Formális szoftverfejlesztés

Ada-Spark tutorial és AtelierB összehasonlítás

Györgyi Csaba
gycsaba96@gmail.com

Tartalomjegyzék

Bevezető

Motiváció

A célunkat talán leginkább a „felelősség” szóval lehetne jellemezni. Képzeljük el azt az esetet, amikor egy – a kezünk közül kiadott – szoftverért mi vagyunk a felelősek, és az üzembe helyezés után is nyugodtan kívánunk aludni. Nagyon gyakran üzleti alapon bevállaljuk a hibás program lehetőségét, mert a hiba utólagos kezelése sokkal olcsóbb, mint annak javítása. Bizonyos esetekben nincs meg ez lehetőség. Ezekben az esetekben az alapos tesztelés mellett szeretnénk, ha a matematika is mellettünk állna, és garantálná a programunk helyességét, amelyért így nyugodt szívvel vállalhatunk felelősséget.

Mielőtt szentnek éreznénk magunkat, angyaloknak, akik a helyesség-bizonyíts által a bugos földi világ felé emelkednek, jegyezzük meg, hogy a formális bizonyításhoz formális specifikáció szükséges, ahol a matematikai garancia nem terjed ki az informális specifikációnak történő leírásnak.

Egyetemi tanulmányaink során számtalanszor tanultunk a helyesség-bizonyított algoritmusok fontosságáról és szerepéről. Ugyan végeztünk formális bizonyításokat, azonban ritkán készült a formális specifikációból összetettebb ténylegesen futtatható gépi kód. Ha ma úgy döntenénk, hogy a továbbiakban csakis formálisan bizonyított kódot vagyunk hajlandóak kiadni a kezünk közül, akkor nincs olyan eszköz, amely a segítene minket a formális specifikációtól kiindulva egy futtatható kód elkészítését.

Ezen igen időigényes és izgalmas kérdések hagyta űr kitöltését hivatott szolgálni ez a tutorial, amely több keretrendszer szemszögéből is megvizsgálja ugyan azt a valósághű feladatot.

A korrektség kedvéért kiemeljük, hogy tanultunk papíron elvégezhető bizonyításokat, de ezen esetekben egy elég absztrakt, általános algoritmusleírás szemantikáját használtuk. Ezzel a megközelítéssel az a problémánk, hogy könnyen elronthatjuk a papiros bizonyítást, vagy az implementációnál figyelmen kívül hagyjuk az absztrakt algoritmus és a programozási nyelv szemantikái közötti apró eltéréseket, amelyek hibás működést eredményezhetnek.

Feladatleírás

A későbbiekben vizsgált és megoldott feladat a egy termosztát komponens elkészítése. A egység feladata, hogy a folyamatosan érkező hőmérsékletadatokat eltárolja, de legfeljebb 10-et. Ha új elem érkezik, de már 10 elem el van tárolva, akkor mindig a legrégebbi kerüljön törlésre. Legyen képes a komponens az eltárolt időintervallum elemeire visszaemlékezni az alapján, hogy hány időegységgel korábbi adatot kívánunk felidézni.

Megállapíthatjuk, hogy ez leginkább egy window adatszerkezet.

Az egyszerűség kedvéért tegyük fel, hogy a hőmérsékletértékek -100 és +100 közötti egészekként vannak reprezentálva beleértve a végpontokat is.

Szerkezet

A továbbiakban két keretrendszer eszköztárán keresztül vizsgáljuk a korábban bemutatott feladatot. Az első esetben az Ada-Spark rendszerébe vezetjük be a kedves olvasót egy tutorialon keresztül. Ezt követően ugyanerre a feladatra adott megoldásunkat mutatjuk be az AtelierB keretrendszerében kiemelve a fontosabb különbségeket és tapasztalatainkat.

Minden egyes rész végén közlünk egy-egy linkgyűjteményt, amelyek tartalma további segítséget és információt tartalmaz a jobb és könnyebb megértés elősegítése érdekében. A helyesség-bizonyított szoftverfejlesztésnek igen kicsi a publikus közössége, amely nehézzé teszi az érdeklődőknek ebbe a világba történő betekintést. Ebből az okból kifolyólag egy-egy jó linkgyűjtemény igazi kincset ér.

Ada-Spark tutorial

Főbb alapgondolatok

Az Spark nem az Ada nyelv egy szintaktikus leszűkítése és szemantikus kiterjesztése a helyességbizonyítás eszköztárával.

Az eredeti Ada nyelv bizonyos elemei túl bonyolulttá és nehezen bizonyíthatóvá teszik a programot és annak elemezhetőségét, ilyen például a goto utasítás és a mellékhatásos kifejezések, függvények, ezért ezek használata nem megengedett a Sparkban.

Az Adához képest kiterjesztés a bizonyítandó szerződések lehetősége. Ezeknek több fajtája van, de mi a következőeket fogjuk intenzíven használni:

- **Pre.** Megadja az alprogram előfeltételét. A függvény vagy utasítás csak akkor használható, ha ez a feltétel teljesül.
- **Post**. Megadja az alprogram utófeltételét. Az eljárás meghívása után ezen feltételek fennállását kell garantálni.
- **Global.** Megadja, hogy a package szinten globális változók közül, melyekhez és milyen jogosultsággal férhet hozzá az alprogram a futása során. Meg lehet adni, hogy mely változók bemenetiek, kimenetiek, illetve melyikeket szabad írni és olvasni is.

Ezek alapján egy paritást ellenőrző függvény deklarációja pozitív egészekre a következőképpen nézhet ki:

```
function Even(n: in Integer) return Boolean
     with
           Global => null,
           Pre => (n>0),
           Post => (Even'Result = (n mod 2 = 0) );
```

Ezekben a szerződésekben előírjuk, hogy a package globális változóiból semelyikhez sem férhetünk hozzá, a bemenetünk kötelezően pozitív (Mj.: Ezt más bemeneti típussal is elérhettük volna.), illetve, hogy az eredmény a bemenet párosságával kell hogy ekvivalens legyen.

A szerződéseknek a betartását formális bizonyítással kell alátámasztani. Ennek megfelelően a fenti egyszerű példánkban garantálni kell, hogy az implementáció az előfeltétel teljesülése esetén a megadott adatokból olyan eredményt állít elő, amely megfelel az utófeltételnek. Másrészt a függvény használatakor a meghívás helyén bizonyítanunk kell, hogy az átadott paraméter megfelel az előfeltétel követelményeinek, majd ezt követően az utófeltétel állítását felhasználhatjuk a további programrészeink bizonyításánál.

A bizonyítandó állításokat a keretrendszer generálja automatikusan. A programozó segítésére automatikus bizonyítók ezeket megpróbálják bebizonyítani kisebb nagyobb sikerrel. Ha egy automatikus bizonyítás sikertelen, akkor kézi bizonyítás szükséges. Erre több lehetőségünk van:

- **Beépített interaktív bizonyító használata.** Alapból rendelkezésre áll, azonban a használata nem túl kényelmes.
- **A bizonyítandó állítás belátása külső interaktív bizonyítóval.** Ilyen lehet akár a más tárgyainkból megismert Coq is. Ennek telepítése kissé körülményes, illetve ha sikerrel is járunk, a bizonyítandó állításainkhoz olyan mennyiségű új definíció és axióma fog tartozni, amelyek megismerése egy kezdő szinten túlságosan időigényes lenne.
- **Assertek és ghost kód használata.** Ezekkel az eszközökkel tényleges hatással nem bíró utasításokat írhatunk a kódunkba, amelyek segítik az automatikus bizonyítók heurisztikáit, hogy mely részállításokat érdemes belátni.

A kínálkozó lehetőségek közül mi az utolsót fogjuk használni. Ennek egyik oka, hogy így nem kell egy új környezetet elsajátítanunk, elegendő a programozói megközelítés. Az AtelierB dokumentációja például kimondja, hogy a bizonyítás nem része a programozói tevékenységnek. Másik oka, hogy az első kettő lehetőség túlzottan időigényes, és minimális publikusan elérhető anyag van hozzá.

Döntésünknek köszönhetően a programozói tevékenységünket úgy is felfoghatjuk, hogy az automatikus bizonyítok „személyében” kaptunk egy kollégát, aki az apró, viszonylag egyszerű összefüggéseket képes meglátni. Hogy őt segítsük, és így ő is segítsen minket, nem kell mást tennünk, mint a kódba írni olyan plusz részeket, amelyek a kódgenerálásra nincsenek hatással, de őt nagy mértékben segítik.

Az egyik legfontosabb kulcsszó talán az egyszerűség. A Spark az egyszerűen specifikálható és implementálható programok belátásában tud a legnagyobb segítségünkre lenni.

Tutorialunk lényegét ezen egyszerűség és az automatikus bizonyítókkal való együttműködést bemutatása adja.

Hasznos linkek:

- a nyelv megszorításairól bővebben: https://docs.adacore.com/spark2014-docs/html/ug/en/source/language_restrictions.html
- a szerződésekről bővebben: https://docs.adacore.com/spark2014-docs/html/ug/en/source/subprogram_contracts.html

Ada-Spark telepítése

Az Ada-Spark keretrendszerét telepíthetjük saját gépünkre is, vagy használhatjuk az általam előre összerakott Docker konténert. Ez utóbbi a grafikus felület miatt csak linux rendszereken működik.

Telepítés saját gépre (Ubuntu):

1. Látogassunk el a közösség letöltő oldalára: https://www.adacore.com/download.

2. Töltsük le a rendszerünknek megfelelő Community csomagot. Ez tartalmazni fog minden számunkra fontos programot egyben összecsomagolva. Ez az állomány megközelítőleg 500MB.
3. Futtassuk a letöltött fájlt. Ez elvégzi a rendszer telepítését. Ezek után a szükséges futtatható állományok a default beállítások mellett elérhetőek a */opt/GNAT/2019/bin* könyvtéárban.
4. Szükség esetén adjuk hozzá a telepítési könyvtárat a PATH környezeti változónkhoz.

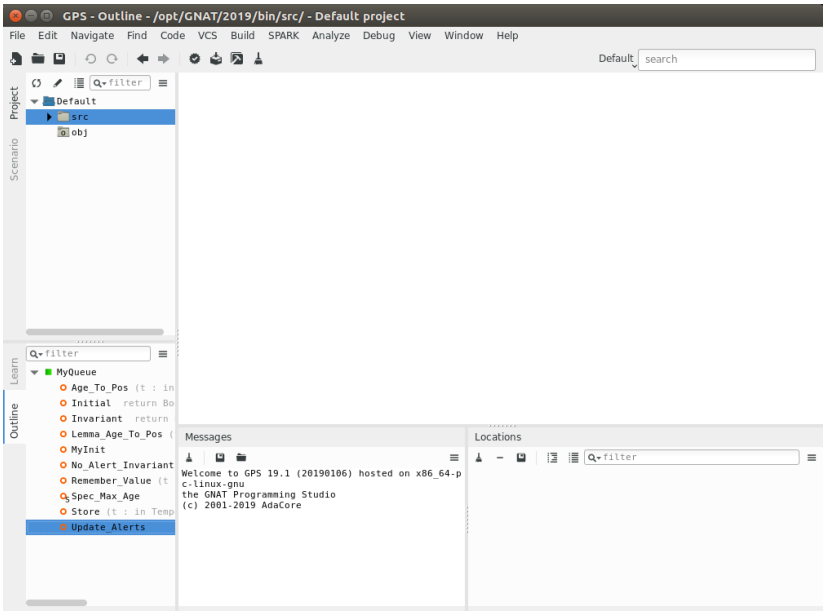
Hasznos linkek:

- hivatalos telepítési útmutató: https://docs.adacore.com/spark2014-docs/html/ug/en/install.html
- GNAT community letöltése: https://www.adacore.com/download
- az általam összerakott docker konténer (egyéb forrásokkal): https://ikelte-my.sharepoint.com/:f:/g/personal/gycsaba96_inf_elte_hu/EvHcjbsKM4pAhamnELfy5kgBen1BohCgMTbcap1mmg_8SQ?e=PK6D2b

Használat

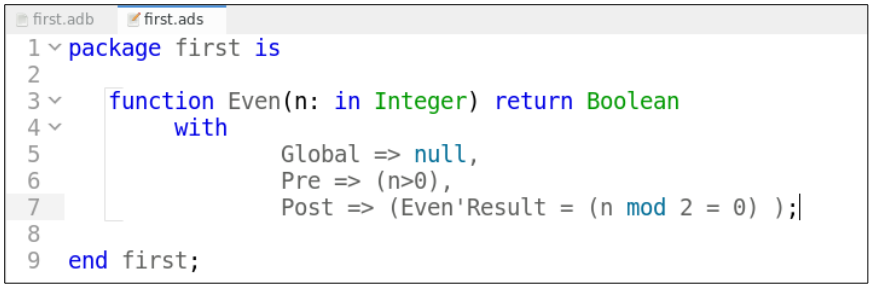
A következőekben egy igen egyszerű feladaton keresztül elsajátítjuk a környezet alap szintű használatát.

1. A fejlesztés során a GNAT Studiot fogjuk használni, amely az Ada révén már ismerős lehet számunkra. Ennek elindításához lépjünk a telepítési könyvtárba (*/opt/GNAT/2019/bin*), majd adjuk futtassuk a programot a *./gps* utasítással. Ekkor a következő ablaknak kell fogadnia minket:

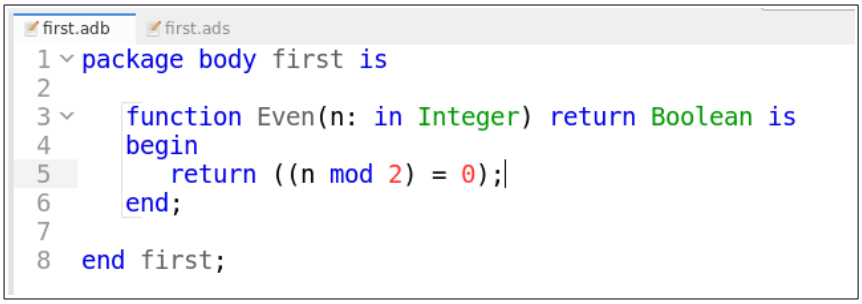


2. Hozzunk létre egy Ada csomagot *first* néven. Jobb-klikk az *src* mappán > New > Ada Package.

3. Másoljuk be a korábban látott paritást eldöntő függvényünk deklarációját a specifikációs fájlba (first.ads).



4. Az implementációba pedig az alábbi megoldást:



5. Jelenleg egy teljesen valid Ada kódunk van. Hogy ki tudjuk aknázni a Spark eszköztárát a csomagunkban és az implementációjában is be kell kapcsolnunk a spark módot a „*with SPARK_Mode => On*” aspecttel. A teljes kód tehát a következő:

first.ads:

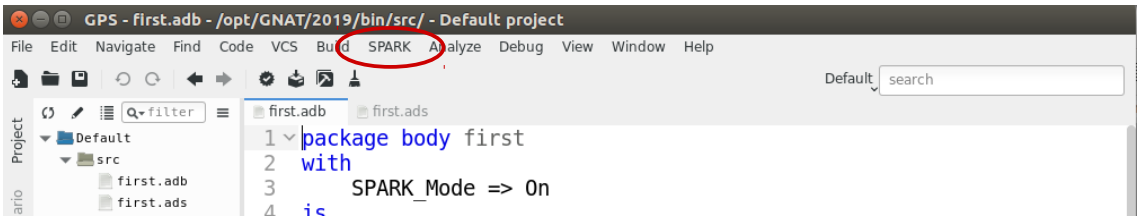
```
package first
with
    SPARK_Mode => On
is

   function Even(n: in Integer) return Boolean
     with
           Global => null,
           Pre => (n>0),
           Post => (Even'Result = (n mod 2 = 0) );

end first;
```

first.adb:

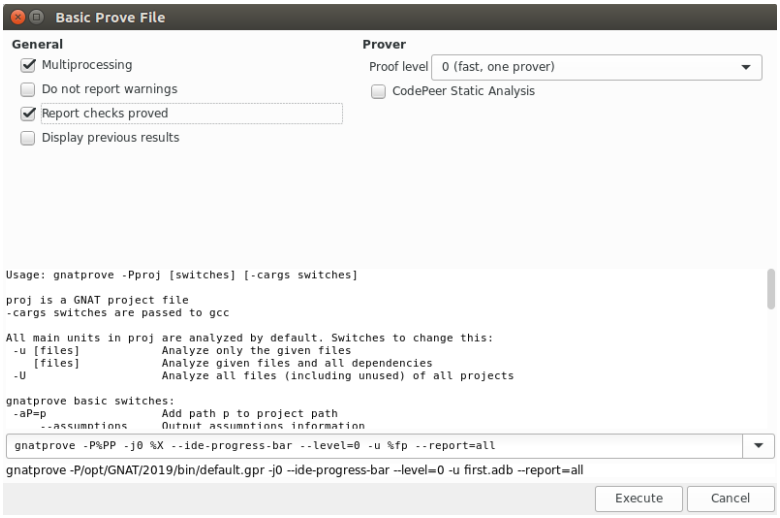
```
package body first
with
    SPARK_Mode => On
is

   function Even(n: in Integer) return Boolean is
   begin
      return ((n mod 2) = 0);
   end;

end first;
```

Megjegyzés: *vegyük észre, hogy a Spark ilyen fajta integrációja nagyban megkönnyíti a nem bizonyított részekkel való együttműködést. Lehetőségünk van arra is, hogy egy csomagnak csak a specifikációjában használjuk a Spark módot, és az implementáció helyességét nem ellenőrizzük. Ez lehetőséget nyújt számunkra a szerződésalapú programozás szabályainak szigorúbb és megengedőbb ellenőrzésére.*

6. A Spark elenőrzőit A *SPARK* > *Prove File* menüponttal használhatjuk a kiválasztott csomagon.

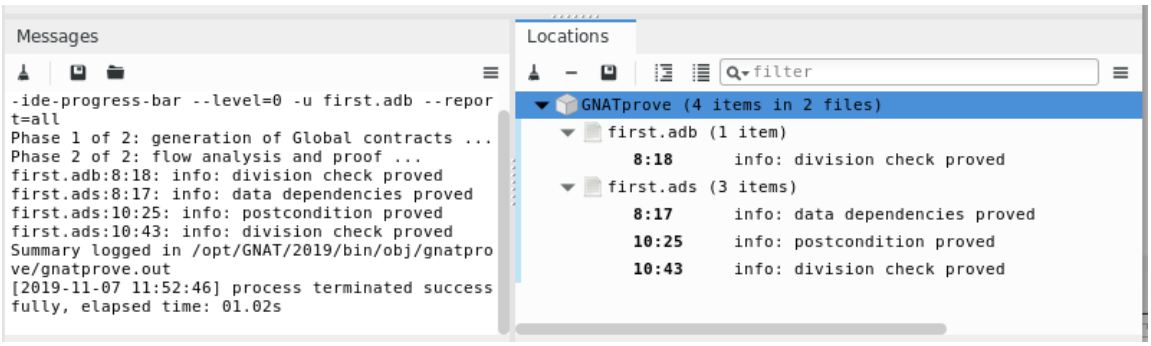


Ezt a menüpontot kiválasztva felugrik egy ablak, amelyben az automatikus bizonyítókat konfigurálhatjuk. Itt néha érdemes a *„Report checks proved”* opciót bekattintani, ez kiírja majd nekünk a vizsgálat eredményét egy lista formájában is. A másik opció amire a későbbiek során szükségünk lesz, a *„Proof level”* kiválasztása. Ezzel lehetőségünk van több bizonyítót is használni egyre erősebb heurisztikákkal és egyre hosszabb megengedett futási időkkel. Megjegyzendő, hogy az erősebb heurisztikák ugyan bonyolultabb összefüggéseket is megtalálnak, de sokkal könnyebben tévednek el a bizonyításban, és futnak timeoutra.



Az *Execute* gombbal indítsuk el a bizonyítókat.

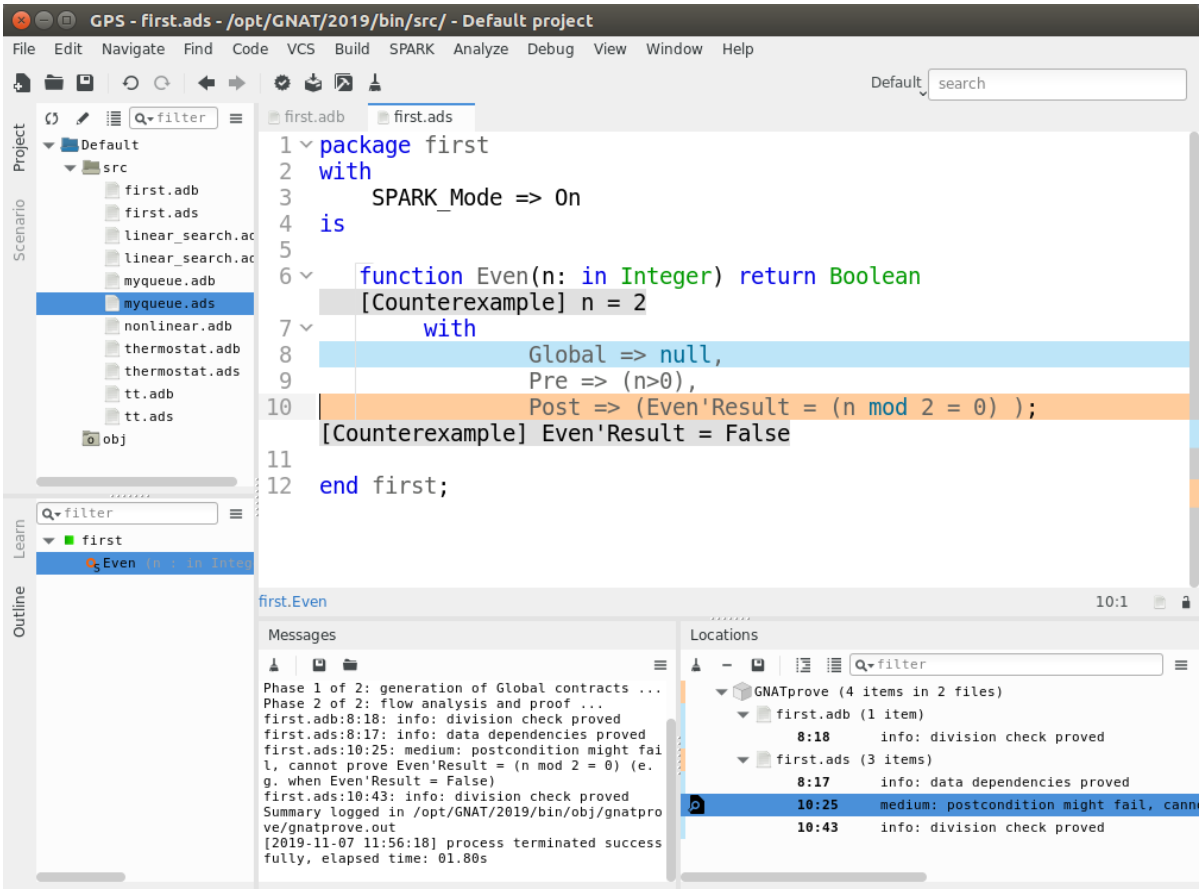
7. Az ablak bal alsó sarkában láthatjuk hogy minden bizonyítandó állításunk sikeresen belátásra került.



Ezzel készen is van az első bizonyított függvényünk, ha nem is túl bonyolult.

Kiegészítés

Most próbáljunk meg szándékosan hibát véteni. Az implementációban vizsgáljuk 0 helyett az 1-gyel való egyenlőségét a maradéknak.



Ekkor az automatikus bizonyítás a várakozásainknak megfelelően sikertelen, de ezen felül ellenpéldát is megpróbál számunkra szolgáltatni. Ha a sikertelen bizonyítás melletti nagyító ikonra kattintunk, akkor ezt a példát még a kódunkba is behelyettesíti.

Hasznos linkek:

- egy hosszabb tutorial a lineáris keresésre: https://docs.adacore.com/spark2014-docs/html/ug/en/tutorial.html
- még több példa és magyarázat: https://learn.adacore.com/courses/intro-to-spark/index.html

Feladatmegoldás menete

Elképzelés

A megoldásunk igen egyszerű lesz. Egy 10 elemű tömbben tároljuk majd az elemeket. Felveszünk egy mutatót, amiben mindig a következő felülírandó tömbelemre mutat.

Kezdetben ez a mutató a 0. elemre mutat, és amikor beteszünk egy elemet növeljük az értékét. Ha Már túlfutnánk a tömbön, visszaállítjuk a 0-ra.

Ennek a ciklikus mozgásnak köszönhetően ha felülírunk egy régi elemet, az biztosan a legrégebbi lesz.

Ezt az algoritmust humán szemmel helyesnek gondoljuk, de egy formális bizonyítóval mint tapasztalni fogjuk sokkal több érvre lesz szükségünk.

Saját típusok és altípusok

Első lépésben vezessük be a saját típusainkat és altípusainkat, amelyekkel majd jól kifejezhetjük a feladatunk tulajdonságait és alkotóelemeit.

```
-- *** TYPES ***

subtype Temperature is Integer range -100 .. 100;
type Array_Dom is mod 10;
type Time_Dom is mod 10;
type Temperature_Array is array(Array_Dom) of Temperature;
type Validity_Array is array(Array_Dom) of Boolean;
type Age_Array is array(Array_Dom) of Time_Dom;
```

A hőmérsékleteinket korlátozzuk be -100 és +100 közé. Ez azért fontos, hogy ha később össze szeretnénk adni két hőmérsékletértéket, be tudjuk bizonyítani, hogy nem lesz túlcsordulás.

Fontos döntés, hogy a tömbjeink és az időintervallumaink típusát moduló típusoknak vesszük fel. Ennek köszönhetően a 9. elem után nem kell nekünk kiírni majd a modulókat a kódban, ráadásul a bizonyító is jobban ismeri ezeket a típusokat, és készségesebb lesz az állítások belátásában, mintha a moduló tulajdonságait is be akarnánk vele láttatni.

A specifikációban az van előírva, hogy mindig a legrégebbi elemet kell kitörölnünk. Az egyes értékek korát egy `Age_Array` típusú tömbbel fogjuk majd jellemezni.

A package állapota

Ahogyan azt Adában megszokhattuk, a csomagunknak lehetnek sajt változói, amelyek egy belső állapotot képesek így nyilvántartani.

Ezeket normál esetben illene priváttá tenni, de hogy egyszerűbb legyen a specifikáció megfogalmazása ezeket publikusan hagyjuk.

```
   -- *** VARIABLES (they should be hidden) ***

   Temperatures : Temperature_Array := ( others => 0 );
   Value_Count : Integer := 0;
   Next_Position : Array_Dom := 0;
   Ages : Age_Array := ( 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 ) with  Ghost;
```

A `Temperatures` tömbben tároljuk a hőmérsékletértékeinket, a `Value_Count` és `Next_Position` változók rendre a tárolt elemek számát és a következő felülírandó elem sorszámát tartalmazza. Az `Ages` tömb, pedig az egyes értékekhez tartozó kort tárolja, amelyeket viszont a program futása során nem kívánunk valóban nyilván tartani, csak a bizonyítást és a specifikációt megírását segíti, ennek megfelelően úgynevezett „ghost” változónak jelöljük.

Ezeket az állapotot leíró változókat egyből inicializáljuk is, hogy használatra készek legyenek. Az `Ages` tömb tartalmát azért egyesével csökkenő értékekként vettük fel – hiába nincsenek még eltárolt értékeink –, mert így a későbbiekben sokkal könnyebb lesz invariáns tulajdonságot mondanunk. A program futása során végig megfigyelhető lesz ez az egymásutániság, csak ciklikusan eltolva.

Hogy a Sparkot is tudassuk arról, hogy inicializálandó változóink vannak, a package-hez fel kell vennünk még pár aspectet, amelyek ezt kijelentik:

```
package MyQueue
with
  Abstract_State => State,
  Initializes    => (State, Temperatures, Ages, Value_Count, Next_Position),
  SPARK_Mode => On
is  …
```

A State absztrakt állapotot konvencionálisan felvettük, összetettebb programok esetén a csomagok együttdolgozását segítené. Ennek következtében viszont a package body-t is ki kell egészítenünk:

```
package body MyQueue
with
  SPARK_Mode => On,
  Refined_State => ( State => null )
is …
```

A package állapotához kapcsolódóan állításokat és segédfüggvényeket is megfogalmazhatunk ghost eljárások és függvények formájában.

Az állapotot jellemző `Initial` és `Invariant` hasznos Boolean függvényeink írják majd le a package invariánsát és kezdeti feltételét. Ezek segítségével könnyebb lesz definiálnunk a termosztátunk elő és utófeltételeit.

A `Spec_Max_Age` eljárás törzse teljesen üres, csak a példa kedvéért szerepel. Lényegében egy bizonyos feltétel alapján képes (egy triviális) következtetést levonni. Ezt az eljárást meghívva, be tudnánk látni az utófeltétel állításának igaz voltát egy összetettebb alprogramban. Ezek az eljárások lényegében lemmaként szolgálnak a bizonyításainknál.

```
procedure Spec_Max_Age with
    Ghost,
    Pre => Invariant,
    Post => (for all i in Ages'Range => Integer(Ages(i))<10) ;


 function Initial return Boolean with Ghost;

 function Invariant return Boolean with Ghost;
```

Az invariáns és inicializálási feltételeink legyenek a következőek.

```
 function Initial return Boolean is
   (
        Next_Position = 0
      and then
        (Ages(0) = 9)
      and then
        (for all i in Array_Dom => (Ages(i) = Ages(i+1)+1 ))
   );

 function Invariant return Boolean is
   (
      (Value_Count<=10)
      and then
      (
         for all i in Array_Dom => ( Ages(i) = Ages(i+1)+1 )
      )
      and then
      (
         Ages(Next_Position)=9 --or Integer(Next_Position) = Value_Count)
      )

     );
```

Mint látni fogjuk, ezek kellően erős feltételek. Ezek a definíciók az Ada nyelvnek megfelelően a package törzsében kaptak helyet.

A speciális `Ages` inicializációnknak köszönhetően az invariánsunk egyszerűbb lehet. Ez azért is hasznos, mert az automata bizonyító a több lehetőség között könnyebben eltéved. A későbbiekben is törekedni fogunk az egyszerűségre éppen ezen okból kifolyólag.

Ez eddigi szerződéseket az automata bizonyító készséggel belátja. Most hogy definiáltuk az állapotunkat és annak tulajdonságait, nekiláthatunk a műveleteink specifikálásához és implementálásához.

Értékek eltárolása

Az egyik műveletünk a termosztáthoz az, hogy értékeket tudjunk eltárolni. Ezt a műveletet a következőképpen specifikálhatjuk:

```
procedure Store(t:Temperature) with
  Global => (
              In_Out => (Temperatures,Value_Count,Next_Position,Ages)
            ),
  Pre => Invariant,
  Post => (  Invariant
           and
             (if Value_Count'Old<10 then Value_Count=Value_Count'Old+1)
           and
             ( Ages(Next_Position'Old) = 0)
           and
             ( for all i in Array_Dom =>
                  (
                       Integer(i) >= Value_Count'Old
                       or
                       (Ages(i) = Ages'Old(i)+1)
                  )
              )
           and
             (
               for all i in Array_Dom =>
                  (
                       i=Next_Position'Old
                       or
                       Temperatures'Old(i) = Temperatures(i)
                  )
             )
          );
```

A Global részben írási és olvasási jogot adunk a megfelelő változókhoz. Az előfeltételben csak az invariáns teljesülését követeljük meg, ezzel biztosítva, hogy rendeltetésszerű használat esetén bármikor meghívható a művelet. Az utófeltételben a az invariáns további teljesülésén túl rendre a következőeket követeljük meg:

- Ha még nincs tele a tömbünk, az elemszám növekedjen eggyel.
- Az újonnan betett elem kora 0 legyen .
- Minden korábbi érték kora eggyel növekedjen.
- Az újonnan eltárolt értéken kívül egyik másik se változzon meg.

Ez igen sok feltétel. Új feladat esetében érdemes inkrementálisan az egyes utófeltételt szigorítva írni a kódot. Az az eljárás egy bizonyíthatóan helyes implementációja a következő:

```
procedure Store(t:Temperature) is
begin
   if Value_Count<10 then
      Value_Count := Value_Count+1;
   end if;

   Temperatures(Next_Position):=t;
   Next_Position := Next_Position+1;

   for i in Array_Dom loop
      Ages(i) := Ages(i) + 1;
   end loop;
end;
```

Az egyetlen helyességbizonyításhoz szükséges plusz erőfeszítés az eljárás végi ciklus, amiben az értékek korát nyilvántartó ghost változóként deklarált tömböt frissítjük. Ez ránézésre lineárissá teszi az amúgy konstans futási időnket, de ez csak a látszat. Mivel ghost változóról van szó ezért tényleges kód nem generálódik ehhez a részhez.

A bizonyító egy picivel magasabb fokozaton képes belátni az implementáció helyességét.

Értékek visszaolvasása

Másik műveletünk az egyes értékek visszaolvasása. Ennek specifikációja és implementációja sokkal egyszerűbb, helyeségének belátása viszont annál trükkösebb.

Specifikáció

A műveletet eljárásként a következőképpen specifikálhatjuk.

```
procedure Remember_Value(t:in Time_Dom;v:out Temperature) with
  Global => (
                Input => (Temperatures,Next_Position),
                Proof_In => (Ages,Value_Count),
                Output => (e_place)
            ),
  Pre => (
             Invariant
             and
             Integer(t)<Value_Count
          ),
  Post => (
              Invariant
              and
              v = Temperatures(e_place)
              and
              Ages(e_place)=t
           );
```

Az algoritmus csak a hőmérsékletek tömbjét és a következő üres pozíciót használhatja fel a számításaihoz a csomag globális változóiból, az egyes elemek korát és az értékek számát pedig csak a bizonyításhoz használhatjuk fel, de az értékeket nem módosíthatjuk.

Az előfeltételben az invariánson kívül megköveteljük, hogy csak olyan létező időpillanat értékét kérhessük le.

Az utófeltételünket valahogyan úgy fogalmazhatnánk meg, hogy létezik egy olyan érték, aminek a kora a megadott `t` érték, és az ehhez tartozó hőmérséklettel legyen egyenlő a `v` kimeneti paraméter. Ezt egy egzisztenciális kvantor használatát sugalmazza, azonban annak nem csak a használata lenne körülményes jelen esetben, hanem a bizonyítása is sokkal nehezebb. Ennek kikerülésére felvettünk egy újabb globális ghost változó `e_place` néven, és csak írási jogot adunk rá, ezzel elérve, hogy a létezik kifejezésünk bizonyítása helyett egyből egy tanút adjunk. Cserébe annyi árat fizettünk, hogy így a műveletünknek mellékhatása lett, és így a Spark szabályai érdekében csak eljárásként definiálható.

Implementáció

Az implementációra egyik ötletünk lehet egy lineáris keresés. Ez azonban egyrészt kevésbé lenne hatékony, ugyanis lineáris futási idejű megoldás is adható. Másrészt a korokat egy ghost változt (`Ages`) kellene effektíven, nem csak a bizonyításhoz használnunk.

Az adatszerkezetünk működését és a invariánsban is megfogalmazott korok szerinti ciklikus rendezettséget kiszámolva egy egyszerű modulós képlettel meghatározható a kívánt érték. (A megfelelő mod típus választásának köszönhetően ez a moduló nem kerül kiírásra.)

```
procedure Remember_Value(t:in Time_Dom;v:out Temperature) is
begin
   e_place := Next_Position-Array_Dom(t)-1;
   v := Temperatures(Next_Position-Array_Dom(t)-1);
end;
```

A képletünk ugyan egyszerű, de az automata bizonyító nem képes belátni segítség nélkül. Ennek megoldására definiálni fogunk egy segédlemmát, amely az időpillanatok és az egyes tömbelemek közötti kapcsolatot adja meg a `Next_Position` változó segítségével. Ez a segédtétel lényegében a képletünk helyességét fogalmazza meg.

```
procedure Lemma_Age_To_Pos(t:in Time_Dom;idx: out Array_Dom) with
  Ghost,
  Pre => (Invariant),
  Post => ( Ages(idx) = t and idx = Next_Position-Array_Dom(t)-1);
```

Ennek bizonyítása után valószínűleg az elnevezés miatt az automata bizonyító képes belátni az implementációnk helyességét. (Ha erre mégse lenne képes, mindössze annyit kellene tennünk, hogy az implementációnkban meghívjuk `t` és `e_place` paraméterekkel.)

Segédlemma bizonyítása

A bizonyítást meglepőmódon egy algoritmus implementálásával adjuk meg. Az eljárásunknak egy olyan törzset adunk meg, amely miközben egy lineáris kereséshez hasonlóan megtalálja a megfelelő korú elem indexét, egyben az arra adható egyszerűbb képletet is belátja.

Mivel egy ilyen keresés önmagában elég bonyolult, ezért különböző `Assert` utasításokat helyezünk el a kódunkban. Ezek részcélok az automata bizonyítónak, amelyek belátására kötelezzük, s a későbbiekben ezeket megpróbálja felhasználni.

```
procedure Lemma_Age_To_Pos(t:in Time_Dom;idx: out Array_Dom) is
   guess : Array_Dom := Next_Position;
begin
   pragma Assert( Ages(guess)=9 and guess = Next_Position);
   for i in 0..t loop
      guess := guess - 1;
      pragma Assert (
                     Ages(guess) = i
                     and
                     guess = Next_Position-Array_Dom(i)-1);
   end loop;
   pragma Assert (guess = Next_Position-Array_Dom(t)-1);
   idx := guess;
end;
```

A keresést mindig a következő felülírandó pozícióval kezdjük, amelynek mint tudjuk a kora már 9. Nyilván az előző elem a 0 korú, és ha tudjuk egy elemről hogy annak a kora *k*, akkor az előtte lévőnek biztosan *(k+1) mod 10*. Ezzel a módszerrel eliterálva `t`-ig, megkapjuk a bizonyítandó állítást.

A munkánkat ismét nagyban megkönnyítette a megfelelő mod típus használata.

Az assertekkel való bizonyítás nem csak egyszerűbben kivitelezhetőbb, mint egy interaktív prover használatának elsajátítása, de a programozót is segíti a saját gondolatainak leellenőrzésében. Ráadásul így csak az automatabizonyítóval képesek vagyunk belátni a lemmánk helyességét.

Utolsó simítások

A kódunkat kiegészítettük egy `Init` eljárással is, amivel alaphelyzetbe lehet hozni a termosztátunkat. Ezen kívül írtunk egy főprogramot (main.adb), ami a megírt csomagunkat használja, és így a kódunk futtatható és kipróbálható.

AtelierB összehasonlítás

A következőekben a feladatunk AtelierB-ben történő implementációját és bizonyítását vetjük össze a Sparkos megoldásunkkal. Tekintve, hogy az AtelierB elsajátítása sokkal több időt vesz igénybe alapszinten is, ezért ez a rész kevésbé lesz tutorial jellegű, inkább csak az elgondolásbeli különbségekre hívja fel a figyelmet.

Főbb alapgondolatok

Az AtelierB az úgynevezett B-módszerhez készített fejlesztőkörnyezet. A B módszertanának egyik lényege, hogy egy formális specifikációból kiindulva a addig „finomítjuk” a kódot, amíg az egyszerű és lefordítható utasításokká nem alakul. A specifikáció és a finomítások konzisztenciáját bizonyítanunk kell.

Akárcsak a Sparkban itt is segítségünkre van egy automatabizonyító, amelynek képességei itt is igen korlátosak. Bizonyos esetekben már a specifikáció írása során is szükségünk lehet emberi beavatkozásra a bizonyításokhoz. Tekintve azonban, hogy csak specifikációt írunk a fejlesztés elején, a ghost kódok és assert pragmák használata eleve kizárt. Megoldásunkban így kénytelenek leszünk az interaktív bizonyítót is igénybe venni.

A B-nyelvben az úgynevezett gépek alkotják az enkapszuláció alapegységét. Ezt leginkább az Ada nyelv package konstrukciójának lehetne megfeleltetni. A gépeket számtalan tulajdonságukkal adjuk meg, ezek közül a számunkra relevánsak a következőek:

- **VARIABLES:** a gép belső állapotát leíró változók,
- **INVARIANT:** a gép invariánsa gép változóival és logikai kifejezésnyelvvel leírva,
- **INITIALISATION:** a gép változóinak kezdeti feltöltése, ennek már ki kell elégítenie az invariánst,
- **OPERATIONS:** a gép műveletei, amelyek állapotváltozást idézhetnek elő, bizonyítanunk kell, hogy az invariánst megőrzik,
- **ASSERTIONS:** az invariánsból következő segédállítások.

A változók a specifikációban először csak absztraktak, és tovább finomíthatóak konkrét változókká, vagy egyéb kifejezésekké.

A műveletek által okozott változást nem elő és utófeltételükkel adjuk meg, mint a Sparkban, hanem a hatásukat írjuk le úgynevezett helyettesítési szabályokkal. Tekintsük az alábbi műveletet:

```
inc(n) =
PRE
      n:NAT
      &
      n mod 2 = 0
BEGIN
      x := x+1
END
```

Ebben az esetben be kell látnunk, hogy ha az invariáns és a megadott előfeltételek teljesülnek , akkor az invariánsban `x`-et `x+1`-re helyettesítve továbbra is igaz állítást kapunk. Ebből gondolhatjuk, hogy minél összetettebb az invariánsunk, potenciálisan annál több dolgot kell belátnunk.

Az invariánsból bizonyos állítások kiszervezhetőek az ASSERTATIONS részbe, és ekkor ezek ugyan úgy felhasználhatóak a bizonyításnál információként, mint az invariáns.

Elképzelés

A feladatunk és rá adott megoldásunk változatlan. A könnyebb összehasonlíthatóság kedvéért pont ugyan úgy specifikáljuk a feladatunkat.

Specifikáció

Mint azt korábban említettük egy absztrakt gépet kell megadnunk specifikáció gyanánt.

Saját definíciók

A saját feladat specifikus fogalmainkat DEFINITIONS részben adhatjuk meg. Ezek a C/C++-hoz hasonlóan csak egyszerű szöveges behelyettesítést eredményeznek még a kód értelmezése előtt.

```
DEFINITIONS
      TEMPERATURE == -100..100 ;
      TIME_DOM == 0..9;
      ARRAY_DOM == 0..9
```

A gép állapota

Változóknak vegyük fel ugyan azokat, mint a Sparkos megoldásban. Ezek mivel a specifikációban vannak, alapból mindegyikük absztrakt, nem kell külön megjelölnünk ghost változókat, hiszen még semmi sincs implementálva.

A változóink annyira absztraktak, hogy a típusukat is az invariánsban kell megfogalmaznunk.

```
VARIABLES
      temps, ages,
      next_pos, value_count
```

Teljesen elképzelhető, hogy egy változót úgy implementálunk a későbbiekben, hogy nem tartozik hozzá tényleges konkrét változó, hanem valahogyan a többi érték függvényében adjuk meg. Erre látni fogunk példát az `ages` változónk esetében.

A változók kezdeti értékére az INITIALISATION részben tehetünk megszorításokat. Ezt a szekciót mint egy „konstruktor” képzelhetjük el, és a hatásával definiáljuk. A minél nagyobb általánosság kedvéért párhuzamos értékadásokat használunk. Ez az AtelierB-ben bevett gyakorlat.

```
INITIALISATION
      next_pos := 0
      ||
      value_count := 0
      ||
      ages :: { ff | ff:ARRAY_DOM --> TIME_DOM
            & !(ii).(ii:ARRAY_DOM => ff(ii) = 9-((10 + ii -0) mod 10))
            & ff(0) = 9 }
      ||
      temps := (ARRAY_DOM)*{0}
```

Az `ages` függvényünket egy „legyen eleme” művelettel inicializáljuk. Olyan függvény legyen az értéke, amely utána az invariánsnak is eleget tesz. Könnyen belátható, hogy a lehetséges értékek halmazának számossága pontosan 1.

A `temps` függvényünket egy direktszorzatként adtuk meg.

Invariáns

Az invariánsunk tartalmazza egyrészt a változók típusát. Másrészt a már ismerős eösszefüggéseket a változóink között.

```
INVARIANT
      // *** TÍPUSOK ***
      (
            next_pos : ARRAY_DOM
            &
            value_count: 0..10
            &
            temps : ARRAY_DOM --> TEMPERATURE
            &
            ages : ARRAY_DOM --> TIME_DOM
      )
      &
      // *** EGYÉB MEGKÖTÉSEK ***
      (
            //!(ii).(ii:ARRAY_DOM => ages(ii) = (ages((ii+1) mod 10)+1) mod 10)
            //&
            ages(next_pos) = 9
            &
            !(ii).(ii:ARRAY_DOM => ages(ii) = 9-( (10+ii-next_pos) mod 10))
      )
```

A korábban tömb változóink most totális függvényekként vannak reprezentálva.

A változóink közötti összefüggéseknél vegyük észre, hogy átrendeztük kissé a lapjainkat. A Sparkos megoldásunkban volt egy ciklikusságot leíró invariánsunk (itt kikommentelve), illetve egy ebből következő lemmánk (itt az utolsó állítás), amely a korok és pozíciók kapcsolatáról beszél a következő pozíció függvényében. Ez akkor egy remek döntés volt, mivel ciklusokhoz remekül illeszkedett az a fajta tulajdonság, könnyű volt vele a bizonyítás. AtelierB-ben viszont látjuk majd, hogy sokkal kényelmesebb dolgunk van, ha egyből a segédállítást vesszük be az invariánsba.

Megjegyzés: elméletileg az invariánsunk lehetett volna a régi, és ekkor a segédállításunkat az ASSERTIONS részben vettük volna fel, ennek belátásához viszont csak az interaktív bizonyítót vehettük volna igénybe.

Értékek eltárolása

Előfeltételként csak annyit követelünk meg, hogy az eltárolandó érték a megfelelő tartományba essen.

A művelet hatását ANY-WHERE-THEN konstrukcióval adjuk meg. Az ANY részben lehetőségünk van új változó(k) megnevezésére, a WHERE részben az új és régi változóink közötti összefüggésekre tehetünk megszorításokat, végül a THEN részben már használhatjuk effektív módon az új vétozót/változókat.

```
store(nn) =
      PRE
            nn:TEMPERATURE
      THEN
            ANY new_ages
            WHERE
                  new_ages : ARRAY_DOM --> TIME_DOM
                  &
                  !(ii).(ii:(ARRAY_DOM)-{next_pos}
                        => new_ages(ii) = ages(ii)+1 )
                  &
                  new_ages(next_pos) = 0
                  &
                  new_ages((next_pos+1) mod 10) = 9
            THEN
                  ages := new_ages
                  ||
                  value_count := min({value_count+1,10})
                  ||
                  next_pos := (next_pos+1) mod 10
                  ||
                  temps(next_pos) := nn
            END
      END;
```

Lényegében azt írtuk elő a WHERE részben, hogy a régi értékek kora eggyel nőjön, az új érték kora pedig legyen 0.

Az ANY-WHERE-THEN egy kényelmes eszköz, nagy kifejezőerővel, de hátulütője, hogy ez a finomításokban és az implementációban egy „létezik”-es állítás belátását írja majd nekünk elő.

Érdekesség, hogy a fordító nem vizsgálja, hogy az előírt tulajdonságú változók egyáltalán léteznek-e. Ez kellemetlen módon csak a későbbi fázisokban derülhet ki. Hasonló a helyzet az előfeltétellel. Nem ellenőrzi a rendszer, hogy van-e olyan eset, amikor alkalmazható a művelet.

Értékek visszaolvasása

Hasonló eszközökkel definiáljuk az értékek visszaolvasását, mint az eltárolását.

```
vv <-- remember(tt) =
      PRE
            tt:TIME_DOM
            &
            tt<value_count
      THEN
            ANY pp
            WHERE
                  pp:ARRAY_DOM
                  &
                  ages(pp) = tt
            THEN
                  vv := temps(pp)
            END
      END
```

Ennél a műveletnél a megfelelő pozíció kifejezése talán sokkal természetesebb az emberi szemnek:

„A `vv` értéke egy azon a pozíción lévő hőmérséklet, amelynek a kora `tt`.”

Implementáció

Finomítás

Az AtelierB-ben az implementáció nem más mint a specifikáció egy olyan szintű finomítása, amely már gépi kódra fordítható. Az a finomítási lánc akár több elemet is tartalmazhatna, de a feladat egyszerűségére való tekintettel mi egyből implementáljuk a specifikációt.

Az implementáció ugyan úgy egy gép, ennek megfelelően vannak változói, azonban ezek már konkrét változók. A változók mellett ugyanúgy megadunk egy invariánst is. Ez az invariáns nem csak az újonnan deklarált konkrét változókhoz kötődő megszorításokat taglalja, de látja a specifikáció absztrakt változóit is, és a közöttük lévő kapcsolatot is megfogalmazza.

Az implementációnk lényegében ugyan az lesz, mint a Sparkos megoldásunk esetén.

```
CONCRETE_VARIABLES
      c_temps,
      c_next_pos, c_value_count

INVARIANT
      // *** TÍPUSOK ***
      (
            c_temps : ARRAY_DOM --> TEMPERATURE
            &
            c_next_pos : ARRAY_DOM
            &
            c_value_count : 0..10
      )
      &
      // *** KAPCSOLAT A SPECIFIKÁCIÓVAL ***
      (
            c_temps = temps
            &
            c_next_pos = next_pos
            &
            c_value_count = value_count
      )
```

Vegyük észre, hogy az `ages` absztrakt változóhoz nem hoztunk létre konkrét változót. Nem csoda, hiszen tényleg nem akarunk neki a valóságban helyet foglalni, és az értékeit eltárolni, hanem a korábban megadott képletünkre támaszkodva csak kiszámítani szükség esetén. Ezzel lényegében hasonló eredményt érünk el, mint a Spark ghost változóival. Fontos különbség azonban, hogy az operációkban ezekre a korábbi absztrakt változókra már nem hivatkozhatunk. Továbbá a bizonyító ki fogja kényszeríteni tőlünk az `ages` függvény kiszámítási módjának megadását.

Műveletek és inicializáció

Az inicializáció és a műveletek megvalósítása már csak a lényegi és effektív kódot tartalmazza. Nem csoda, hiszen minden egyes utasítását gépi kóddá kell tudnunk fordítani.

A megoldásunk lényegében ugyan az, mint a Spark esetében, csak hiányoznak a ghost kódok és assertek.

```
INITIALISATION
      c_next_pos := 0;
      c_value_count := 0;
      c_temps := (ARRAY_DOM)*{0}

OPERATIONS

store(nn) =
BEGIN
      c_temps(c_next_pos) := nn;
      c_value_count ←- inc_if(c_value_count);
      c_next_pos := (c_next_pos+1) mod 10
END;

vv <-- remember(tt) =
BEGIN
      // Next_Position-Array_Dom(t)-1
      vv := c_temps( (10+c_next_pos - tt - 1) mod 10 )
END
```

Az `inc_if` egy másik util gép művelete, amely maximum 10-ig növeli a változó értékét.

Az érdekesség kedvéért vessünk egy pillantást arra, milyen c kódot generál az AtelierB az implementációnkhoz.

```
#include "tm2.h"

/* Clause IMPORTS */
#include "utils.h"

/* Clause CONCRETE_CONSTANTS */
/* Basic constants */

/* Array and record constants */
/* Clause CONCRETE_VARIABLES */

static int32_t tm2__c_temps[10];
static int32_t tm2__c_next_pos;
static int32_t tm2__c_value_count;
/* Clause INITIALISATION */
```

```
void tm2__INITIALISATION(void)
{

    unsigned int i = 0;
    utils__INITIALISATION();
    tm2__c_next_pos = 0;
    tm2__c_value_count = 0;
    for(i = 0; i <= 9;i++)
    {
        tm2__c_temps[i] = 0;
    }
}

/* Clause OPERATIONS */

void tm2__store(int32_t nn)
{
    tm2__c_temps[tm2__c_next_pos] = nn;
    utils__inc_if(tm2__c_value_count, &tm2__c_value_count);
    tm2__c_next_pos = (tm2__c_next_pos+1) % 10;
}

void tm2__remember(int32_t tt, int32_t *vv)
{
    (*vv) = tm2__c_temps[(10+tm2__c_next_pos-tt-1) % 10];
}
```

Összefoglalás és tapasztalatok

Láthattuk ugyanannak a feladatmegoldásnak két különböző helyességbizonyító keretrendszerben történő bizonyítását.

Megállapíthatjuk, hogy az AtelierB és a Spark nyilván különböznek elgondolásmódjukban, és valószínűleg különböző esetekben lehetnek kényelmesebbek. A főbb megfigyeléseink a következőek:

- Mindkét keretrendszerben jó segíti az automata bizonyítóval való együttműködést, ha a feladatokat olyan apró részekre bontjuk, amennyire csak lehetséges.
- A Sparkban sokkal jobban összefonódik a bizonyítás és implementáció. A ghost változók használata talán természetesebb eszköz egy programozónak. Az AtelierB kifejezetten kijelenti, hogy a bizonyítás nem programozói tevékenység.
- A Spark specifikációk jobban illeszkednek a programozói nyelvhez, míg az AtelierB közelebb áll a logikai jelölésekhez.
- A Sparkkal sokkal könnyebb a bizonyított és nem bizonyított programkódok összekapcsolása, minthogy csak az Ada nyelv egy kiterjesztéséről beszélünk.
- Az AtelierB-ben a specifikáció finomításával éri el az implementáció részletességét, míg a Sparkban ghost kódokkal tudjuk egyszerűsítéseink helyességét igazolni.
- A Sparknál sokkal egyszerűbben kontrollálható, hogy mit kívánunk belátni formálisan és mit nem.

A Sparkhoz hasonló helyességbizonyító eszköz a Krakatau (http://krakatoa.lri.fr/) a Java nyelvhez írva.