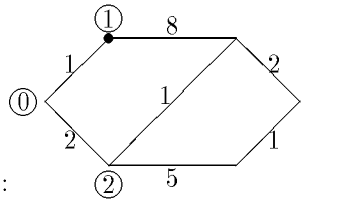
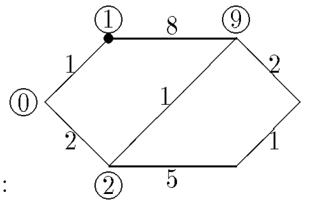
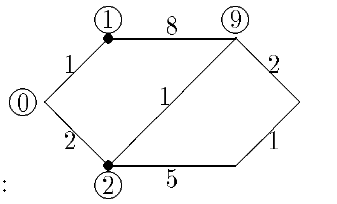
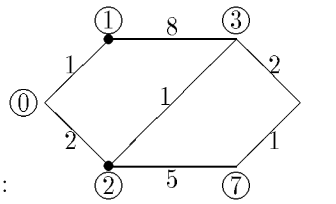
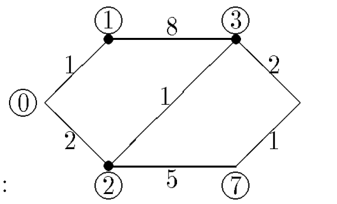
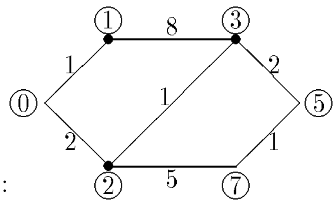
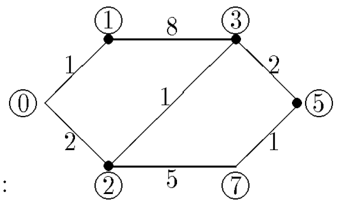
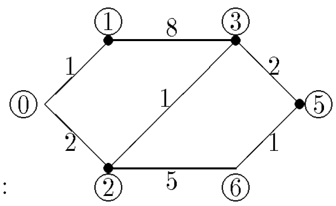
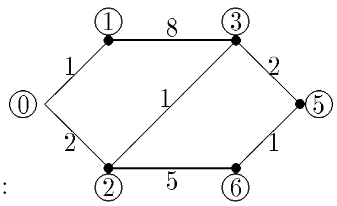
Update distance	Pick new node
d_0: 	p_0: 
d_1: 	p_1: 
d_2: 	p_2: 
d_3: 	p_3: 
d_4: 	p_4: 

Table 1: An example for the Dijkstra Algorithm

dijkstraalg

1.1. *The* DIJKSTRA *algorithm.* For all $n < N$ we will define

- a set $C_n \subseteq \mathbb{N}$, the set of *computed* nodes at stage n,
- a function $\mathsf{nxt}_n : \mathbb{N} \to \mathbb{N}$, giving for every node the *next* node in a shortest path from a to 0, as far as it is known at stage n,
- a function $d_n : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$, giving for every node the *distance* from 0 along the path to 0 determined by the next-function, as far as it is known at stage n,
- p_n, the node *picked* at stage n.

The reason for including the next-function is that it is needed for writing a specification of the algorithm in 1.2. In any case, it seems reasonable that the algorithm not only returns minimal distances, but also corresponding paths.

Let

$$C_0 := \{0\}, \qquad \mathsf{nxt}_0(a) := 0, \qquad d_0(a) := w(a, 0).$$

Assume that at stage n we have picked p_n such that

$$p_n < N \wedge p_n \notin C_n \wedge \forall a{<}N\,(d_n(a) < d_n(p_n) \to a \in C_n). \tag{1}$$

Let

$$\begin{aligned}
C_{n+1} &:= C_n \cup \{p_n\},\\
\mathsf{nxt}_{n+1}(a) &:= \begin{cases} p_n & \text{if } d_n(p_n) + w(a, p_n) < d_n(a),\\ \mathsf{nxt}_n(a) & \text{otherwise,}\end{cases}\\
d_{n+1}(a) &:= \min\{d_n(p_n) + w(a, p_n), d_n(a)\}.
\end{aligned}$$

Note that the algorithm is relative to another simple search algorithm pick that for a given set $C \subsetneq V$ and distance function $d \colon V \to \mathbb{N} \cup \{\infty\}$ yields some $\mathsf{pick}(C, d) \in V \setminus C$ which among all $a \in V \setminus C$ has minimal distance $d(a)$.

In table 2 below we show how the next-function is constructed, for the same example as before.

spec

1.2. *Specification.* Let $\#C$ denote the number of elements of C (see 2.2 below for how counting may be treated formally). We shall prove, for every $n < N$

$$\#C_n = n + 1, \tag{2}$$

$$d_n(0) = 0, \tag{3}$$

$$a{\neq}0 \to \mathsf{nxt}_n(a) \neq a, \tag{4}$$

$$\mathsf{nxt}_n(a) \in C_n \wedge d_n(a) = d_n(\mathsf{nxt}_n(a)) + w(a, \mathsf{nxt}_n(a)), \tag{5}$$

$$b \in C_n \to d_n(a) \leq d_n(b) + w(a, b) \wedge (a < N \to d_n(a) < d_n(b) \to a \in C_n). \tag{6}$$

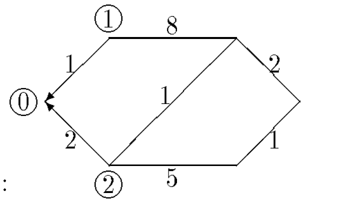
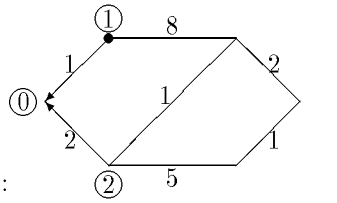
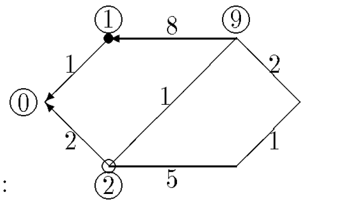
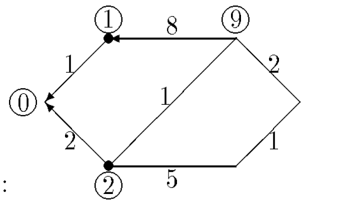
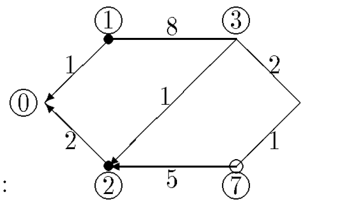
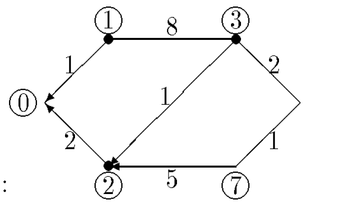
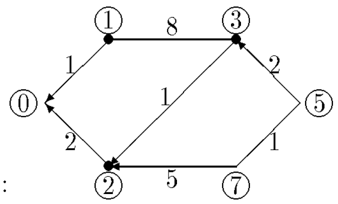
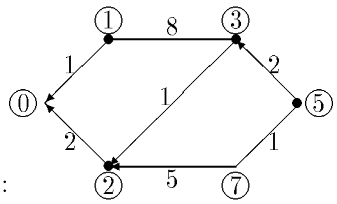
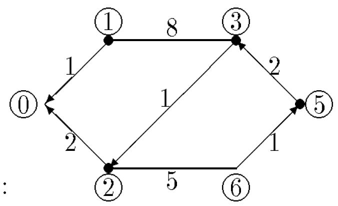
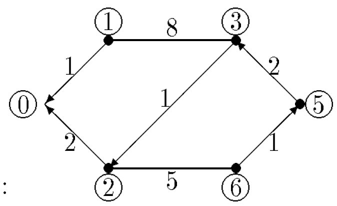
Update distance, next	Pick new node
d_0: 	p_0: 
d_1: 	p_1: 
d_2: 	p_2: 
d_3: 	p_3: 
d_4: 	p_4: 

Table 2: An example for the DIJKSTRA Algorithm, with the next-function

Let us first argue that (2)–(6) for $n := N-1$ can be viewed as a specification of the DIJKSTRA algorithm.

Every node $a < N$ is in C_{N-1} by (2). By (4) and (5), $d_{N-1}(\mathsf{nxt}_{N-1}(a)) < d_{N-1}(a)$ for $a \neq 0$. Hence the path $a, \mathsf{nxt}_{N-1}(a), \mathsf{nxt}_{N-1}(\mathsf{nxt}_{N-1}(a)), \dots$ must lead to 0, and by (5) $d_{N-1}(a)$ gives the distance of a along this path. It remains to show that any other path f with associated distance function d yields a distance greater than or equal to this one, i.e.

1.3. Theorem. *(Correctness).*

$$\begin{aligned}\forall d, f, n\,[f(0) = 0 \to \forall m{<}n{+}1\,(f(m) < N) \to \\ \forall m{<}n{+}1\,[d(f(m)) + w(f(m), f(m+1)) = d(f(m+1))] \to \\ d_{N-1}(f(n)) \leq d(f(n))].\end{aligned}$$

Of course, we could avoid to introduce the variable d and instead of $d(f(n))$ use $\sum_{i=1}^{n} w(f(i-1), f(i))$. However, the formal proof gets more readable when we use d.

Proof. By induction on n. *Base* $n = 0$. Immediate from (3). *Step* $n+1$. First note that by (2) at stage $N-1$ all nodes are computed. Hence

$$\begin{aligned}d_{N-1}(f(n+1)) &\leq d_{N-1}(f(n)) + w(f(n), f(n+1)) && \text{by (6)}\\ &\leq d(f(n)) + w(f(n), f(n+1)) && \text{by ind. hypothesis}\\ &= d(f(n+1)).\end{aligned}$$

□

1.4. It remains to prove the invariants (2)–(6), by induction on n. The base case $n = 0$ is obvious. For the step case assume that (2)–(6) hold for n, and that p_n is chosen according to (1), i.e.

$$p_n < N \wedge p_n \notin C_n \wedge \forall a{<}N\,(d_n(a) < d_n(p_n) \to a \in C_n).$$

lem

Lemma. $\forall a{\in}C_n\,(d_{n+1}(a) = d_n(a))$.

Proof. Let $a \in C_n$ be given. The claim follows from the definition of $d_{n+1}(a)$ and

$$\begin{aligned}d_n(a) &\leq d_n(p_n) && \text{since otherwise } p_n \in C_n \text{ by ind. hypothesis (6)}\\ &\leq d_n(p_n) + w(a, p_n).\end{aligned}$$

□

Now (2) follows from $p_n < N \wedge p_n \notin C_n$, and (3), (4) follow from the definition and the induction hypothesis.

1.5. *Proof of* $\forall a\,(\mathsf{nxt}_{n+1}(a) \in C_{n+1})$. By definition we have $\mathsf{nxt}_{n+1}(a) \in \{\mathsf{nxt}_n(a), p_n\}$, and by induction hypothesis (5) $\mathsf{nxt}_n(a) \in C_n \subseteq C_{n+1}$, and also $p_n \in C_{n+1} = C_n \cup \{p_n\}$.

1.6. *Proof of* $\forall a\,(d_{n+1}(a) = d_{n+1}(\mathsf{nxt}_{n+1}(a)) + w(a, \mathsf{nxt}_{n+1}(a)))$. *Case* 1: $d_n(p_n) + w(a, p_n) < d_n(a)$. Note that $d_{n+1}(p_n) = d_n(p_n)$, since $d_{n+1}(p_n) = \min\{d_n(p_n) + w(p_n, p_n), d_n(p_n)\}$. Then we have

$$\begin{aligned}
&d_{n+1}(a) \\
&= d_{n+1}(p_n) + w(a, p_n) && \text{by definition and the note above} \\
&= d_{n+1}(\mathsf{nxt}_{n+1}(a)) + w(a, \mathsf{nxt}_{n+1}(a)) && \text{by definition of } \mathsf{nxt}_{n+1}(a).
\end{aligned}$$

Case 2: Otherwise. Then by definition $d_{n+1}(a) = d_n(a)$ and $\mathsf{nxt}_{n+1}(a) = \mathsf{nxt}_n(a)$, hence the claim follows from the induction hypothesis and lemma 1.4, using $\mathsf{nxt}_n(a) \in C_n$.

1.7. *Proof of* $\forall a, b\,[b \in C_{n+1} \to d_{n+1}(a) \leq d_{n+1}(b) + w(a, b)]$. Let $b \in C_{n+1}$. *Case* $b \in C_n$. Then

$$\begin{aligned}
d_{n+1}(a) &\leq d_n(a) && \text{by definition} \\
&\leq d_n(b) + w(a, b) && \text{by induction hypothesis} \\
&= d_{n+1}(b) + w(a, b) && \text{by the lemma.}
\end{aligned}$$

Case 2: $b = p_n$. Again note that $d_{n+1}(p_n) = d_n(p_n)$, since $d_{n+1}(p_n) = \min\{d_n(p_n) + w(p_n, p_n), d_n(p_n)\}$. Then

$$\begin{aligned}
d_{n+1}(a) &\leq d_n(p_n) + w(a, p_n) && \text{by definition} \\
&= d_{n+1}(p_n) + w(a, p_n) && \text{by the remark} \\
&= d_{n+1}(b) + w(a, b).
\end{aligned}$$

1.8. *Proof of* $\forall a, b\,[b \in C_{n+1} \to a < N \to d_{n+1}(a) < d_{n+1}(b) \to a \in C_{n+1}]$. Assume $b \in C_{n+1}$, $a < N$ and $d_{n+1}(a) < d_{n+1}(b)$. *Case* $d_n(p_n) + w(a, p_n) < d_n(a)$, i.e. $d_{n+1}(a) = d_n(p_n) + w(a, p_n)$, hence

$$\begin{aligned}
d_n(p_n) &\leq d_n(p_n) + w(a, p_n) \\
&= d_{n+1}(a) && \text{by definition} \\
&< d_{n+1}(b) && \text{by assumption} \\
&\leq d_n(b) && \text{by definition.}
\end{aligned}$$

Subcase $b \in C_n$. Then from the induction hypothesis and the inequality above we have $p_n \in C_n$, a contradiction. *Subcase* $b = p_n$. Then the inequality above immediately yields the contradiction $d_n(p_n) < d_n(p_n)$.

Case otherwise, i.e. $d_{n+1}(a) = d_n(a)$, hence

$$\begin{aligned}
d_n(a) &= d_{n+1}(a) && \text{by definition} \\
&< d_{n+1}(b) && \text{by assumption} \\
&\leq d_n(b) && \text{by definition.}
\end{aligned}$$

Subcase $b \in C_n$. Then from the induction hypothesis we obtain $a \in C_n$. *Subcase* $b = p_n$. Then the inequality above yields $a \in C_n$ by the choice of p_n.

2 Some comments on the formalization

2.1. *How to represent the algorithm in the formal proof.* There are several ways to enter the definition of the program: One might use a constant together with a set of defining equations, as has been done in the informal proof given above. This would give rise to a lot of equational reasoning in the formal proof, which can easily be avoided if we enter the definition as a term rewrite system and identify terms that have the same normal form.

Still, there are two ways to implement the term rewrite system: Either one uses a constant, again, and implements the rewrite rules for this constant directly, or one enters a term that computes the algorithm using the appropriate recursors already implemented in the system. Both methods have their advantages and disadvantages:

- the term that computes the algorithm gets rather long and shows up in the formula stating its correctness more than once, so that this formula can get unreadable. This does not happen if we use constants instead (similar to the informal proof).
- on the other hand, if we use constants, we are only able to deal with the case distinctions used in the program by using a set of equations, again, since these case distinctions do not show up explicitly and are only computed when the constant is applied to concrete data.
- finally, if we use a term, we do not have to worry about termination and confluence of our term rewrite system, since these are already checked for the recursors we are using.

To illustrate these points and the use of program extraction in our example, we show some details of the formalization:

The Dijkstra algorithm (cf. 1.1) is given by the term

```
(nat-rec
  (([a]a=0) # ([a]0) # (w 0))
  ([n,cp*nxt*d]
   (([a] [if a=(pick cp d)) then true else cp a]) #
    ([a] [if (d (pick cp d))+(w a (pick cp d)) < (d a)
             then (pick cp d)
             else (nxt a)]) #
    ([a] [if (d (pick cp d))+(w a (pick cp d)) < (d a)
             then (d (pick cp d))+(w a (pick cp d))
             else (d a)])))
(nn-1)
```

For readability we have replaced

`(car cp*nxt*d)`	by	`cp`,
`(car (cdr cp*nxt*d))`	by	`nxt`,
`(cdr (cdr cp*nxt*d))`	by	`d`.

This term defines triples of sets `cp` and functions `nxt` and `d` for every n by recursion over n using the recursor `nat-rec`. The result is the triple defined for `nn-1` (corresponding to $N-1$ in the informal proof). We have chosen to represent the pick-function (computing p_n) by a constant instead of a term, because we do not need the explicit definition of this function in the proof, we only make use of the properties given in (1).

We have already mentioned that the invariants (2)–(6) would become unreadable, if we use this term. Program extraction can solve this dilemma by providing a way to separate the specification for the algorithm (i.e. the invariants (2)–(6)) from the algorithm itself: We prove the goal formula

```
all n. n<nn -> ex cp,nxt,d.
               count cp = n+1 &
               d 0 = zero_natinf &
               (all a. cp (nxt a) &
                       d a = d (nxt a) + w a (nxt a)) &
               all a,b. cp b -> d a <= d b + w a b &
                                 (a<nn -> d a < d b -> cp a)
```

Note that we have left out invariant (4) to make the formal proof shorter. Anyway, we can easily prove (4) from (5), if we assume $w(a, b) = 0$ iff $a = 0 = b$ (instead of $a = b$), so adding (4) to the formula above would give just a slight generalization.

The first step in the proof is an induction over `n`. In the base case, we assume `0<nn` and instantiate the existential quantifiers appropriately:

```
(ex-intro (pt "[a] a=0"))
(ex-intro (pt "[a] 0"))
(ex-intro (pt "[a] w a 0"))
```

This already shows that using program extraction, we can enter the algorithm in almost exactly the same way as we have done in the informal proof. In the step case, let `cp1`, `nxt1` and `d1` be the corresponding functions from the induction hypothesis. We first show the lemma

```
ex c. c<nn & (cp1 c -> F) & all a. a<nn -> d1 a < d1 c -> cp1 a
```

(cf. (1)) and then define `cp`, `nxt` and `d` for `n+1` by entering

```
(ex-intro (pt "[a] [if a=c then true else cp1 a]"))
```

```
(ex-intro (pt "[a] [if d1 c + w a c < d1 a
                        then c else nxt1 a]"))
(ex-intro (pt "[a] [if d1 c + w a c < d1 a
                        then d1 c + w a c else d1 a]"))
```

To illustrate the benefits we gain from entering the algorithm using terms of our language and the use of term rewrite systems, we show the formal proof of the left side of invariant (5): We have to show

```
[if [if (d1 c)+(w a c)<(d1 a) then c else nxt1 a]=c
    then true
    else cp1([if (d1 c)+(w a c)<(d1 a) then c else nxt1 a])]
```

Looking at the term, it seems obvious to do this by the following case analysis:

```
(cases (pt "(d1 c)+(w a c)<(d1 a)"))
```

Now we have to show

```
(d1 c)+(w a c)<(d1 a) ->
[if [if true then c else nxt1 a]=c
    then true
    else cp1([if true then c else nxt1 a])]
```

and

```
((d1 c)+(w a c)<(d1 a) -> F) ->
[if [if false then c else nxt1 a]=c
    then true
    else cp1([if false then c else nxt1 a])]
```

In the first case, we simply normalize the terms in the formula and have to show `(d1 c)+(w a c)<(d1 a) -> T` , where `T` stands for the truth, so this case is trivial. In the second case, we need the corresponding induction hypothesis, which is referenced by the assumption variable `ih12` :

```
ih12:all a.cp1(nxt1 a) & (d1 a)=(d1(nxt1 a))+(w a(nxt1 a))
```

We instantiate this using

```
(inst-with 'ih12 'a 'left)
```

and get

```
ih12i:cp1(nxt1 a)
```

The normalized formula we have to show is

```
[if (nxt1 a)=c then true else cp1(nxt1 a)]
```

We simplify this formula using

```
(simp 'ih12i)
```

and hence get

```
[if (nxt1 a)=c then true else true]
```

which normalizes to `T`.

count

2.2. *Counting.* In the formal proof we also need some elementary properties of a counting operator $\#C$ assigning to every set C the number of its elements. Let $\#C := \#_N C$, where $\#_j C$ is the number of elements of C less than j, so

$$\#_0 C \;:=\; 0, \qquad \#_{j+1} C \;:=\; \#_j C + \begin{cases} 1 & \text{if } j \in C \\ 0 & \text{otherwise} \end{cases}$$

Lemma.

$$\#_j \emptyset = 0, \tag{7}$$
$$\#_j C \le j, \tag{8}$$
$$j \le \#_j C \to b < j \to b \in C, \tag{9}$$
$$j \le b \to \#_j (C \cup \{b\}) = \#_j C, \tag{10}$$
$$b < j \to b \notin C \to \#_j (C \cup \{b\}) = 1 + \#_j C, \tag{11}$$
$$\forall b{<}j\, (b \in C) \to j \le \#_j C. \tag{12}$$

Proof. (7) and (8). By induction on j, using the definition of $\#_j C$.

(9). By induction on j. *Base* $j = 0$. Then the second premise is false. *Step* $j+1$. *Case* $j \in C$. Then $j + 1 \le \#_{j+1} C = \#_j C + 1$, hence $j \le \#_j C$, hence $b \in C$ by induction hypothesis. *Case* $j \notin C$. Then $j + 1 \le \#_{j+1} C = \#_j C$ and $\#_j C \le j$ by (8); this is a contradiction.

(10). By induction on j. *Base* $j = 0$. The claim follows from $\#_0 C = 0$. *Step* $j + 1$. Assume $j + 1 \le b$, hence $j < b$. We must show

$$\#_j (C \cup \{b\}) + \begin{cases} 1 & \text{if } j \in C \cup \{b\} \\ 0 & \text{otherwise} \end{cases} = \#_j C + \begin{cases} 1 & \text{if } j \in C \\ 0 & \text{otherwise} \end{cases}$$

Because of $j \ne b$ we have $j \in C \;\equiv\; j \in C \cup \{b\}$, hence the claim follows from the induction hypothesis.

(11). By induction on j. *Base* $j = 0$. Then the first premise is false. *Step* $j + 1$. We must show

$$\#_j (C \cup \{b\}) + \begin{cases} 1 & \text{if } j \in C \cup \{b\} \\ 0 & \text{otherwise} \end{cases} = 1 + \#_j C + \begin{cases} 1 & \text{if } j \in C \\ 0 & \text{otherwise.} \end{cases}$$

Case $b < j$. Because of $j \neq b$ we have $j \in C \equiv j \in C \cup \{b\}$, hence the claim follows from the induction hypothesis. *Case* $b = j$. Then we are in the first case of the left hand side and in the second case of the right hand side. But $\#_j(C \cup \{b\}) = \#_j C$ by (10).

(12). By induction on j. *Base* $j = 0$. Then the claim is trivial. *Step* $j+1$. From $j \in C$ by definition we have $\#_{j+1} C = \#_j C + 1$; the claim now follows from the induction hypothesis. □

pick

2.3. *The* pick-*function.* In (1) we have made use of the rather obvious fact that from a nonempty set of 'undone' nodes we can pick one with least value w.r.t. a given measure function. Now we want to be more specific about the computation procedure for this effect. We give an informal proof which relates to the formal proof in much the same way as the informal correctness proof for Dijkstra's algorithm to its formalization.

Starting from the initial node $j = 0$, we go through all nodes and select at each stage the 'best' node found up to that point (and N if have not encountered an undone node yet). More precisely, we define c_j for every $j \leq N$ by recursion over j:

$$c_0 := N,$$
$$c_{j+1} := \begin{cases} c_j & \text{if } j \in C \text{ or } d(c_j) < d(j), \\ j & \text{otherwise.} \end{cases}$$

where C is the set of computed nodes and d the given measure function (corresponding to some C_n and d_n in Dijkstra's algorithm). Assuming $d(N) = \infty$, we can inductively show the following invariants for c_j:

$$\#_j C < j \to c_j < N \wedge c_j \notin C \wedge (a < j \to d(a) < d(c_j) \to a \in C), \tag{13}$$
$$\#_j C = j \to c_j = N \tag{14}$$

It immediately follows that $\mathsf{pick}(C, d) := c_N$ is the node we were looking for, if $\#C = \#_N C < N$.

Now we prove the invariants for all $j \leq N$. The base case $j = 0$ is trivial. In the step case assume that (13) and (14) hold for c_j.

Proof of $\#_{j+1} C < j + 1 \to c_{j+1} < N \wedge c_{j+1} \notin C \wedge (a < j \to d(a) < d(c_{j+1}) \to a \in C)$.

Case 1: $j \in C \vee d(c_j) < d(j)$, so $c_{j+1} = c_j$. Then we have $\#_j C < j$, since otherwise $c_j = N, j \notin C$ (by definition of $\#_{j+1}$) and therefore $\infty = d(N) = d(c_j) < d(j)$, a contradiction. So we get $c_{j+1} < N$, $c_{j+1} \notin C$ and $a < j \to d(a) < d(c_{j+1}) \to a \in C$ from the induction hypothesis. All we have to show is $a = j \to d(a) < d(c_{j+1}) \to a \in C$. This follows from $j \in C \vee d(c_j) < d(j)$.

Case 2: $j \notin C \wedge d(j) \leq d(c_j)$, so $c_{j+1} = j$. We have $j < N$ and $j \notin C$ by assumption. Let $a < j + 1$ with $d(a) < d(j)$. We have to show $a \in C$.

Since $d(j) < d(j)$ is impossible, we have $a < j$. *Case* 2.1: $\#_j C < j$. Then we get $a \in C$ from the induction hypothesis and $d(a) < d(j) \leq d(c_j)$. *Case* 2.2: $\#_j C = j$. Then $a \in C$ follows from (9).

Proof of $\#_{j+1} C = j+1 \to c_{j+1} = N$. We get $\#_j C = j$ and $j \in C$ from the definition of $\#_{j+1}$. Hence $c_{j+1} = c_j$ by definition and $c_j = N$ by induction hypothesis. □

The formalization of this proof uses the same techniques as the formalization of the correctness proof for the DIJKSTRA algorithm: The pick-function is represented by the term

```
[cp,d]
nat-rec
 nn
 [j,c] [if cp j then
           c else
           [if (d c)<(d j) then
               c else
               j]]
```

and we use program extraction, again, to enter its definition in the same intuitive way as in the informal description.

formal

2.4. *Informal versus formal proofs.* When interactively carrying out the proof, we found that the concepts and arguments used in the informal proof had to be modified considerably in order to find useful invariants. However, the necessity to actually carry out all details also helped in making the final proof more perspicious.

3 Conclusion

So the final algorithm is entered in two stages. First, we prove the existence of c_j (for every C, d and j) satsfying (13) and (14). For C, d and the special case $j = N$ this yields $\mathsf{pick}(C, d) := c_N$. Using this pick-function for the lemma

```
ex c. c<nn & (cp1 c -> F) & all a. a<nn -> d1 a < d1 c -> cp1 a
```

we then prove the goal formula. In this way we have proved our algorithm correct.

We can then go on and let the machine extract an implementation of the algorithm from the formal proof. Of course, the result will be the same as what we had in mind and quite explicitely put into the existence-introduction steps of our formal proof. However, the program extracted from the proof is executable code, and it is reliable since the correctness of the formal proof it is extracted from can be machine-checked.

References

[1] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[2] Ulrich Berger and Helmut Schwichtenberg. Program extraction from classical proofs. In D. Leivant, editor, *Logic and Computational Complexity, International Workshop LCC '94, Indianapolis, IN, USA, October 1994*, volume 960 of *Lecture Notes in Computer Science*, pages 77–97. Springer Verlag, Berlin, Heidelberg, New York, 1995.

[3] Ulrich Berger and Helmut Schwichtenberg. The greatest common divisor: a case study for program extraction from classical proofs. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs. International Workshop TYPES '95, Torino, Italy, June 1995. Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 36–46. Springer Verlag, Berlin, Heidelberg, New York, 1996.

[4] Christopher Alan Goad. *Computational uses of the manipulation of formal proofs*. PhD thesis, Stanford University, August 1980. Stanford Department of Computer Science Report No. STAN–CS–80–819.

[5] Helmut Schwichtenberg. Programmentwicklung durch Beweistransformation: Das Maximalsegmentproblem. *Sitzungsber. d. Bayer. Akad. d. Wiss., Math.-Nat. Kl.*, pages 8*–12*, 1997.