Wildcard Matching in the Spirit of E. W. Dijkstra

Herwig Egghart, Scientific Games International

June 25, 2015

Let **a,b**... denote symbols from some alphabet **Σ**, and let ∗ be a special symbol outside **Σ**.
s,t... shall denote finite strings over **Σ**, and **p,q**... shall denote finite strings over **Σ**∪**{∗}**.

Then we define the wildcard-matching relation **~** between "sample" and "pattern" strings by the following little calculus:

(0) ⊢ s ~ s

(1) ⊢ s ~ ∗

(2) s ~ p, t ~ q ⊢ s·t ~ p·q

The two axioms (0) and (1) capture our intuition that any string matches itself, and any string matches a star. Rule (2) expresses the compatibility between wildcard matching and string concatenation (indicated by a dot-shaped infix operator, which is non-symmetric and associative).

Is this calculus a good basis to derive a program that decides whether **s ~ p** ?
No, because it only enumerates successful matches, while our program will also have to detect unsuccessful matches. Therefore we shall rephrase the properties of **~** in a more systematic way based on all possible strings on both sides of the relation.

The sample (LHS) can either be the empty string **ε**, or some arbitrary symbol **a** followed by a tail string **s**. The pattern (RHS) can either be **ε**, or some non-star symbol **b** followed by tail **p**, or a star symbol followed by tail **p**. If we combine the two LHS and the three RHS cases, we obtain the following six laws:

(3) ε ~ ε Ξ true

(4) a·s ~ ε Ξ false

(5) ε ~ b·p Ξ false

(6) a·s ~ b·p Ξ a = b ∧ s ~ p

(7) ε ~ ∗·p Ξ ε ~ p

(8) a·s ~ ∗·p Ξ a·s ~ p ∨ s ~ ∗·p

Although most of these laws are fairly obvious, the total ensemble (3 – 8) is not as simple as calculus (0 – 2) any more. Therefore we had better convince ourselves that both definitions are equivalent, i.e. that the calculus is sound and complete with respect to the laws.

Let us start with soundness, and more specifically with the two axioms. Can (0) and (1) be derived from (3 – 8) for arbitrary **s** ? Well, **s** is either empty or non-empty. (0) for empty **s** evaluates to:

ε ~ ε
= { (3) } Remark The curly brackets contain a hint to justify the proof step. (End of Remark.)
true

Now let **s** be non-empty, composed of **a** and **t**, say:

```
    a·t ~ a·t
=   { (6) }
    a = a  ∧  t ~ t
=   { drop true conjunct }
    t ~ t
```

This looks much like our original proof obligation **s ~ s**, except for the different letter! However, since **t** is always one symbol shorter than **s**, repeated application of this proof step eventually leads to **s = ε**, for which (0) holds as we have seen. Thus axiom (0) is sound.

Similarly, we observe for axiom (1) and empty **s**:

```
    ε ~ *
=   { (7) with p := ε }
    ε ~ ε
=   { (3) }
    true
```

Otherwise let **s = a·t** again:

```
    a·t ~ *
=   { (8) with p := ε }
    a·t ~ ε  ∨  t ~ *
=   { (4) }
    false  ∨  t ~ *
=   { drop false disjunct }
    t ~ *
```

Like with axiom (0), we have gotten rid of one symbol in this proof step. So due to the same induction argument, axiom (1) is sound as well.

To prove the soundness of rule (2), we have to show

(9) $s \sim p \;\wedge\; t \sim q \;\Rightarrow\; s{\cdot}t \sim p{\cdot}q$ (concatenation lemma)

for arbitrary samples and patterns.

The consequent **s·t ~ p·q** calls for a case distinction based on the structure of **s** and **p**, because they appear at the beginning of **s·t** and **p·q**, and our laws work from left to right. So we have the six cases we already know from the construction of the six laws.

Case 3:

```
    ε ~ ε  ∧  t ~ q
=   { (3) }
    true  ∧  t ~ q
=   { drop true conjunct }
    t ~ q
=   { ε unit element }
    ε·t ~ ε·q
```

Case 4:

$a \cdot s \sim \varepsilon \ \wedge \ t \sim q$
$=$ { (4) }
$\text{false} \ \wedge \ t \sim q$
$\Rightarrow$ { ex falso quodlibet }
$a \cdot s \cdot t \sim \varepsilon \cdot q$

Case 5:

$\varepsilon \sim b \cdot p \ \wedge \ t \sim q$
$=$ { (5) }
$\text{false} \ \wedge \ t \sim q$
$\Rightarrow$ { ex falso quodlibet }
$\varepsilon \cdot t \sim b \cdot p \cdot q$

Case 6:

$a \cdot s \sim b \cdot p \ \wedge \ t \sim q$
$=$ { (6) }
$a = b \ \wedge \ s \sim p \ \wedge \ t \sim q$
$\Rightarrow$ { induction hypothesis }
$a = b \ \wedge \ s \cdot t \sim p \cdot q$
$=$ { (6) }
$a \cdot s \cdot t \sim b \cdot p \cdot q$

Case 7:

$\varepsilon \sim * \cdot p \ \wedge \ t \sim q$
$=$ { (7) }
$\varepsilon \sim p \ \wedge \ t \sim q$
$\Rightarrow$ { induction hypothesis }
$\varepsilon \cdot t \sim p \cdot q$
$\Rightarrow$ { (7) or (8) depending on t }
$\varepsilon \cdot t \sim * \cdot p \cdot q$

Case 8:

$a \cdot s \sim * \cdot p \ \wedge \ t \sim q$
$=$ { (8) }
$(a \cdot s \sim p \ \vee \ s \sim * \cdot p) \ \wedge \ t \sim q$
$=$ { $\wedge$ distributes over $\vee$ }
$(a \cdot s \sim p \ \wedge \ t \sim q) \vee (s \sim * \cdot p \ \wedge \ t \sim q)$
$\Rightarrow$ { induction hypothesis twice }
$a \cdot s \cdot t \sim p \cdot q \ \vee \ s \cdot t \sim * \cdot p \cdot q$
$=$ { (8) with $s, p := s \cdot t, p \cdot q$ }
$a \cdot s \cdot t \sim * \cdot p \cdot q$

Hence the concatenation lemma (9) always holds, and so the whole calculus (0 – 2) is sound.

Now to the completeness part: Can the calculus produce <u>every</u> wildcard match that holds according to laws (3 – 8) ? For law (3), the answer is easy:

⊢ { (0) with s := ε }
ε ~ ε

Laws (4) and (5) only specify false matches – which cannot be produced as the calculus is sound – so there is nothing left to show.

Laws (6 – 8) specify conditional matches, with all RHS occurrences of ~ (s ~ p; ε ~ p; a·s ~ p, s ~ ∗·p) containing fewer symbols than the corresponding LHS occurrences (a·s ~ b·p; ε ~ ∗·p; a·s ~ ∗·p). Hence we can use ⊢ RHS as induction hypothesis and only need to show that ⊢ LHS follows, ideally by showing RHS ⊢ LHS.

For law (6):

a = b, s ~ p
⊢ { (0) with s := a }
a ~ b, s ~ p
⊢ { (2) }
a·s ~ b·p

For law (7):

ε ~ p
⊢ { (1) with s := ε }
ε ~ ∗, ε ~ p
⊢ { (2) }
ε ~ ∗·p

Law (8) requires two proofs, because any of the conditions **a·s ~ p**, **s ~ ∗·p** is sufficient.
Let us start with the easier part by showing **a·s ~ p** ⊢ **a·s ~ ∗·p**:

a·s ~ p
⊢ { (1) with s := ε }
ε ~ ∗, a·s ~ p
⊢ { (2) }
a·s ~ ∗·p

Can we show **s ~ ∗·p** ⊢ **a·s ~ ∗·p** as well? Unfortunately not, because our calculus has no mechanism for prepending a symbol to the sample alone. If **p** is empty, however, we have:

s ~ ∗
⊢ { (1) with s := a·s }
a·s ~ ∗

And if **p** is non-empty, the only possible source of ⊢ **s ~ ∗·p** is an application of (2) with **∗·p** emerging from some shorter prefix **∗·q** and a non-empty suffix **r**. At the same time, **s** emerges from parts **t** and **u**, say:

t ~ ∗·q, u ~ r ⊢ s ~ ∗·p

Now suppose **(⊢ t ~ ∗·q) ⇒ (⊢ a·t ~ ∗·q)** for all **q** shorter than **p**.
Then we can prove **(⊢ s ~ ∗·p) ⇒ (⊢ a·s ~ ∗·p)** by the following induction step:

```
    ⊢ s ~ ∗·p
⇒   { undo application of (2) }
    ⊢ t ~ ∗·q,  u ~ r
⇒   { induction hypothesis }
    ⊢ a·t ~ ∗·q,  u ~ r
⇒   { (2) }
    ⊢ a·t·u ~ ∗·q·r
=   { s = t·u  ∧  ∗·p = ∗·q·r }
    ⊢ a·s ~ ∗·p
```

Together with the induction basis **(⊢ s ~ ∗·ε) ⇒ (⊢ a·s ~ ∗·ε)** already proved before, this discharges our last proof obligation for the completeness of calculus (0 – 2). So not only do the six laws agree with everything the calculus produces, but the calculus is also able to produce whatever the laws claim to be true. Thus, laws (3 – 8) are an appropriate basis for our intended programming task.

* * *

Let **s** and **p** be the input strings we try to match, and let **y** and **n** be two Boolean outputs indicating whether **s ~ p** ("yes" or "no"). The idea is to initialize both flags with false and wait till the right answer is known. Then we shall change one of the two flags to true.

To make sure that **y** and **n** refer to the original values of **s** and **p**, we introduce the rigid variable **M** for the matching predicate **s ~ p** before program execution. With these ingredients, our program can be specified as follows:

{ M ≡ s ~ p } PROG { (y ≡ M) ∧ (n ≡ ¬M) }

So **y** and **n** depend on **M**, which is of course invisible to the program.
But maybe we can keep **M ≡ s ~ p** invariant while modifying **s** and **p**, thus allowing the program to set the correct flag nevertheless. The invariant should also include a weakened form of the postcondition, which is easily achieved by replacing ≡ with ⇒ and initializing both flags with false.

```
PROG:
    { M ≡ s ~ p }
    y, n := false, false;
I:  { (M ≡ s ~ p) ∧ (y ⇒ M) ∧ (n ⇒ ¬M) }
    do y = n →      { I ∧ ¬y ∧ ¬n }
            BODY    { I }
    od
    { I ∧ y ≠ n }
    { (y ≡ M) ∧ (n ≡ ¬M) }
```

Now is the time to exploit the six laws of wildcard matching! Depending on the structure of **s** and **p**, the loop body should either set a flag or shorten at least one string under invariance of **I**. Either way, the number of false flags plus the total string length will decrease, which ensures termination of the loop. (We have already seen that strings get shorter from LHS to RHS in the completeness proof.)

In the following text, the functions **H** and **T** ("head" and "tail") shall decompose a non-empty string into its first symbol and the rest.

```
BODY:
    { I ∧ ¬y ∧ ¬n }
    if p = ε →
            if s = ε → FRAG3
            [] s ≠ ε → FRAG4
            fi
    [] p ≠ ε →
            if H.p ≠ * →
                    if s = ε → FRAG5
                    [] s ≠ ε → FRAG6
                    fi
            [] H.p = * →
                    if s = ε → FRAG7
                    [] s ≠ ε → FRAG8
                    fi
            fi
    fi
    { I }
```

Note that the six fragments FRAG3 to FRAG8 correspond to the laws (3 – 8).

```
FRAG3:
    { I ∧ s = ε ∧ p = ε }
    { I ∧ M }
    y := true

FRAG4:
    { I ∧ s ≠ ε ∧ p = ε }
    { I ∧ ¬M }
    n := true

FRAG5:
    { I ∧ s = ε ∧ H.p ≠ * }
    { I ∧ ¬M }
    n := true

FRAG6:
    { I ∧ s ≠ ε ∧ H.p ≠ * }
    { I ∧ (M Ξ H.s = H.p ∧ T.s ~ T.p) }
    if H.s = H.p →
            { I ∧ (M Ξ T.s ~ T.p) }
            s, p := T.s, T.p
    [] H.s ≠ H.p →
            { I ∧ ¬M }
            n := true
    fi
```

FRAG7:

{ I ∧ s = ε ∧ H.p = * }
{ I ∧ (M Ξ s ~ T.p) }
p := T.p

So far, everything went smoothly because **~** appears at most once on the RHS of laws (3 – 7). By contrast, the RHS of law (8) is a disjunction of two **~** terms, asserting **M Ξ s ~ T.p** ∨ **T.s ~ p** at the beginning of FRAG8.

But which of these alternatives should we pursue first? Skip the star at **H.p** and try to match **s** against **T.p**? Or skip **H.s** and still face a star to be matched? Well, the latter choice only perpetuates the problem; and its repeated application would discard the whole sample without learning anything new.

So let us first look beyond the star and compare the beginning of **T.p** with the beginning of **s** instead. Note that to avoid a premature commitment, we shall leave **s** and **p** unchanged and just look for a common prefix of **s** and **T.p**:

s = t·u ∧ p = *·t·q

Of course, this is trivial to establish for the empty prefix (t = ε), but maybe we can find the greatest common prefix by "pushing" **t** deeper into **u** and **q**, and then know enough to re-establish **I**.

FRAG8:

{ I ∧ s ≠ ε ∧ H.p = * }
t, u, q := ε, s, T.p;
J: { I ∧ s ≠ ε ∧ s = t·u ∧ p = *·t·q }
PUSH;
{ J ∧ t maximal }
FINISH
{ I }

For the design of PUSH, we calculate the weakest precondition of the smallest possible push step and invariant J:

wp . (t, u, q := t·H.u, T.u, T.q) . J
= { axiom of assignment }
I ∧ s ≠ ε ∧ s = t·H.u·T.u ∧ p = *·t·H.u·T.q
= { unify with J }
J ∧ u = H.u·T.u ∧ q = H.u·T.q
= { definition of H and T }
J ∧ u ≠ ε ∧ q ≠ ε ∧ H.u = H.q

Not surprisingly, we can keep pushing as long as we haven't run out of symbols and the next symbol agrees in both strings. Before we cast our findings into a loop, however, we remember that undefined sub-expressions are forbidden in a guard and therefore introduce a Boolean flag **d** to indicate that a difference has been found.

```
PUSH:
    { J }
    d := false;
K:  { J ∧ (d ⇒ H.u ≠ H.q) }
    do  u ≠ ε  ∧  q ≠ ε  ∧ ¬d →
            if H.u = H.q →
                    { J ∧ ¬d  ∧  u ≠ ε  ∧  q ≠ ε  ∧  H.u = H.q }
                    t, u, q := t·H.u, T.u, T.q
                    { J ∧ ¬d }
            [] H.u ≠ H.q →
                    { J ∧ (true  ⇒  H.u ≠ H.q) }
                    d := true
                    { K }
            fi
            { K }
    od
    { K ∧ (u = ε  ∨  q = ε  ∨  d) }
    { J ∧ (u = ε  ∨  q = ε  ∨  H.u ≠ H.q) }
```

As a by-product of the last derivation, we have also obtained a precise criterion for the maximality of **t**, viz. that **u** or **q** are empty or start with a different symbol.

Now that we have found the common prefix **t**, let us see if we can generalize law (6) from the elimination of a common head symbol to the elimination of a common prefix:

(10) $t \cdot u \sim t \cdot q \;\; \Xi \;\; u \sim q$ (prefix lemma)

For empty **t**, LHS and RHS are equal already; and if we prepend the symbol **a** to **t**, we obtain the following induction step as a proof of (10):

```
    a·t·u ~ a·t·q
=   { (6) }
    t·u ~ t·q
=   { induction hypothesis }
    u ~ q
```

Back to the postcondition of PUSH, which is the precondition of the missing part FINISH. By far the most interesting case is when we PUSH into another star, i.e. **q** has the shape $*\cdot\mathbf{r}$. So how can we compute $\mathbf{t}\cdot\mathbf{u} \sim *\cdot\mathbf{t}\cdot*\cdot\mathbf{r}$?

Obviously $\mathbf{u} \sim *\cdot\mathbf{r}$ is a sufficient condition for $\mathbf{t}\cdot\mathbf{u} \sim *\cdot\mathbf{t}\cdot*\cdot\mathbf{r}$; all we need is calculus (0 – 2):

```
    u ~ *·r
├─  { (0) }
    t ~ t,  u ~ *·r
├─  { (1) }
    ε ~ *,  t ~ t,  u ~ *·r
├─  { (2) twice }
    t·u ~ *·t·*·r
```

But is it also a necessary condition? Again, we start with empty **t** and try to prove:

$$u \sim *\cdot *\cdot r \;\Rightarrow\; u \sim *\cdot r$$

This is trivial for **u := ε**:

$$\begin{array}{ll} & \varepsilon \sim *\cdot *\cdot r \\ = & \{\ (7) \text{ with } p := *\cdot r\ \} \\ & \varepsilon \sim *\cdot r \end{array}$$

Now the induction step **u := a·u**:

$$\begin{array}{ll} & a\cdot u \sim *\cdot *\cdot r \\ = & \{\ (8) \text{ with } s, p := u, *\cdot r\ \} \\ & a\cdot u \sim *\cdot r \;\vee\; u \sim *\cdot *\cdot r \\ = & \{\ \text{induction hypothesis}\ \} \\ & a\cdot u \sim *\cdot r \;\vee\; u \sim *\cdot r \\ = & \{\ (8) \text{ with } s, p := u, r\ \} \\ & a\cdot u \sim r \;\vee\; u \sim *\cdot r \;\vee\; u \sim *\cdot r \\ = & \{\ \vee \text{ idempotent}\ \} \\ & a\cdot u \sim r \;\vee\; u \sim *\cdot r \\ = & \{\ (8) \text{ backward}\ \} \\ & a\cdot u \sim *\cdot r \end{array}$$

Note that the last two calculations consist only of equivalence transformations,
so we have even proved the stronger lemma:

(11) $u \sim *\cdot *\cdot r \;\Xi\; u \sim *\cdot r$ (eclipse lemma)

With this as induction basis, we shall now prepend another symbol **a** to the common prefix **t**:

$$\begin{array}{ll} & a\cdot t\cdot u \sim *\cdot a\cdot t\cdot *\cdot r \\ = & \{\ (8) \text{ with } s, p := t\cdot u, a\cdot t\cdot *\cdot r\ \} \\ & a\cdot t\cdot u \sim a\cdot t\cdot *\cdot r \;\vee\; t\cdot u \sim *\cdot a\cdot t\cdot *\cdot r \\ = & \{\ (10) \text{ with common prefix } a\cdot t\ \} \\ & u \sim *\cdot r \;\vee\; t\cdot u \sim *\cdot a\cdot t\cdot *\cdot r \end{array}$$

Now we are stuck because of the symbol **a** in the second disjunct.
If we could somehow eliminate this symbol, however, the road would be free to exploit our induction hypothesis **t·u ~ ∗·t·∗·r ⇒ u ~ ∗·r**. What we need is something like the eclipse lemma (11), except that the symbol that disappears is now a non-star:

(12) $u \sim *\cdot b\cdot r \;\Rightarrow\; u \sim *\cdot r$ (black-hole lemma)

Pictorially speaking, the star "swallows" an adjacent symbol, but this time there is no hope that the ⇐ direction holds as well. Again we start with **u := ε**:

$$\begin{array}{ll} & \varepsilon \sim *\cdot b\cdot r \\ = & \{\ (7) \text{ with } p := b\cdot r\ \} \\ & \varepsilon \sim b\cdot r \\ = & \{\ (5) \text{ with } p := r\ \} \\ & \text{false} \\ \Rightarrow & \{\ \text{ex falso quodlibet}\ \} \\ & \varepsilon \sim *\cdot r \end{array}$$

Now the induction step **u** := **a**·**u**:

```
    a·u ~ *·b·r
=   { (8) with s, p := u, b·r }
    a·u ~ b·r  ∨  u ~ *·b·r
⇒   { induction hypothesis }
    a·u ~ b·r  ∨  u ~ *·r
=   { (6) with s, p := u, r }
    (a = b  ∧  u ~ r)  ∨  u ~ *·r
⇒   { drop conjunct }
    u ~ r  ∨  u ~ *·r
=   { (1) with s := a }
    (a ~ *  ∧  u ~ r) ∨ (a ~ *  ∧  u ~ *·r)
⇒   { (9) twice }
    a·u ~ *·r  ∨  a·u ~ *·*·r
=   { (11) }
    a·u ~ *·r  ∨  a·u ~ *·r
=   { ∨ idempotent }
    a·u ~ *·r
```

This completes the proof of the black-hole lemma (12),
and we can resume the induction step **a·t·u ~ *·a·t·*·r ⇒ u ~ *·r**.

After applying (8) and (10) to **a·t·u ~ *·a·t·*·r**, we continue:

```
    u ~ *·r  ∨  t·u ~ *·a·t·*·r
⇒   { (12) with u, b, r := t·u, a, t·*·r }
    u ~ *·r  ∨  t·u ~ *·t·*·r
⇒   { induction hypothesis }
    u ~ *·r  ∨  u ~ *·r
=   { ∨ idempotent }
    u ~ *·r
```

In summary, we have shown the easy part **u ~ *·r ⊢ t·u ~ *·t·*·r** as well as the more challenging part **t·u ~ *·t·*·r ⇒ u ~ *·r**. Thus, we can use the following theorem in the design of FINISH:

(13) t·u ~ *·t·*·r ≡ u ~ *·r (eclipse theorem)

Let us now analyze the remaining cases permitted by assertion **J ∧ (u = ε ∨ q = ε ∨ H.u ≠ H.q)**.
If **u** and **q** are both empty, we have **t ~ *·t**, which is always true:

```
⊢ { (0) }
    t ~ t
⊢ { (1) }
    ε ~ *,  t ~ t
⊢ { (2) }
    t ~ *·t
```

If both are non-empty, let **u** = **a·v** and **q** = **b·r** with **a ≠ b**:

```
    t·a·v ~ ∗·t·b·r
=   { (8) }
    t·a·v ~ t·b·r  ∨  T.(t·a·v) ~ ∗·t·b·r
=   { (10) }
    a·v ~ b·r  ∨  T.(t·a·v) ~ ∗·t·b·r
=   { (6) with a ≠ b }
    false  ∨  T.(t·a·v) ~ ∗·t·b·r
=   { drop false disjunct }
    T.(t·a·v) ~ ∗·t·b·r
```

If only **u** is empty, we have **t ~ ∗·t·b·r** and see intuitively that the pattern is "too long", yielding the proof obligation:

(14) t ~ ∗·t·b·r Ξ false

And if only **q** is empty, **t·a·v ~ ∗·t** means that **t·a·v** ends with suffix **t**, so we can immediately shrink the sample to the appropriate length.

Even without a formal treatment of the last two cases, the design of FINISH is already clear from **t ~ ∗·t**, (14), **t·a·v ~ ∗·t·b·r Ξ T.(t·a·v) ~ ∗·t·b·r**, and the eclipse theorem (13).

```
FINISH:
    { J  ∧  (u = ε  ∨  q = ε  ∨  H.u ≠ H.q) }
    if q = ε →
            if u = ε → y := true
            [] u ≠ ε → SHRINK
            fi
    [] q ≠ ε →
            if H.q ≠ ∗ →
                    if u = ε → n := true
                    [] u ≠ ε → s := T.s
                    fi
            [] H.q = ∗ → s, p := u, q
            fi
    fi
    { I }
```

With regard to termination, we notice that each branch achieves progress by setting a flag or shortening a string. In the last branch, the sample might stay the same (if t = ε), but the pattern always loses its first star.

Now to the outstanding proof of (14), which calls for a counting argument over the non-star symbols in a pattern. Guided by calculus (0 – 2), we define a "non-star counting function" **N** from patterns to natural numbers:

(15) N.s = |s|

(16) N.∗ = 0

(17) N.(p·q) = N.p + N.q

The expression **|s|** above denotes the length of **s** measured in symbols.
As a next step, we postulate the following relationship between **~** and **N**:

(18) $s \sim p \;\Rightarrow\; |s| \geq N.p$ (general counting lemma)

The proof is by induction for all wildcard matches enumerated by calculus (0 – 2).

Case 0: $\vdash\; s \sim s$

$|s| \geq N.s$
$=$ { (15) }
$|s| \geq |s|$
$=$ { $\geq$ reflexive }
true

Case 1: $\vdash\; s \sim *$

$|s| \geq N.*$
$=$ { (16) }
$|s| \geq 0$
$=$ { string theory }
true

Case 2: $s \sim p,\; t \sim q \;\vdash\; s{\cdot}t \sim p{\cdot}q$

$|s| \geq N.p \;\wedge\; |t| \geq N.q$
$\Rightarrow$ { arithmetic }
$|s| + |t| \;\geq\; N.p + N.q$
$=$ { string theory }
$|s{\cdot}t| \;\geq\; N.p + N.q$
$=$ { (17) }
$|s{\cdot}t| \;\geq\; N.(p{\cdot}q)$

So if the pattern $*{\cdot}\mathbf{t}{\cdot}\mathbf{b}{\cdot}\mathbf{r}$ in (14) is really "too long" for sample **t**,
we should now be able to write this down as a formal calculation:

$N.(*{\cdot}t{\cdot}b{\cdot}r)$
$=$ { (17) three times }
$N.* + N.t + N.b + N.r$
$=$ { (16) ; (15) twice }
$0 + |t| + 1 + N.r$
$>$ { arithmetic }
$|t|$

The result looks like a negated RHS of (18) (with s, p := t, $*{\cdot}t{\cdot}b{\cdot}r$),
from which the negation of the LHS follows: $\mathbf{|t| < N.(*{\cdot}t{\cdot}b{\cdot}r) \;\Rightarrow\; \neg\, t \sim *{\cdot}t{\cdot}b{\cdot}r}$

This completes the proof of (14), and the only thing left to do is the design of SHRINK satisfying:

$\{\, I \;\wedge\; s = t{\cdot}u \;\wedge\; u \neq \varepsilon \;\wedge\; p = *{\cdot}t \,\}$
SHRINK
$\{\, I \,\}$

What follows is essentially another counting argument, but this time without any stars in the pattern. The impact is that the number of non-stars **N.p** is simply the pattern length **|p|**, and the "≥" from the general counting lemma (18) changes to equality:

(19) $s \sim t \;\Rightarrow\; |s| = |t|$ (special counting lemma)

The proof is analogous to the proof of (18), except that we omit axiom (1).

Case 0: $\vdash\; s \sim s$

$|s| = |s|$
$=$ { = reflexive }
true

Case 2: $s \sim p,\; t \sim q \;\vdash\; s{\cdot}t \sim p{\cdot}q$

$|s| = |p| \;\wedge\; |t| = |q|$
$\Rightarrow$ { Leibniz' Principle } Remark I.e. function application preserves equality. (End of Remark.)
$|s| + |t| \;=\; |p| + |q|$
$=$ { string theory }
$|s{\cdot}t| = |p{\cdot}q|$

Applied to the design of SHRINK, this means that as long as **|s| > |t|**, the first disjunct of

$$s \sim *{\cdot}t \;\Xi\; s \sim t \;\vee\; T.s \sim *{\cdot}t$$

is always false, so **s := T.s** preserves **I**. The precondition of SHRINK even tells us how often we can do this, because **s = t·u** means that the length difference to be eliminated equals **|u|**.

Well, and when **|s| = |t|**, either **s** is empty or **T.s ~ ∗·t** violates the general counting lemma (18) because **|T.s| = |s| - 1** and **N.(∗·t) = 0 + |t|**. In both cases, **s ~ ∗·t Ξ s ~ t**.

The simplest way to iterate **|u|** times is to shrink **s** and **u** simultaneously and make **u ≠ ε** the guard of the loop. Another benefit is the invariance of **|s| = |t·u|**, the "counting version" of **s = t·u**:

wp . (s, u := T.s, T.u) . ((M Ξ s ~ ∗·t) $\wedge$ |s| = |t·u|)
$=$ { axiom of assignment }
(M Ξ T.s ~ ∗·t) $\wedge$ |T.s| = |t · T.u|
$=$ { string theory }
(M Ξ T.s ~ ∗·t) $\wedge$ |s| - 1 = |t·u| - 1 $\wedge$ u ≠ ε
$=$ { arithmetic }
(M Ξ T.s ~ ∗·t) $\wedge$ |s| = |t·u| $\wedge$ u ≠ ε
$=$ { (19) considering |s| > |t| }
(M Ξ s ~ t $\vee$ T.s ~ ∗·t) $\wedge$ |s| = |t·u| $\wedge$ u ≠ ε
$=$ { (8) with a, s, p := H.s, T.s, t }
(M Ξ s ~ ∗·t) $\wedge$ |s| = |t·u| $\wedge$ u ≠ ε

Now we have all the ingredients to spell out the missing part of FINISH.

```
SHRINK:
    { I ∧ s = t·u ∧ u ≠ ε ∧ p = *·t }
L:  { I ∧ p = *·t ∧ |s| = |t·u| }
    do u ≠ ε →
            { L ∧ u ≠ ε }
            s, u := T.s, T.u
            { L }
    od;
    { L ∧ u = ε }
    { I ∧ p = *·t ∧ |s| = |t| }
    p := T.p
    { I }
```

So not only does SHRINK re-establish **I** (for which a simple **s := T.s** would have sufficed), but it also optimizes away subsequent PUSH;FINISH cycles doomed to failure because of left-over symbols in one string. This completes the design of our program.

* * *

A few final remarks are in order:

(i) The program variable **t** (capturing the common prefix in FRAG8) is vital for the correctness proof but largely ignored by the program itself – except for the two maintenance assignments **t := ε** and **t := t·H.u**. Hence, for execution on a real machine, the variable and both assignments can be optimized away.

(ii) Another good candidate for an optimization is the point between **q := T.p** and PUSH in FRAG8. If q is empty there, PUSH has nothing to do, but it may take a while to SHRINK away the remaining sample. On the other hand, axiom (1) leaves no doubt that the final answer is "yes", so we may as well immediately set the flag **y**.

```
FRAG8:
    { I ∧ s ≠ ε ∧ H.p = * }
    u, q := s, T.p;
    { I ∧ p = *·q }
    if q = ε →
            { I ∧ p = * }
            { I ∧ M }
            y := true
    [] q ≠ ε →
            PUSH;
            FINISH
    fi
    { I }
```

(iii) Our initial assumption of star-free samples (formally captured by $* \notin \Sigma$) may be nice in theory but hard to enforce in practice. So let us postulate the existence of two "different" star symbols, together with the convention that a sample star never equals a pattern star. In fact, most of our program implicitly works that way due to special guards for pattern stars. However, the plain comparison **H.u** = **H.q** in the PUSH loop needs the extra conjunct **H.q ≠ *** to enforce inequality among two stars:

```
if  H.u = H.q  ∧  H.q ≠ *  →  u, q := T.u, T.q
[]  H.u ≠ H.q  ∨  H.q = *  →  d := true
fi
```

(iv) After elimination of **t** (which grows from **ε** to a new string of appended symbols), only the following string instructions remain in the program:

a) Check if string = ε
b) Extract H.string
c) Assign T.string to a variable

Thus, even the most primitive string representations – like pointers to zero-terminated memory chunks – are graciously accommodated:

a) Compare pointer target with zero
b) Read symbol from pointer target
c) Store incremented pointer value

(v) Despite our careful design of FINISH and SHRINK, there is still room to reduce the worst-case asymptotic running time if we resort to highly sophisticated string-matching algorithms, like the linear-time Knuth-Morris-Pratt solution. However, since our goal was an orderly design from first principles (without pulling any rabbits), such advanced optimizations exceed the scope of this work.

Acknowledgments
Although I never met Dijkstra in person, I got a faithful exposure to his ideas thanks to Edgar Knapp, who had come from Austin to Purdue in 1992, where I became his student one year later. Credit goes to my colleague Gunter Hick, whose lunchtime remark on a non-recursive four-pointer wildcard algorithm triggered my interest in the topic – and again to Edgar Knapp, as an inspiring brainstorming and review partner for its calculational treatment. (End of Acknowledgments.)

APPENDIX: The complete program expressed in "Guarded C"

```
#include <stdio.h>
#include <stdbool.h>
#include <assert.h>

#define SKIP  {}
#define ABORT assert (false)

#define IF if (
#define FI } else ABORT;

#define DO for (;;) { if (
#define OD } else break; }

#define THEN ) {
#define II   } else if (

bool  empty (char* string) { return *string == 0; }
char  head  (char* string) { return *string;      }
char* tail  (char* string) { return string + 1;   }

main (int argc, char* argv [])
{
    char* s = argv [1];
    char* p = argv [2];

    bool y = false;
    bool n = false;

    DO y == n THEN
        IF empty (p) THEN
            IF empty (s) THEN y = true;
            II!empty (s) THEN n = true;
            FI
        II!empty (p) THEN
            IF head (p) != '*' THEN
                IF empty (s) THEN n = true;
                II!empty (s) THEN
                    IF head (s) == head (p) THEN s = tail (s); p = tail (p);
                    II head (s) != head (p) THEN n = true;
                    FI
                FI
            II head (p) == '*' THEN
                IF empty (s) THEN p = tail (p);
                II!empty (s) THEN
                    char* u = s; char* q = tail (p);
                    IF empty (q) THEN y = true;
                    II!empty (q) THEN
                        bool d = false;
                        DO !empty (u) && !empty (q) && !d THEN
                            IF head (u) == head (q) && head (q) != '*' THEN
                                u = tail (u); q = tail (q);
                            II head (u) != head (q) || head (q) == '*' THEN
                                d = true;
                            FI
                        OD
                        IF empty (q) THEN
                            IF empty (u) THEN y = true;
                            II!empty (u) THEN
                                DO !empty (u) THEN
                                    s = tail (s); u = tail (u);
                                OD
                                p = tail (p);
                            FI
                        II!empty (q) THEN
                            IF head (q) != '*' THEN
                                IF empty (u) THEN n = true;
                                II!empty (u) THEN s = tail (s);
                                FI
                            II head (q) == '*' THEN s = u; p = q;
                            FI
                        FI
                    FI
                FI
            FI
        FI
    OD
    if (y) printf ("yes\n");
    if (n) printf ("no\n");
}
```