Tartalomjegyzék

1. fejezet

Bevezetés

Napjainkban számos olyan számítástechnikai feladattal találkozhatunk, ahol egy hiba súlyos következményekkel járhat. Nemcsak ezeken a helyeken lenne jó azonban, ha matematikai eszközökkel tudnánk garantálni, hogy a felhasznált szoftver maradéktalanul megfelel a specifikációjának.

A programtulajdonságok matematikai bizonyítása azonban igen nehéz feladat, ezért ezt legtöbbször csak korlátozott tudású, specializált nyelvekre, vagy általános programnyelvek szűk részhalmazaira valósítják meg. A mindennapi életben előforduló feladatok megoldására viszont ezek a nyelvek nem alkalmasak.

Nagyon fontos eredmény, hogy létezik már olyan általános célú programnyelv is, amelynek egy jól használható részhalmazát jellemezhetjük elsőrendű logikai állításokkal: a funkcionális Clean nyelven írt programok tulajdonságait a Sparkle tételbizonyító segítségével bizonyíthatjuk. A Sparkle sem kezel egyelőre néhány fontos nyelvi elemet. Ilyenek a reaktív, a környezetükkel kölcsönhatásba lépő Clean folyamatok külső objektumokat képviselő kifejezései, és ide tartozik a Clean nyelv legújabb verziójának egyik újdonsága, a dinamikus kód is.

Egy fontos lépés lenne a reaktív Clean folyamatok helyességbizonyítása felé, ha azok úgy használhatnának dinamikus kódot, hogy a kóddal együtt megkapják azok jellemző programtulajdonságait is, amiket aztán felhasználhatnánk a folyamat saját tulajdonságainak bizonyításához.

Ennek támogatására elkészült egy általános architektúra, amely bármely dinamikus kódot támogató programnyelvhez illeszthető, és amely célja, hogy a dinamikus kódot és az ahhoz megfogalmazott és bebizonyított állításokat hatékonyan használhassuk fel úgy, hogy közben a biztonság ne sérüljön.

A jelen diplomamunka feladata az architektúra egy adott részének vizsgálata, a Clean programnyelvhez illesztése, és lehetőség szerint ahhoz egy prototípus készítése.

2. fejezet

A felhasznált eszközök

Ebben a fejezetben bemutatom a felhasznált eszközök, a Clean programnyelv és a Sparkle tételbizonyító rendszer azon tulajdonságait, amelyek a kitűzött feladat szempontjából fontos szerepet játszanak.

2.1. A Clean nyelv

A programozási nyelvek több fő kategóriába oszthatók. Az egyik ilyen kategória a funkcionális nyelveké. A funkcionális nyelvek központi fogalma, mint a kategória neve is mutatja, a matematikai értelemben vett függvény. Az ilyen nyelveken írt programok nem egy végrehajtandó utasítássorozatot, határoznak meg, hanem kiértékelendő kifejezéseket tartalmaznak. Egy kifejezés lehet egy függvény vagy egy függvény valamilyen alkalmazása aktuális paraméterekre.

A funkcionális is tovább csoportosíthatók. A Clean [1, 2] egy tisztán funkcionális nyelv. A változók ebben a nyelvben csak egyszer kaphatnak értéket, így többek között nem engedi meg a nyelv a „mellékhatásokat" sem. Ahhoz ugyanis, hogy egy függvény a rajta kívül álló, globális állapoton változtathasson, elengedhetetlen lenne az értékadás művelete. A Clean nyelv függvényeinek nagyon fontos tulajdonsága a „hivatkozási átláthatóság". Ez azt jelenti, hogy egy függvény ugyanazokkal a paraméterekkel hívva mindig ugyanazt az eredményt adja. Ez a tulajdonság teszi lehetővé, hogy egy függvényalkalmazás kiértékelésekor a függvény annak definíciójával legyen helyettesíthető, amiben a formális paraméterek minden előfordulását az aktuális paraméterekkel helyettesítjük. Szintén a hivatkozási átláthatóság eredménye, hogy megnyílik az út a helyességbizonyítás felé, hiszen így alapvető matematikai és logikai eszközöket használhatunk a programokról való érveléshez anélkül, hogy először a programhoz egy absztrakt és távoli matematikai modellt kelljen készítenünk.

A hivatkozási átláthatóság előfeltétele a Clean egy további lényeges tulajdonságának, a lusta kiértékelésnek is. Egy kifejezést csak akkor értékel ki a rendszer, ha annak az értékére valóban szükség van. Ez jelentősen növeli a nyelv kifejezőerejét, pld. végtelen hosszú listákat használhatunk programjainkban. Ha azonban egy kifejezés értékére biztosan szükségünk lesz, szigorú, mohó kirtékelést is előírhatunk rá.

A Clean nagy kifejezőerejéhez számos más jellemzője is hozzájárul. Ilyenek például az erős típusrendszere, magasabbrendű függvények használatának lehetősége, mintaillesztés a függvények formális paramétereiben, túlterhelhető nevek és a modulrendszer.

Egy általános célú programnyelvvel szemben természetes elvárás, hogy reaktív, a valós környezetével kölcsönhatásba lépő programokat írhassunk benne. Ehhez kezelni kell a program valós környezetének objektumait, pld. a lemezeket, nyomtatót, kijelzőt stb. is. A hivatkozási átláthatóság megtartásánál komoly akadályt jelent, hogy ezek az objektumok csak destruktív felülírással kezelhetők, így a nekik megfeleltetett programbeli struktúrák értékének is változnia kell. A Clean megoldása erre a problémára a unique, „egyedi" típusattribútum [3]. Egy unique típushoz tartozó változóra nem lehet egyszerre egynél több hivatkozás a Clean programot leíró kifejezésgráfban. Ezzel a korlátozással elérhető, hogy a unique típusú változók destruktív felülírása esetén is megmaradjon a hivatkozási átláthatóság.

A programoknak a környezetükön kívül szükségszerűen egymással is kommunikálniuk kell. A kommunikáció azonban nem feltétlenül csak adatcserét jelenthet. A Clean függvényeinek függvényeket is adhatunk paraméterül; ehhez hasonlóan jogos elvárás, hogy a programok is tudjanak kódot cserélni. A nyelv biztonságának megőrzéséhez azonban a típushelyességet is garantálni kell. Ahhoz, hogy ezt futási időben megkapott kódnál illetve adatoknál megtehessük, szükség van a dinamikus típusok [4] kezelésére, amit a 2.0-s verziójú Clean új, Dynamic típusával, valamint a dinamikus mintaillesztéssel valósít meg. Ez utóbbi azt jelenti, hogy a mintákban típusspecifikációt is megadhatunk, és egy Dynamic típusú objektum csak akkor illeszthető a mintára, ha a statikus típusa megegyezik a megkövetelttel. A Clean disztribúcióhoz tartozó StdDynamic könyvtár függvényeinek segítségével egy dinamikus típusú objektumot fájlba írhatunk, és onnan azt visszaolvashatjuk, így lehetőség nyílik a dinamikus kód cseréjére.

2.2. A Sparkle tételbizonyító rendszer

Mint a Clean nyelv bemutatásában már utaltam rá, a tisztán funkcionális nyelvek különösen alkalmasak arra, hogy rájuk vonatkozó logikai állításokat matematikai és logikai eszközökkel bizonyítsunk be.

Léteznek általános célú tételbizonyító eszözök, mint pld. az ML alapú Isabelle/HOL [6], vagy a COQ [7]. Ezek a rendszerek készen adják a logikai tételbizonyítás eszközeit, a használatukhoz viszont le kell képezni a programnyelv szemantikáját a rendszer által kezelhető tények és állítások halmazára. A feladat megoldására tett kísérletek bebizonyították, hogy túlnyomó részt megoldható, de rengeteg munkát igényel, és néhány lényeges tulajdonságot vagy nem lehet leképezni, vagy annak leképezése a használhatatlanságig bonyolítaná a modellt [8].

A munkaigényességnek pont a bizonyítórendszerek általános célúsága az oka. Ahhoz, hogy a rendszereket a feladatok minél szélesebb körének megoldására lehessen használni, azokat nagyon magas szintű absztrakcióval kellett kialakítani. Az általános jelleg miatt nem kerülhettek bele a csak egy-egy nyelv illetve terület specialitásai, amik ugyan az eszközeikkel többnyire megfogalmazhatóak, de csak nagyon körülményesen.

A lusta kiértékelést azonban egyik általános rendszer sem kezeli. A Clean programok helyességbizonyítása esetén ez olyan jelentős szemantikai különbségeket szülhet a konkrét és az absztrakt programok között, ami a rendszer használhatóságát erősen lecsökkenti. Az egyik lehetséges eredmény, hogy csak olyan állításokat tudunk a konkrét programokra bizonyítani, amelyek igazsága nem függ a kiértékelés sorrendjétől. A másik pedig az, hogy csak olyan konkrét Clean programokról érvelhetünk, amelyekben minden kifejezésre előírjuk a szigorú kiértékelést. A garantálhatóan helyes programok köre mindkét esetben csökken, például elveszítjük többek között annak a lehetőségét, hogy végtelen listákat használhassunk bizonyítottan helyes módon, helyes eredményt adva.

Hamar felismerték, hogy az általános bizonyítórendszerek használatához szükséges munkamennyiséggel lényegesen jobb képességű helyességbizonyítóhoz juthatunk a logikai rendszer tudása és minél teljesebb körű eszköztára helyett inkább a Clean nyelv (illetve általában véve a lusta kiértékelésű tisztán funkcionális nyelvek) sajátosságait szem előtt tartva. Ennek eredményeképp elkészült a Clean nyelv saját tételbizonyító eszközének első prototípusa, a CPS (Clean Prover System) [9], majd ez alapján egy teljesebb tételbizonyító, a Sparkle [5].

A Cleanben íródott Sparkle segítségével elsőrendű logikai állításokat bizonyíthatunk be Clean kifejezésekre. Számos jó tulajdonsága teszi hatékony és erős funkcionális tételbizonyító eszközzé.

A Sparkle egyik legfontosabb előnye az általános rendszerekkel szemben az, hogy az absztrakt programok megfogalmazásához az „alap-Clean” (core Clean) nyelvet használja. Ez a Clean egy olyan részhalmaza, ami a rendelkezik a Clean nyelv alapvető tulajonságaival, és amivel a teljes nyelv többi elemének is jó része leírható. A helyességbizonyítás szempontjából ennek nyilvánvaló jelentősége az, hogy az általános eszközök használata esetén a konkrét program absztrakt szintre való leképezése során gyakran fellépő problémák itt nem fordulhatnak elő, továbbá nem kell azzal foglalkoznunk, hogy a leképezés hiányosságai miatt esetleg az absztrakt programra bizonyított állítás a konkrét programra már nem teljesül. A gyakorlati eredménye pedig az, hogy a megírt Clean programjainkat átalakítás nélkül megyithatjuk a Sparkle-ben. Ezt a műveletet már a Cleanhez készült grafikus fejlesztői környezet, a Clean IDE is támogatja, így a fejlesztő egy függvény megírásával egyidőben próbálhatja bizonyítani, hogy az tényleg megfelel annak a specifikációnak, ami alapján készítette.

A bizonyítási lépések során a Sparkle a Clean szemantikáját leíró lusta gráfújraírás szerint végzi el a redukciós és kifejezésértelmező lépéseket. Kezeli továbbá a Clean típusrendszerének is nagy részét. Ez az nyelv két olyan alapvető tulajdonsága, amit az általános rendszerekben egyáltalán nem, vagy csak meglehetősen sok munka árán és torzulva lehet ábrázolni.

A Sparkle logikai eszköztára nem ér fel az általános tételbizonyítók eszközeivel, azonban a logika a Clean tulajdonságait figyelembe véve kiegészült egy jól használható tudásbázissal, aminek jelentős kifejezőerő az eredménye. A tudásbázis két formában jelenik meg. Az egyik a levezetési lépés során alkalmazható taktikák halmaza, amelyben az általános, csak a logikából származtatható lépések mellett megtalálhatóak a tisztán funkcionális nyelvekhez, illetve kifejezetten a Cleanhez illeszkedő lépések is. Egy jellemző példa a „Redukció” (Reduce) nevű taktika, amely a megadott kifejezést a Clean nyelv szemantikája, illetve a kifejezés és annak részei definiciója szerint helyettesíti, „fejti vissza” ugyanazokkal a lépésekkel (általában az aktuális paraméterek egyenletes helyettesítése a definíciókba), ahogyan ezt a Clean fordító és a futtatókörnyezet maga is tenné. Mint korábban írtam, a hivatkozási átláthatóság következményeként az így kapott, redukált kifejezés szabadon helyettesítheti az eredetit. Egy másik tipikus taktika, amely kihasználja a nyelv tulajdonságainak ismeretét, a rekurzív adatszerkezetek, illetve definíciók szerinti indukció. Az egyik leggyakrabban használt adatszerkezetet, a listát feldolgozó függvények esetén ezt a taktikát legtöbbször mindenféleképpen alkalmazni kell, ha eredményre akarunk jutni.

A tudásbázis másik lényeges megjelenése az axiómák definíciója, illetve a már bebizonyított tételek nagy száma, amik a Sparkle részét képezik. Ezeket a saját bizonyításainkban tényként használhatjuk, ha az őket témakör szerint csoportosító szakaszokat (sectionök) a saját munkánk mellett megnyitjuk. Az alapvető adattípusok és az azokhoz tartozó műveletek lényeges tulajdonságainak bizonyításával így már általában nem kell foglalkozni, ami rengeteg munkát spórol meg a Sparkle használójának. Ilyen alapvető típusok a logikai típus, az egész számok és a listák. Ezeken felül logikai azonosságokból is kapunk egy jól használható készletet.

Egy bizonyítás során a következők állnak rendelkezésünkre:

- A bizonyítandó állítás
- Hipotézisek
- Taktikák
- Tételek (sajátjaink és a Sparkle „beépített" tételei is)

Egy bizonyítási lépéshez egy alkalmazható taktikát kell választanunk a rögzített listából, és annak megfelelő paramétereket adni. A Sparkle a taktika szerint átalakítja a hipotéziseinket és/vagy a bizonyítandó állítást. A taktikák ugyan rögzítettek, de rendkívül rugalmasan paraméterezhetőek, így több közülük meglehetősen általános. A „Rewrite" (újraírás) taktikával például testszőleges kifejezést újraírhatunk egy hipotézis vagy tétel alapján, ha azok illeszthetőek. A már említett „Reduce" taktikának is számos paramétere van a feldolgozandó kifejezésen kívül. Megadhatjuk többek között, hogy csak egy redukciós lépést hajtson végre a Sparkle, vagy a normálformát szeretnénk megkapni, vagy hogy a kapott állítás a hipotéziseink közé kerüljön, avagy a célállításba implikációs feltételként.

A taktikák alkalmazásában a Sparkle segítséget nyújt a használójának. Alapvető szolgáltatása, hogy kiszűri azokat a taktikákat, amelyeket biztosan nem tudunk illeszteni sem a hipotézisekre, sem az állításra. Az alkalmazható taktikáknál továbbá a paraméterek nagy részéhez megkapjuk a választható értékek listáját. A legnagyobb segítség azonban az, hogy a rendszer a taktikák egy részhalmazát felhasználva minden lépésben értelmes javaslatokat ad, hogy melyik taktikát milyen paraméterekkel alkalmazzuk. Annak a valószínűségét, hogy az adott taktika közelebb visz az állításunk bebizonyításához, 0-100-ig pontozza a Sparkle. Beállítható egy alsó korlát, ami felett egy taktikát automatikusan alkalmaz a rendszer, így egyszerűbb kifejezésekre és állításokra teljesen automatikus tételbizonyításra is lehetőség nyílik. Erre az irodalomban konkrét példát is találhatunk [5].

Az állításainkat és a készülő, illetve elkészült bizonyításainkat természetesen elmenthetjük. A Sparkle-lel kapott tételekhez hasonlóan ezeket szakaszokba rendezhetjük, így az összetartozó állításokat egy fájlban kezelhetjük.

A munkát grafikus felület segíti. Egyszerűbb tételeket az állítás megadása után egérkattintásokkal, billentyűzethasználat nélkül bizonyíthatunk. A pusztán vizuális élményt kínáló lehetőségeken, például a háttérszínek szabad állíthatóságán kívül pedig hasznos grafikus elemeket is nyújt a rendszer. A színekkel kiemelt szöveg mellett ilyen a logikai szimbólumok grafikus megjelenítése is.

A Sparkle által nem kezelt nyelvi elemek között van néhány olyan (pld. makrók), amelyekre a rendszer egyszerűen kiterjeszthető lenne. Jónéhány elemet viszont, mint például a unique és a dinamikus típusok, elméleti problémák miatt nem kezel a Sparkle. Támogatásukhoz a logika kiterjesztésére, illetve maga a Clean rendszer támogatására lenne szükség. Ezekre később látni fogunk néhány példát és magyarázatot.

3. fejezet

Elméleti háttér

Ebben a fejezetben a probléma elméleti hátterét mutatom be. Szerepelni fog egy példa, amely a motivációt mutatja, egy architektúra-modell, amely a példában felvetett probléma általános megoldására törekszik, és aminek egy része a jelen diplomamunka tárgya, valamint a hasonló feladatokra már létező megoldások, illetve elméleti megfontolások áttekintése.

3.1. Dinamikus Clean folyamatok temporális tulajdonságainak ellenőrzése

A szakasz alapját a [10] cikk, és az abban implicit módon felvetett problémák képzik.

A Clean nyelven íródott reaktív programok az ObjectIO csomag használatán keresztül érhetik el a környezetük objektumait. Mint már volt róla szó, ezek az objektumok a nyelvben unique típusúak, aminek az az eredménye, hogy a hozzájuk tartozó változók értéke destruktívan felülírható, ám a hivatkozási átláthatóság megmarad. A destruktív felülírás csak a háttérben történik, a programjainkban ezeket az objektumokat pontosan ugyanúgy használhatjuk, mint a nem unique-okat, persze a hivatkozások számának korlátját figyelembe véve.

A külvilágnak azon részét, amit a programozó nem hozhat létre, csak változtathat rajta, a programnak kívülről kell kapnia. Egy Clean program futtatása a Start függvény kiértékelését jelenti, aminek típusa reaktív programok esetén `*World -> *World`[1]. A `World` típus egy eleme egy olyan összetett objektum, ami tartalmazza a program környezetének,

[1] A visszatérési érték más összetett típus is lehet, ha annak a *World része.

a külvilágnak a program által használható komponenseit. A '*' jelzi, hogy a típus unique.

Mivel értékadás nincs a nyelvben, a Start függvény paramétereként kapott értéket nem tárolhatjuk egy globális változóban, hanem azt minden olyan függvénynek át kell adnunk, amelyik azt használni szeretné. Ahhoz, hogy a unique típusú elemekre vonatkozó korlátozást betarthassuk, az ilyen függvényektől a külvilág általuk megváltoztatott értékét vissza kell kapnunk, és azután már csak arra hivatkozhatunk. Ennek a technikának explicit környezetátadás a neve.

A technika alkalmazásában egy sajátos funkcionális nyelvi elem, a '`let`', illetve annak egy speciális változata, a '`#`' (let-before, vagy hash-let) nyújt segítséget. A „`let` *definíciók* `in` *kifejezés*" szerkezet egy új láthatósági tartományt definiál, használatával azt érhetjük el, hogy a definíciók a kifejezésre nézve lokálisak lesznek. A '`#`' segítségével pedig ehhez hasonló módon egymásba is ágyazhatjuk a láthatósági tartományainkat. A szerkezeteket használva nem kell a hívott függvény által visszaadott unique értéknek egy új nevet adni, és ügyelni arra, hogy a régire már ne hivatkozzunk, hanem ehelyett beágyazhatunk egy új láthatósági tartományt, és az értékünket ugyanolyan nevű változóhoz rendelhetjük, ami az előző változatot elfedi.

A környezetátadás technikája és let-before szerkezetek együttes használata azt eredményezi, hogy a külvilág elemeit használó kódrészek imperatív programozási stílusjegyeket hordoznak. A külvilágot változtató programok tulajdonságait hagyományos elsőrendű logikával csak rendkívül körülményesen lehetne megfogalmazni, az imperatív programok helyességbizonyításához használt legyengébb előfeltételszámításra épülő kiforrott módszerek viszont pont az ilyen esetekben alkalmazhatók jól. Az imperatív stílus megjelenése a funkcionális programokban további motivációt ad arra, hogy ezeket a bevált módszereket próbáljuk itt is alkalmazni.

Absztrakt, párhuzamos, imperatív programok helyességbizonyítására, sőt, szintézisére már korábban készült egy modell [11], amiben párhuzamos programok viselkedését funkcionális módon specifikálhatjuk. A bizonyításokhoz, illetve szintézishez használt kalkulus a temporális logikai alapokon nyugvó UNITY-val analóg, de azzal ellentétben leggyengébb előfeltételszámítást használ. A [10] cikk szerzői ennek a modellnek a segítségével bizonyították be temporális logikai állításokat két reaktív Clean folyamatról, amiből itt az elsőt mutatjuk be (3.1. ábra)[2], amely dinamikus komponenst is tartalmaz. A program fela-

[2]Az itt szereplő program a cikkben szereplő első példa egy módosított változata, amire az azóta megjelent 2.0-s Clean verzió és 1.2.2-es ObjectIO verzió újdonságai és változásai miatt volt szükség. A bemutatott program a nem implementált funkcióktól eltekintve, illetve az első nem implementált szakaszt az „`= (checkecFun, pst)`" kifejezéssel helyettesítve szintaktikailag helyes, de a Clean statikus és dinamikus linkerei hibát jeleznek az összeszerkesztéskor. Mivel a komponensek külön-külön működnek, így a hiba oka

```
module dynamic_bubblesort_Clean2
import StdEnv, StdIO, StdDynamic
Start :: *World -> *World
Start world
  = startIO NDI unsortedlist initialize [] world
where
  unsortedlist = [100,98..0]
  initialize :: (PSt [a]) -> PSt [a] | TC, < a
  initialize pst
    # (fname, pst) = getFileName pst
    # (dynamicSweep, pst) = readSweep fname pst
    = snd (openTimer undef (Timer 0 NilLS
          [TimerFunction (noLS1 (bubble dynamicSweep))]) pst)
  getFileName pst
    # (maybeFileName, pst)     = selectInputFile pst
    | isNothing maybeFileName  = getFileName (appPIO beep pst)
    | otherwise                = (fromJust maybeFileName, pst)
  readSweep :: String (PSt [a]) ->
    ( ([a] -> (Bool,Int,Int)), (PSt [a]) )  | TC, < a
  readSweep fname pst
    # (ok, dynamicFun, pst) = readDynamic fname pst
    | not ok
      = abort "Could not read Dynamic!"
    # (checkedFun, pst) = typeCheck dynamicFun pst
    /******* Nem implementált szakasz (1) ********/
    | semanticCheck specification checkedFun  = (checkedFun,pst)
    | otherwise                  = abort "Semantic check failed!"
    /***** Nem implementált szakasz (1) vége *****/
  where
    typeCheck :: Dynamic (PSt [a]) ->
      ( ([a] -> (Bool,Int,Int)), (PSt [a]) )  | TC, < a
    typeCheck (f :: ([a^] -> (Bool,Int,Int))) pst = (f, pst)
    typeCheck _ pst = abort "Type check failed!"
    /******* Nem implementált szakasz (2) ********/
    specification = Spec (
      \list -> (isSorted, i, j)
    where
      isSorted = (forall k in 0..list.dom-2): list!!k <= list!!(k+1))
        not isSorted -> (i<j and list!!j < list!!i)
    )
    /***** Nem implementált szakasz (2) vége *****/
  bubble :: ([a] -> (Bool,Int,Int)) NrOfIntervals (PSt [a])
            -> PSt [a]  | TC, < a
  bubble sweep _ pst=:{ls=list}
    | sorted = closeProcess pst
    | otherwise = {pst & ls=updateAt i (list!!j)
                      (updateAt j (list!!i) list)}
  where
      (sorted,i,j) = sweep list
```

3.1. ábra. A dynamic_bubblesort program a Clean 2.0 szintaxishoz igazítva data, hogy a folyamat unique állapotában tárolt listát rendezze. A rendező függvény a rendezettség megállapítására dinamikus kódot használ.

A Sparkle használatával ellentétben itt első lépésként meg kell alkotni a konkrét program egy absztrakt modelljét, amin a bizonyítás elvégezhető. Ez egy általános reaktív Clean program esetében meglehetősen nehéz feladat lenne, a konkrét példában azonban jól azonosíthatóak az állapottér komponensei és az állapotátmenetekért felelős programrészek. Az általános absztrakt program, és az alább használt specifikációs eszközök definíciói megtalálhatóak a modell leírásában [11].

A programról azt bizonyítják a cikk szerzői, hogy az elkerülhetetlenül fixpontba jut, és hogy a folyamat állapotában tárolt lista az eredeti lista rendezett permutációja. Formálisan ez a következő specifikációs kritériumokat jelenti:

$$\uparrow \hookrightarrow \mathrm{FP} \qquad (1)$$

$$\mathrm{FP} \Rightarrow (pst.ls \in perms(unsortedlist) \wedge sorted(pst.ls)) \qquad (2)$$

A specifikációs tulajdonságot (2) a szerzők egy invariánssal (3) és egy gyengébb fixponttulajdonsággal (4) helyettesítik, és ezt bizonyítják sikeresen az absztrakt programra.

$$\mathrm{inv}(pst.ls \in perms(unsortedlist)) \qquad (3)$$

$$\mathrm{FP} \Rightarrow sorted(pst.ls) \qquad (4)$$

A bizonyítás során fel kell használni a programban szereplő, a dinamikus kódra vonatkozó specifikációs állításokat, amelynek helyességét a dinamikus kód beolvasása előtt a konkrét programban a semanticCheck függvény hivatott ellenőrizni. A Clean nyelv egyelőre semmilyen formában nem képes logikai specifikációk ellenőrzésére, sőt maguk a specifikációk sem adhatóak meg ilyen formában.

3.2. Hordozható, bizonyított állítások

Az előző szakaszban láttuk, hogy milyen hasznos lenne, ha a dinamikus kódra felhasználásával egyidőben be tudnánk bizonyítani, hogy az megfelel a rá vonatkozó specifikációs kikötéseknek. Erre két nyilvánvaló út létezik, ám az egyik közülük még nagyon sokáig csak elméleti megoldás maradhat.

valószínűleg az, hogy a még csak kísérleti stádiumban lévő Dynamic implementáció nem kezeli helyesen az ObjectIO folyamatokat.

Az utópisztikus megközelítés szerint csak a kódot és a specifikációt alapul véve dönthetnénk el, hogy a kód helyes-e. Olyan algoritmus azonban sajnos még nem létezik, amelyik ezt tetszőleges elsőrendű, sőt, temporális logikai állításra automatikusan meg tudná tenni, ráadásul a lefordított, bináris kódot alapul véve. Ha létezne is ilyen, számításigénye valószínűleg akkor is lehetetlenné tenné, hogy ezt futásidejű[3] ellenőrzésre alkalmazzuk.

A másik, reálisabb megoldás, hogy a dinamikus kódhoz csatoljuk a rá vonatkozó bizonyított állítások egy halmazát. Így ahhoz, hogy a kódról eldönthessük, hogy az megfelel a specifikációnak, a kód- és típusellenőrzésen kívül két dolgot kell tenni:

(1) igazolni kell, hogy az állítások valóban teljesülnek az adott kódra, és

(2) be kell bizonyítani, hogy az állítások megfelelnek a specifikációs kikötéseknek.

A második feladatnál sok függ attól, hogy a megfelelést hogyan értelmezzük. Ha logikai implikációt akarunk bizonyítani, akkor egy nehezen, és főleg lassan megoldható problémát kapunk: egy általános logikai tételbizonyító rendszert kell implementálnunk. Ha viszont megelégszünk egy szűkebb relációval, akár futási időben is könnyen megoldható feladathoz is juthatunk (viszont lesz olyan helyes kód, ami nem lesz felhasználható, és a program előállítójának kell külön gondot fordítani arra, hogy ez ne fordulhasson elő).

Az első feladat első látásra semmiben sem különbözik az eredetitől, hiszen itt is logikai állításokat kell bizonyítanunk ugyanarra a forráskódra. Ebben az esetben már viszont kihasználhatjuk, hogy ezek bizonyított állítások. Ha az állítások és a kód mellé az állítások bizonyítását is csatoljuk, akkor az első feladat a következő kettőre bontható tovább:

(3) be kell látni, hogy az állítások a kapott kódra vonatkoznak, és

(4) igazolni kell, hogy a bizonyítások helyesek.

A bizonyítások előállítása nagyon sokáig tarthat, és ehhez emberi interakcióra is szükség lehet. A bizonyítások helyességének igazolása ezzel szemben egyszerű számítási feladat a legtöbb bizonyítási modellben, amit akár futási időben is végrehajthatnánk. Ha nem akarjuk korlátozni az így hordozható állítások körét, akkor viszont az ellenőrzéshez az eredeti forráskódra is szükség lehet.

Az állítások és a kód összetartozása viszont már keményebb dió. Ahhoz, hogy az eredményben biztosan megbízhassunk, az állításokban szereplő összes programszimbólumhoz

[3]Ebben a szakaszban minden olyan műveletet futási időben végzettként kezelek, ami a felhasználó szempontjából akkor történik. Többek között ide tartozik a dinamikus és a statikus kód összeszerkesztése, ami során a típus -és szemantikai ellenőrzést a rendszer végzi (vagy végezhetné).

csatolni kell annak definícióját is, majd az összes szimbólumra ellenőrizni kell a programkódban, hogy ott annak valóban mindegyik előfordulása az adott definíció szerinti. Ezt a kódhelyesség ellenőrzésével párhuzamosan végezhetjük.

Ennél lényegesen egyszerűbb a dolgunk, ha az összetartozás ellenőrzésére nincs szükség, mert valamiért megbízhatunk a kapott információban. Ennek egyik oka lehet, ha ezt az összetartozást már egy hiteles helyen ellenőrizték. Az is megbízható információnak számíthat, ha az állításokat a kódba ágyazva kapjuk, ahol azok a program szimbólumait, és nem a másolatukat használják. Olyan esetekben, amikor a bizonyítás ellenőrzéséhez a forráskód is szükséges, már több lépcsős a hivatkozás: a gépi kódhoz csatolt forráskód szimbólumai a gépi kód szimbólumaira hivatkozhatnak, az állításokban találhatóak pedig a forráskódéra.

A CPPCC architektúra [12] (Certified Proved-Property-Carrying Code, azaz bizonyított tulajdonságokat hordozó hitelesített kód) a programkód előállítója és felhasználója közé egy harmadik résztvevőt is helyez. Az új, középső szint feladata, hogy a lehető legtöbb ellenőrzést elvégezze a linker és a futtatókörnyezet helyett, majd az eredményeket hitelesítésével lássa el. Az így ellenőrzött kód könyvtárakba kerülhet, ahhonan azok már biztonságosan használhatók. A sikeres működés alapfeltétele, hogy a középső szinten működő programokban mind a kódot előállító, mind az azt felhasználó megbízzon. Ez nem csak azt jelenti, hogy annak rosszindulatú ténykedése kizárható, hanem hogy véletlen hibákat sem vét a vele kapcsolatba lépők rovására.

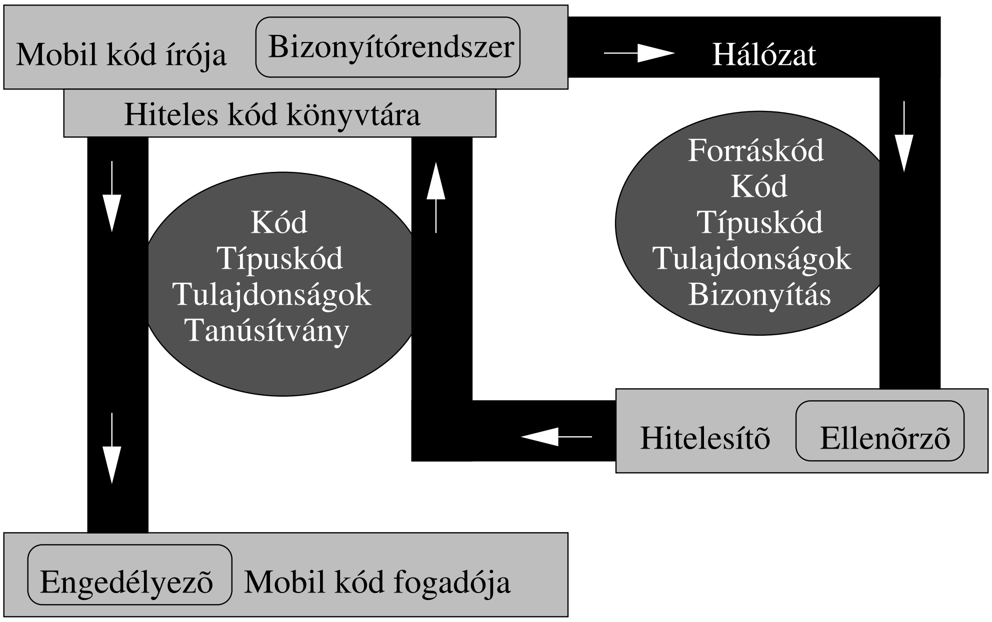
Az alkotók szerint az architektúra elemei a következő feladatokat látnák el (3.2. ábra):

1. A mobil kód előállítója az elkészült program mellé csomagolja az eredeti forráskódot, a típusinformációt, a kód néhány lényeges tulajdonságát, és azok bizonyítását, amit a kód írója egy (fél)automatikus tételbizonyító segítségével állít elő.

2. A hitelesítő a forráskódot felhasználva ellenőrzi az állítások bizonyítását, aztán ellenőrzi, hogy az absztrakt kód tényleg a forráskód helyes fordításával állt elő, végül ellenőrzi, hogy helytálló-e a típusinformáció. Ha mindent rendben talál, a csomagot a forráskód eltávolítása után egy tanúsítvánnyal látja el. Az így hitelesített kódot egy könyvtárba helyezheti.

3. A kód fogadója futási időben ellenőrzi a tanúsítványt, összeveti a tulajdonságokat és a típusinformációt a specifikációval, és engedélyezi, vagy elutasítja a kód alkalmazását.

A jelen diplomamunka konkrét témája ez az architektúra, illetve ennek egy része.



3.2. ábra. A CPPCC architektúra felépítése

3.3. A konkrét feladat

A CPPCC architektúra általános felépítésű, és bármelyik mobil kódot támogató programozási nyelvre alkalmazható. A Clean nyelv rendelkezik az architektúrához szükséges elemek legtöbbjével, így meg lehet kísérelni a nyelvet és az architektúrát egymáshoz illeszteni.

A jelen diplomamunka konkrét feladata a CPPCC architektúra első két komponense, a programkód előállítója és az ellenőrző réteg közötti kapcsolat elemzése, a Clean nyelvhez illesztése, illetve lehetőség szerint egy prototípus implementálása.

Ehhez nézzük először részletesebben az említett két komponenst, és a közöttük lévő kapcsolatot az eredeti formájában!

Az architektúra szerint a mobil kód készítéséhez egy jegyzetekkel ellátott forráskód szolgál alapul. Az egyes komponensekhez írt jegyzetek az adott komponensre jellemző tulajdonságokat tartalmazzák, természetesen csak olyanokat, amik levezethetőek. A típusdeklarációkat a nyelv maga megkövetelheti, vagy a hozzá készített elemző vezetheti le és írhatja a kódba. A fordító akkor tudja sikeresen befejezni a munkáját, ha a tételbizonyító segítségével bizonyítható, hogy minden felsorolt tulajdonság érvényes a forráskódra nézve. Az eredményül adott kimenet tartalmazza az eredeti forráskódott a benne jegyzetként szereplő állításokkal, az állítások bizonyítását, amit a tételbizonyító bizonyítási lépések sorozataként biztosít, a típusinformációt, és a generált absztrakt gépi kódot. Az ezekből készült csomagot aztán a kód előállítója átküldi az ellenőrző, hitelesítő rétegnek.

A kód hitelesítőjének az a feladata, hogy a lehető legtöbb ellenőrzést elvégezze a kapott csomag elemein, ezzel levéve a terhet a futtató rendszer és a dinamikus szerkesztőprogram válláról. A hitelesítéshez ellenőrizni kell az állítások helyességét a forráskódra nézve, a kapott bizonyítás felhasználásával. Ez a feladat egyrészt az egyes bizonyítási lépések, és azok helyes sorrendjének ellenőrzését jelenti, ami teljesen automatikusan, emberi beavatkozás nélkül végezhető, ráadásul a feladat számításigénye sem túl nagy. Másrészt szükség van a forráskód, az állítások, és a bizonyítás összetartozásának ellenőrzésére, ha ez másképpen nem biztosított.

A hitelesítő rendszer másik feladata annak ellenőrzése, hogy az absztrakt kód tényleg az eredeti forráskódból származik. Más olyan ellenőrzésre nem lesz szükség a továbbiakban, amihez a forráskódra szükség lenne, így a műveletek végén az a kapott csomagból törölhető. Ha minden ellenőrzés sikeres volt, a hitelesítő egy tanúsítvánnyal látja el a megmaradt komponensekből készült új csomagot. Az ellenőrző réteg kimenete tehát tartalmazza az absztrakt gépi kódot, a típuskódot, a tulajdonságokat (amiket a forráskód jegyzeteiből előbb ki kell gyűjteni) és a tanúsítványt. Ezt a rendszer visszajuttatja a mobil kód előállítójához, ahol egy könyvtárba kerül.

A két réteg közötti kapcsolat protokollját nem szabja meg az architektúra (bár az automatizáláshoz a hálózat használata valamilyen formában szükséges), mint ahogy a csomagok formátumát sem, hiszen ez érdemi többletet nem adna a rendszerhez, viszont jelentősen csökkentené az architektúra általános jellegét és felhasználhatóságát.

A Clean nyelv, a Clean Dynamics és a Sparkle tulajdonságait figyelembe véve célszerűnek tűnt ezen az architektúrán a konkrét implementáció kedvéért kisebb változtatásokat végezni, amelyeket egy későbbi fejezetben fogok bemutatni. Az architektúra alapelve azonban nagyon jól illeszkedik a nyelvhez és a kiegészítő eszközökhöz.

3.4. Hasonló problémák az irodalomban

Az ötlet, hogy a mobil absztrakt gépi kód mellé adjuk át annak tulajdonságait, és a tulajdonságok bizonyítását, nem teljesen újkeletű. Korábban született egy ennél lényegesen általánosabb és több komponensből álló architektúra, a PCC (Proof-Carrying Code) [14], amelynek a megfelelő komponensek összevonása után a CPPCC architektúra váza is speciális eseteként adódhat (a hitelesítéstől eltekintve).

A PCC architektúra alapja, hogy a kód egyes pontjain elhelyezett állítások segítségével egy előre megadott specifikáció alapján egy megfelelő eszköz „helyességi feltételeket” („verification condition”) generál. Ezek olyan logikai feltételek, amelyeket a kód adott pontján teljesítve garantált, hogy a program megfelel a feltételek generálásához használt specifikációnak. Az állítások teljesülését bizonyítani kell, és a bizonyítást a kódhoz kell csatolni. A kód biztonságos felhasználásához végül elég a bizonyítást ellenőrizni.

A bizonyítások megírhatóak „kézzel”, de egyes esetekben automatikus bizonyításuk is elképzelhető. A vázolt architektúra egyik fontos eredménye, hogy bizonyos típusú nyelv és specifikáció esetén megoldható, hogy a bizonyítást már maga a fordító elkészítse. A specifikáció természetesen nem fogható a jelen diplomamunkában engedélyezett elsőrendű logikai állításokhoz. Elsősorban biztonságos, típushelyes memóriahasználatot, illetve néhány esetben futási időre, illetve tárhasználatra vonatkozó korlátokat lehet így garantálni. Előnye a megközelítésnek, hogy az optimalizált kódra is adhatunk garanciát, csak a bizonyítást kell arra is megfelelően elkészíteni, amit a fordító valószínűleg meg tud tenni.

Az architektúra illusztrációjaként egy leegyszerűsített C kódból garantáltan típushelyes gépi kódot előállító fordítóprogramot mutat be a szerző. Később az architektúra alapján elkészült egy másik fordítóprogram is, amely Java bytekódból úgy állít elő gépi kódot, hogy az bizonyíthatóan megtartja a Java osztályai és a bájtkód tulajdonságai által garantált biztonságos memóriahasználatot [15].

Többen is próbálkoztak már programozási nyelvek valamilyen tételbizonyítóba rendszerbe ágyazásával [16, 17]. Az ilyen kísérletek célja sokszor az, hogy a tételbizonyító program támogatásával az adott nyelven írt programok helyességét bizonyítani tudjuk, de gyakran a cél a programnyelvnek magának a vizsgálata. Ez jelentheti a tulajdonságok feltárását és elemzését, vagy a szemantika helyességének, azaz pontosságának és ellentmondásmentességének vizsgálatát. Átütő sikert ezzel a módszerrel még senkinek nem sikerült elérni. A legtöbb esetben vagy a programnyelvet kell erősen leszűkíteni, vagy pedig a beágyazás marad hiányos.

A CPPCC architektúra egyéb komponensei is felmerültek már megoldandó problémaként. A rétegek közötti kapcsolat egyik kérdése például, hogy milyen formában szállítsuk a forráskódot illetve az elkészített dinamikus kódot, és hogyan hitelesítsük az átküldött adatokat. A népszerű, és mobil kódot szintén támogató Java nyelv ehhez a JAR fájlformátumnak megfelelő archívumfájlokat használ [13]. Egy JAR fájl egy speciálisan összeállított ZIP archívum, amely a forráskód, leellenőrzött bájtkód és a futáshoz szükséges egyéb fájlok összefogása és tömörítése mellett lehetővé teszi, hogy a benne szereplő kódot a készítője digitális aláírással hitelesítse.

4. fejezet

Elemzés és megvalósítás

4.1. A megoldás vázlata

A feladat megoldásának felépítése az általános CPPCC architektúra felépítését követi, az egyes komponensek megvalósításánál azonban már akadnak különbségek. Ezekre a többnyire Clean fordító, a Dynamics és a Sparkle felépítése illetve működése miatt volt szükség.

A két fő komponens, a kód előállítója és az ellenőrző rendszer szerepe az architektúrában leírtak szerinti. A kódot tartalmazó csomagot egy erre készült eszköz állítja elő. Ehhez felhasználja az eredeti forráskódot, a lefordított kód egy részét, a Clean fejlesztőkörnyezete által generált projektleíró fájlt, és a Sparkle segítségével készített, állításokat és azok bizonyításait tartalmazó fájlt.

A hitelesítő réteg első feladata a kapott csomag ellenőrzése. Ezután a kapott forrásfájlokat és lefordított könyvtárakat felhasználva előállítja a továbbítani kívánt objektum dinamikus gépi kódját. Következő lépésként a rendszer ellenőrzi az állítások és bizonyítások helyességét a forrásfájlokat is felhasználva, egy külön erre a feladatra készült eszköz segítségével. Ha minden rendben zajlott, a csomagból kiveszi a forrásfájlokat, és beteszi helyette az elkészült dinamikus kódot, illetve a bizonyítottan helyes állítások listáját, a bizonyításokat immár mellőzve.

Az általános architektúrához hasonlóan ez az implementáció sem foglalkozik a két komponens közötti kommunikációval. A kommunikáció jelen esetben azt jelenti, hogy az egyik rendszer által készített kimenete, ami egyetlen fájlból áll, ell kell juttatni a másik rendszerhez. A csomag sikeres ellenőrzéséhez szükséges minden további fájl, információ és paraméter megtalálható ebben az egy fájlban. A fájlt a fejlesztés és tesztelés során egyszerűen lemezműveletek segítségével továbbítottam egyik rétegtől a másikhoz, a valós életbeli felhasználáshoz azonban ez nem elég. Fájlok hálózati továbbítására azonban már számtalan technika és technológia létezik, amelyek közül az egyik rögzítése csak plusz munkát, egyszerű kódolási vagy konfigurálási feladatot jelentett volna, újdonságot azonban nem hordozna. Az architektúra jelen implementációjának felhasználása esetén így szabadon kiválaszthatjuk a rendelkezésre álló módszerek közül az adott környezethez legjobban illőt.

A hitelesítő réteg az ellenőrzések sikeres befejezése, és a csomag összeállítása után nem generál tanúsítványt. Ez a jelen implementáció egyik hiányossága, amit később feltétlenül orvosolni kell. Ehhez többfajta algoritmus is létezik, így a feladat az, hogy a megfelelő kiválasztása után azt meg kell valósítani.

Az általános architektúra és a jelen implementáció közötti leglényegesebb különbség, hogy a dinamikus kód, amelyet a végfelhasználó is meg fog kapni, nem a kódelőállító rétegben keletkezik, hanem az ellenőrző rétegben. Ennek több oka is van, részletes magyarázat (a további különbségek részletezésével együtt) későbbi szakaszokban található.

A következőkben áttekintést adok az általános architektúra által nyitva hagyott pontokra vonatkozó konkrét választásaimról, megoldásaimról.

4.1.1. A küldött adatok formátuma

A kódelőállító és az ellenőrző réteg futásának eredménye is egy nagyobb mennyiségű adat. A két rendszer által létrehozott kimenő adatcsoportnak sok közös tulajdonsága és eleme van. Ezek közül a legfontosabb az, hogy mindkettő fájlt, illetve fájlokat tartalmaz, továbbá hogy az ezeken felül generált információ meghatározott számú és formátumú adatból áll. A fájlok egymásra hivatkozhanak (a forrásfájlok például tipikusan ilyenek), így ahhoz, hogy azokat felhasználhassuk, nemcsak a tartalmukra lesz szükségünk, hanem a nevükre, valamint az egymáshoz viszonyított helyükre az eredeti könyvtárszerkezeten belül. Azt is fontos észrevenni, hogy az adatok értelmesek ugyan külön-külön is, de felhasználni csak együtt lehet őket.

Ez alapján logikusnak tűnik, hogy az eredmény egyetlen fájlból, valamilyen archívumból álljon, ami az összes szükséges fájlt tartalmazza, az eredetivel megegyező könyvtárszerkezetben. A további információk típusuk vagy felhasználási helyük alapján csoportosíthatók, és a csoportok fájlban tárolhatók, amik rögzített névvel és helyen bekerülhetnek az archívumba.

Hasonló feladat megoldására már született hasonló megoldás. A Java JAR fájlformátuma [13] (amely mellesleg elsősorban szintén mobil kód terjesztésére szolgál) egy archívum, amely bájtkódot és további adatokat, és esetleg forráskódot is tartalmaz. A rögzített könyvtárszerkezetbe rendezett fájlok összecsomagolásához a készítők nem látták értelmét saját archívumformátum kidolgozásához. Az ő választásuk napjaink egyik legelterjedtebb formátumára, a ZIP-re esett.

A ZIP formátum több szempontból is előnyös. Az összecsomagolás mellett tömöríti is az adatokat, ami ekkora adatmennyiségnél hálózaton, vagy akár lemezen való átvitelénél is már számít. Elterjedtsége miatt jól használható, ha platformfüggetlenségre törekszünk, mert a formátumot kezelő eszközök szinte minden operációs rendszerre íródtak már. Az archívum kibontása, sőt, összeállítása is viszonylag gyors folyamat.

Előnyei miatt én is ezt a formátumot választottam. Az archiváláshoz nem írtam külön kódot, hanem azt kezelő külső alkalmazást használtam. Így több időm maradt érdekesebb, több újdonságot hordozó témák feldolgozására, viszont így csak prototípusként használható az alkalmazás. Éles bevetéshez célszerű lenne a formátum kezelését is saját kóddal megoldani.

Az két réteg eredményeként előálló ZIP archívum ugyanazt a könyvtárstruktúrát használja, ugyanazokkal a rögzített információs fájlokkal. Az archívum felépítése könyvtárakra bontva a következő (további részleteket a következő szakaszok adnak):

- *Clean Files*: Ez a könyvtár tartalmazza a forráskódot és az eredeti gépi kódot megfelelően alkönyvtárakba rendezve, a forrásfájlokat és kódkönyvtárakat összefogó projekt fájlt, illetve a generált dinamikus kódot.

- *Information*: Ez a könyvtár tartalmazza az összes, nem fájlként bejövő adatot, csoportosítva és rögzített nevű fájlokba írva. Jelenleg két fájl található benne:

  - *Names*: Ebben a fájlban kulcs-érték párokkal leírható adatok találhatóak, ahol a kulcs rögzített értékeket vehet fel. A fájlban egy sor egy adatot tartalmaz, "*kulcs* = *érték*" formában. Jelenleg két olyan rögzített kulcsot használ a rendszer, amelyek itt megjelenhetnek.
  - *Target*: Ez a fájl adja meg a nevét és a típusát annak a függvénynek vagy struktúrának, amiből a dinamikus kódot fogjuk készíteni. A fájlban három sor szerepel. Az első sorban az objektum neve áll, a másodikban mindig két kettőspont van, a harmadik pedig az objektum típusát tartalmazza. A Clean nyelvben a típusjelölést a „::” szimbólum vezeti be, így a három sor együtt egy érvényes Clean deklarációt tartalmaz.

A két fájl mindig megtalálható az archívumban. A hitelesítő réteg eredménye nem tartalmaz kulcs-érték párokkal leírható adatokat, így az ott kapott archívumban a *Names* fájl üres. Ez később természetesen változhat, ha kiderül, hogy az egyelőre nem implementált futásidejű ellenőrzéshez további adatokra van szükség. A *Target* fájl soha nem lehet üres, és a tartalmát a hitelesítő réteg műveletei nem változtatják.

- *Theorems*: Ebben a könyvtárban két fájlt szerepelhet:
  - *Szakasz fájl*: A tételbizonyító rendszer, a Sparkle által elmentett szakasz, ami állításokat és azok teljes vagy részleges bizonyításait tartalmazza, kiegészítve további adatokkal, ami a fájl értelmezéséhez szükséges. Ennek a kiterjesztése kötelezően „`.sec`”.
  - *Proved*: Ebben az elmentett szakasz azon állításai vannak, amik bizonyítottan helyesek. A fájl egy sora egy állítást tartalmaz „*név* : *állítás*” formában. A *név* a tétel szakaszbeli neve, az *állítás* pedig az a karaktersorozat, amivel a felhasználó a Sparkle-ban az állítást annak létrehozásakor definiálta. Ez azért szükséges, mert a szakasz fájlban szereplő állításleírás sok olyan kiegészítő információt és módosítást tartalmaz, ami azt sokszor olvashatatlanná teszi.

A szakasz fájlban szereplő plusz információ a specifikáció és a bizonyított állítások összevetéséhez a kód fogadójának oldalán nem feltétlenül szükséges, ám ha az implementáció ott is a Sparkle-t veszi alapul, akkor jól jöhet, ellenkező esetben pedig a csomagból eltávolítható.

4.1.2. A komponensekről röviden

Mindkét réteg több parancssoros eszközből áll, amelyek egymást hívva végzik el a feladatukat. Az egymás közötti kommunikációhoz visszatérési értéket állítják, fájlok meglétét ellenőrzik, illetve az üzeneteiket fájlokba írják. A felhasználó által indított programok hibajelzési és egyéb üzenetei a standard outputjukon jelenik meg. Közös az eszközökben, hogy mindegyiket a Clean 2.0 disztribúcióhoz képest rögzített helyre kell tenni, és onnan indítani.

Mivel a Dynamics és a Sparkle is csak Windows alá készültek el, a jelen implementáció is csak Windows alatt fut. A UNIX rendszerekkel ellentétben itt nem határolódnak el élesen a grafikus felülettel rendelkező és nem rendelkező programok, ami a mi esetünkben inkább hátrányként jelentkezik. A standard output helyes kezeléséhez a programoknak konzol módban kellene futniuk, de ekkor minden esetben megnyílik egy konzolablak, még ha nem is ír rá semmit az adott program. A Clean rendszer által a programokhoz szerkesztett futtatórendszer pedig konzol módú programoknál pedig a program befejeződésekor mindig kiírja a Start függvény visszatérési értékét, és azután a felhasználó gombnyomására vár. Ha nem konzol módú programot írunk, akkor viszont a standard kimenet olvasása válik nehézzé, mert az mindenféleképpen egy új konzolablakba íródik. A problémák elkerüléséhez a helyes megoldás valószínűleg az lenne, ha Clean eszközökhöz hasonlóan a programok kimenete mindig fájlokban jelenne meg. A felhasználó szempontjából ez ugyan több kényelmetlenséget okozna, viszont elősegítené a műveletek további automatizálását. A programok ennek megfelelő átalakítása a felépítésük miatt minimális ráfordítással járna.

A kódelőállító réteg a következő komponenseket tartalmazza:

- A *PackFiles* eszköz a főprogram, amelyet a megfelelő paraméterekkel meghívva előállítja a továbbküldendő csomagot.
- Az Info-ZIP[1] csomaghoz tartozó, külső *zip* eszköz hívásával állítja elő a főprogram az előre összerakott könyvtárstruktúrából a kész ZIP archívumot.

A hitelesítő réteg a következő komponensekből áll:

- A *DynamicCertificate* eszköz a főprogram amely a kapott ZIP archívumból az ellenőrzések után egy újat állít elő.
- Az Info-ZIP csomaghoz tartozó, külső *unzip* és *zip* eszközök segítségével kezeli a rendszer a ZIP archívumformátumot.
- A prototípus részét képező *sparkle_check* eszköz feladata, hogy ellenőrizze a kapott Sparkle szakaszt a forráskódra nézve, és kiválogassa a bizonyított állításokat.
- A szintén a prototípushoz tartozó *BatchBuild2* eszköz használatával készíti el a rendszer azt a programot, amit lefuttatva megkapjuk a dinamikus kódot.

4.2. A kódelőállító réteg prototípusa

A kódelőállító réteg prototípusa, a *PackFiles* ezköz a Clean fejlesztői környezetére, a CleanIDE-re épül. Az eszköz felépítésének és működésének ismertetéséhez előbb a CleanIDE és a Clean projektek jellemzőit kell áttekintenünk.

[1]http://www.info-zip.org

4.2.1. A CleanIDE és a projektek

Egy Clean program forrásfájljaitól a végrehajtható programig más, gépi kódot előállító nyelvekhez hasonlóan több lépcsőn keresztül vezet az út. A kód írásának segítésén kívül ezt a folyamatot vezérli a Clean fejlesztői környezete.

A Clean programok forrása modulokra tagolódik. Egy modul egy fájlban helyezkedik el. Modulból háromféle lehet: főmodul, definíciós modul és implementációs modul. Egy Clean programnak pontosan egy főmodulja van, amiben többek között a Start függvény definíciója szerepel. A funkciók megvalósításához a felhasznált függvényeket ebben is definiálhatjuk, vagy további modulokba rendezhetjük. Egy implementációs modulban definiált függvények közül csak azok láthatóak a modult használók számára, amik deklarációi a vele megegyező nevű definíciós modulban (is) szerepelnek. A modulok egymásra az „`import`" kulcsszó segítségével hivatkozhatnak. A definíciós modulokban deklarált függvényeket név szerint importálhatjuk, de lehetőség van egy egész definíciós modul (azaz az összes benne szereplő szimbólum) importálására is. Az importálás művelete a C/C++ nyelv „`#include`" direktívájához hasonlóan tranzitív. A főmodult és az implementációs modulokat tartalmazó fájlok kiterjesztése kötelezően „`.icl`", a definíciós modulok fájljaié pedig „`.dcl`".

Egy tipikus Clean program sikeres fordításához és szerkesztéséhez a forrásfájlokon kívül további fájlokra is szükség lehet. Az operációs rendszertől függő, Clean nyelven nem implementálható műveleteket megírhatjuk más nyelven is (természetesen a Clean nyelv által generált gépi kód előírásaihoz igazodva), és azokból kódkönyvtárat készíthetünk, amit aztán egy megfelelően megírt Clean definíciós modulon keresztül haszálhatunk.

Más programnyelvekhez hasonlóan a Clean esetében is lehetőség van külső tárgykódban („`.obj`" vagy „`.o`" fájl), illetve statikus vagy dinamikus könyvtárakban definiált szimbólumok használatára a gépi kódban úgy is, hogy az adott külső kódhoz nem készítünk Clean felületet, hanem azt csak a szerkesztőprogramnak adjuk át. Ilyen hivatkozásokat általában nem a Clean fordító által generált tárgykód tartalmaz, hanem a Clean felületen keresztül felhasznált külső kód.

A felsorolt fájltípusokon kívül egy Cleanben írt program végeredményének (könyvtár vagy végrehajtható fájl) előállítása során mások is keletkeznek. Többek között ilyenek az „ABC kód"-ot, a Clean gráfmanipulációs assembly nyelvére fordított modulokat tartalmazó állományok. A Clean fordítója első lépésben ezeket állítja elő a forráskódból, majd a következő menetben generál belőlük gépi kódot.

A többlépcsős fordítás és a szerkesztés műveleteinek rengeteg paramétere van, amik vagy a készített kód tulajdonságait határozzák meg, vagy a műveletek sikeres végrehajtásához szükségesek. A folyamat sikeres végigviteléhez többek között meg kell adni, hogy hol találhatóak a fordítást és szerkesztést végző eszközök, és hogy azok hol találják a forrás- és egyéb kódfájlokat. A kód befolyásolható tulajdonságai között pedig például megadhatjuk, hogy könyvtárat, vagy végrehajtható fájlt állítsunk elő, használunk-e dinamikus kódot, vagy hogy mekkora legyen a heap és a verem maximális mérete.

A CleanIDE az egy végrehajtható vagy könyvtárfájl létrehozásához szükséges fájlokat és beállításokat egy *projekt*ben fogja össze. Egy projekt beállításait a fejlesztői környezet egyetlen fájlban tárolja, amit a felhasználó is elmenthet, és a rendszer minden fordításkor automatikusan is elment.

A projektfájl egy egyszerű hierarchikus felépítésű szöveges állomány. Egyetlen nem magától értetődő érdekessége a relatív elérési utak kezelése, ami magában a fejlesztőkörnyezetben is megjelenik. Ha egy elérési út prefixe az alkalmazás (a CleanIDE) könyvtára, vagy a projektkönyvtár (a projektfájl könyvtára), akkor az adott prefixet a fejlesztőkörnyezet az „`{Application}`”, illetve „`{Project}`” jelölésre cseréli. Az így kapott relatív elérési utak a projektek hordozhatóságát segítik elő. Sajnos a megvalósításba kisebb hiba csúszott: a prefix-vizsgálat szempontjából a program Windows alatt is megkülönbözteti a kis- és nagybetűket, így előfordul, hogy a alkalmazás illetve a projektkönyvtár alatt lévő fájlok, könyvtárak abszolút elérési úttal szerepelnek a mentett projektfájlban, illetve az IDE-ben.

A fejlesztői környezetben megadhatunk globális paramétereket is, amik minden projektre vonatkoznak. Ilyenek az új projekt létrehozásához alapértelmezett projektbeállítások, és a „környezetek”. Egy *környezet* egy feladattípusra jellemző beállításokat tartalmaz. Itt adhatjuk meg például azon modulok elérési útját, amit az adott feladattípus programjai mindig használnak, vagy a feladattípus programjainak fordítására és szerkesztésére használandó eszközök helyét. Az egyes projektek a környezetek listájából pontosan egyet használnak, amit név szerint azonosítanak. Az elérési utak kivételével az környezethez tartozó beállítások a projekt paraméterei között csak így, implicit módon szerepelnek. A CleanIDE környezetei szabadon módosíthatóak és a lista bővíthető, de egy adott környezet változtatásai minden azt használó projektben érvényesülni fognak. A projekt mentésekor a projekt fájlba a használt környezeten kívül bekerülnek annak beállításai is. Ha egy mentett projekt fájl megyitásakor az IDE nem találja a használt környezetet, akkor azt az alapértelmezett StdEnv környezettel helyettesíti. A projektfájlban tárolt környezeti beállításokat a CleanIDE nem használja fel akkor sem, ha a megadott környezet nem létezik.

4.2.2. A felhasznált kód

A CleanIDE programszerkezete az Clean ObjectIO könyvtára által sugallt felépítést követi. Első lépésként megnyitja és beolvassa a konfigurációs és a környezetek leírását tartalmazó fájlokat. Ezután a bennük lévő, és egyéb összegyűjtött információk segítségével (pld. melyik könyvtárból indították a programot), valamint alapértelmezés szerinti adatokkal inicializálja a felhasznált programstruktúrákat. Ezek között a grafikus elemek, például a főablak is szerepelnek. Végül egy többablakos (MDI) eseménykezelő folyamatot indít, ami további inicializációs lépések végrehajtása után a felhasználói interakciónak megfelelően végzi a további teendőit.

Az IDE meghívható a „`--batch-build`" paraméterrel, és egy projektfájllal is. Ekkor a program a megadott projektfájl alapján próbálja automatikusan létrehozni projektben definiált bináris állományt. Ehhez szekvenciális lépések sorozata szükséges, ám a grafikus IDE függvényeinek és adatstruktúráinak hasznosítása érdekében a program ebben az esetben is elindít egy folyamatot. Az azonban ebben az esetben nem használ ablakokat (NDI), így nincs mód emberi interakcióra.

Az automatikus, parancssoros felület önálló programmá is fordítható, amihez egy megfelelő CleanIDE projekt készült, az ablakok kezelésére szolgáló grafikus elemek teljes mellőzésével. Ehhez a program készítője az inicializációhoz és az NDI folyamat indításához szükséges kódot egy külön főmodulba gyűjtötte. A projekt fordításával előállítható program a *BatchBuild* névre hallgat. A PackFiles eszköz elkészítéséhez ezt a főmodult és projektet vettem alapul.

A CleanIDE programstruktúrái közül a PackFiles elkészítése szemontjából legfontosabb programstruktúra a projektet leíró `Project`, ami egy rekordként implementált átlátszatlan típus. Többek között következő adatokat nyerhetjük ki belőle, illetve változtathatjuk a hozzá definiált függvények segítségével:

- Statikus információk, egy rekord típusú struktúrában
- Globális (a felhasznált környezetben definiált) könyvtárak
- Főmodul neve
- A projektkönyvtár
- A modulok elérési útjai, és egyéb információk

- A linker beállításai, például a felhasznált külső objektumok és kódkönyvtárak elérési útjai
- A statikus könyvtárak adatai (a kódkönyvtárak, a definíciós modulok és a függőségek teljes elérési útjai)
- A használt CleanIDE környezet neve
- A készítendő végrehajtható fájl elérési útja

A statikus információk azok, amelyeket a fordítás és szerkesztés folyamata nem változtat. Ezek közé a modulok, tárgykód fájlok, statikus és dinamikus kódkönyvtárak és a projektbeállítások között megadott mappák teljes elérési útja mellett a CleanIDE rendszer főkönyvtára és a projekt fájlt tartalmazó könyvtár (azaz az „`{Application}`” és a „`{Project}`” szimbólumok értéke) tartozik.

A `Project` típushoz definiált egyéb műveletek között egy-egy függvény szolgál a projektbeállítások fájlba írására és onnan visszaolvasására, valamint egy új projekt létrehozására. Szintén egy egy függvény meghívásával ellenőrizhetjük egy adott modul szintaxisát, lefordíthatjuk azt, gépi kódot generálhatunk belőle, vagy lefordíthatjuk és összeszerkeszthetjük az egész projektet.

A CleanIDE a projektfájl beolvasása után ugyan felülírja az abban található információk egy részét az aktuális adatokkal (például a felhasznált környezetből származó beállításokat, vagy a projektkönyvtárat), a projektfájl beolvasására szolgáló függvény viszont ezt nem teszi meg. Az egyetlen változtatás, amit a függvény elvégez az a relatív („`{Application}`” illetve „`{Project}`” kezdetű) elérési utak kibontása. Ez nagyban hozzájárul a struktúra és a fájlformátum használhatóságához.

A `Project` típus belső reprezentációja tehát egy jól használható felületet kapott, amivel a legtöbb művelet egyszerűen végrehajtható anélkül, hogy a fordító vagy a linker működését mélyebben tanulmányozni kellene. A típust és a műveleteket tartalmazó modulok felhasználását azonban megnehezíti a dokumentáció teljes hiánya. Így például nem derül ki, hogy a projektstruktúrában legalább három helyen megtalálható tárgykódfájl-listák közül melyik tartalmaz külső, illetve a fordítás során előállított tárgykódfájlokat, melyek szükségesek ténylegesen a végeredmény összeszerkesztéséhez, és hogy van-e a listákban ismétlődés, redundancia. Hasonló problémákkal találkozhatunk a modulok és kódkönyvtárak esetén is.

4.2.3. Paraméterek és korlátozások

A PackFiles eszköz a működéséhez szükséges adatok egy részét a parancssori paramétereken keresztül kapja meg, a másik részét pedig a feldolgozandó projekt és a fejlesztőkörnyezet beállításai közül veszi.

Ahhoz, hogy a CleanIDE beállításait megtalálhassa, a futásához szükséges egy olyan Clean 2.0 fejlesztőrendszer, amely a hivatalos verzióhoz képest legfeljebb konfigurációs beállításokban tér el. A „`PackFiles.exe`” programfájlt a fejlesztőrendszer főkönyvtárába, a CleanIDE program mellé kell tenni, és onnan indítani. A program az üzeneteit a konzolra írja. Az üzenetek elolvasásához a konzolkimenetet egy fájlba kell irányítani akkor is, ha az eszközt egy konzolablakból indítottuk[2].

Az eszköz a következő szintaxis szerint indítható:

`PackFiles` <*projektfájl*> <*szakaszfájl*> <*objektumnév*> <*típus*> <*definíciós modul*>

ahol

- az *objektumnév* annak a függvénynek vagy struktúrának a neve, amit dinamikus kódként tovább szeretnénk adni, és amire az állításainkat megfogalmaztuk. Erre az objektumra sűrűn fogok „*központi objektum*”-ként utalni.
- a *típus* ennek az objektumnak a típusa
- a *definíciós modul* az a „`.dcl`” kiterjesztésű fájl, ami az objektumot exportálja
- a *projektfájl* az a „`.prj`” kiterjesztésű CleanIDE alól elmentett projektfájl, ami az objektum lefordításához és felhasználásához szükséges fájlokat és beállításokat összefogja (bővebb magyarázat lentebb található)
- a *szakaszfájl* pedig az a „`.sec`” kiterjesztésű fájl, ami a Sparkle alól elmentett állításainkat és azok bizonyításait tartalmazza. A szakasz tartalmazhat csak részlegesen, vagy egyáltalán nem bizonyított állításokat is.

A parancssorban átadott minden fájlnév lehet abszolút, vagy a PackFiles programhoz viszonyított relatív elérési út.

A program eredménye egy ZIP archívum, aminek a belső felépítése az áttekintésben leírt struktúrát követi. Az *Information* könyvtárban két információs fájl található, a *Theorems* könyvtárban a Sparkle szakasz fájl, a *Clean Files* könyvtár alatt pedig egy teljes projekt,

[2]egyes Windows verziókon elég, ha a kimenetet egy pipe-ba irányítjuk, például a *more* programéba a következő módon: `PackFiles` <*paraméterek*> `| more`

amiben szerepel a központi objektum is, és amely egy megfelelően beállított Clean 2.0 fejlesztőkörnyezetben bármely másik gépen is lefordítható.

Az átadott objektumra, az azt leíró forráskódra és a Sparkle szakaszra több korlátozás és konvenció is fennáll:

- Ahhoz, hogy a csomagot sikeresen el lehessen készíteni, szükség van egy főmodul írására, ahhoz egy CleanIDE projekt létrehozására, és abból egy végrehajtható fájl sikeres előállítására. A főmodul tartalma közömbös, csak a fontos, hogy a központi objektumunkat használja valamilyen módon. Az is elég, ha az StdEnv könyvtár és a megadott definíciós modul importálása után az adott objektumot egyszerűen (esetleg megfelelő paraméterekkel ellátva) a Start függvényhez rendeljük. A *PackFiles* parancsnak az így készített projektfájlt kell átadni.

- A főmodul és a projekt fordításához használt CleanIDE környezetnek az elérési utaktól eltekintve ugyanazokat a beállításokat kell tartalmaznia, mint az CleanIDE alapértelmezett, *StdEnv* nevű környezetének az eredeti állapotában.

- Az objektum neve nem terhelhető túl a programunkban. A felhasznált modulokban másik függvény vagy struktúra azonos névvel akkor sem szerepelhet, ha annak más a típusa.

- A programban nem használhatunk unique típusokat, makrókat, az ObjectIO könyvtárat és dinamikus kódot. A Clean 2.0 StdEnv könyvtára helyett a PackFiles csomaghoz tartozó, azzal funkcionálisan megegyező könyvtárat kell használni. Ez a könyvtár az eredeti változatnál biztonságosabb, ám valamivel kevésbé hatékony. Külön figyelmet igényel, hogy az eredeti StdEnv elérési útja a felhasznált CleanIDE környezetben se szerepeljen.

- Az előbb felsoroltakon kívül a központi objektumunk típusa nem lehet univerzálisan vagy egzisztenciálisan kvantált típus, absztrakt típus vagy szinoníma típus. Ha az objektum több komponensből áll, ezek a megkötések a komponensekre is fennálnak.

- A programban felhasznált forráskódnak és minden további fájlnak vagy a Clean rendszer részét képező modulok között, a fejlesztőrendszer Libraries könyvtárán belül, vagy a projektkönyvtár alatt kell lennie. Máshol elhelyezkedő könyvtárakat nem adhatunk meg sem a projekt, sem a projekthez használt CleanIDE környezet beállításai között. A Libraries könyvtár alatt található kód nem kerül bele a végső csomagba, így ha a beépített könyvtárak működésén változtatni szeretnénk, akkor a hozzájuk tartozó kódot másoljuk a projektkönyvtárunk alá, és ott módosítsuk. Ha saját külső gépi kódú statikus vagy dinamikus könyvtárat, esetleg tárgykód fájlt is használunk, azokat a projektbeállítások között megadott mappákhoz tartozó *Clean System Files* mappába tegyük.

- A Sparkle megengedi ugyan, hogy több egymásra épülő és hivatkozó szakasszal dolgozzunk, ám a PackFiles program csak egy szakaszfájlt fogad. A felhasználó szempontjából ez különösebb megszorítást nem jelent, mert az állítások szabadon tehetők át egyik szakaszból a másikba, amíg nem hivatkozik rájuk másik állítás. A szakasz hivatkozhat a Sparkle részét képező szakaszok állításaira is, de azok nem kerülnek bele a csomagba, így új állításokkal azok a szakaszok nem bővíthetőek.

Az eszköz a korlátozások közül nem ellenőrzi mindegyiket. Az ellenőrzések közül több könnyen pótolható, viszont néhány esetben a megvalósítás jelentős munkával járna. Ha nem ellenőrzött korlátozást szegünk meg, akkor sem generálhatunk biztonságot veszélyeztető, hibás hitelesített csomagot, de a problémák oka csak a hitelesítő réteg hibaüzeneteiből derülhet ki.

4.2.4. A korlátozások magyarázata

Az előző szakaszban felsorolt korlátozások legnagyobb része annak a következménye, hogy a központi objektumunkból dinamikus kódot szeretnénk készíteni, és a Sparkle-ben állításokat szeretnénk rá megfogalmazni. Mindkét művelethez korlátozott nyelvet használhatunk csak. Ezek egy része hiányosság, más része viszont elméleti akadályokba ütközne.

A Sparkle motorja nem kezeli a dinamikus kódot, a makrókat, a unique típusokat, és így az ObjectIO-t sem. Ahhoz, hogy a kódra a Sparkle segítségével állításokat fogalmazhassunk meg, a forrásfájlokat meg kell nyitni az eszközben. Ezt megtehetjük modulonként egyesével, vagy megyithatunk egy projektet is. A modulonkénti megnyitás már egy kisebb program esetében sem járható út, mert az StdEnv és egyéb felhasznált beépített könyvtárak nagyon sok modult tartalmaznak, amiket a függőségek betartásával egyesével megnyitni hosszú és körülményes folyamat lenne. Egy elmentett projektet viszont csak akkor tud sikeresen megnyitni a Sparkle, ha abban minden függőség szerepel és helyesen van megadva, és a függőségek között a központi objektumunkat tartalmazó modul is szerepel. A Sparkle kikötése továbbá, hogy a projekthez tartozzon egy főmodul, aminek a neve megegyezik a projekt nevével. Ezeket a feltételeket csak úgy garantálhatjuk, ha sikeresen létrehozunk egy olyan végrehajtható fájlt, ami az objektumunkat használja.

A dinamikus kód kísérleti implementációja még több korlátozással rendelkezik: túlterhelt típusokat, unique típusokat, univerzálisan vagy egzisztenciálisan kvantált típusokat, absztrakt típusokat és szinoníma típusokat nem lehet dinamikus kóddá, illetve onnan visszaalakítani. Egy Dynamic típusú objektum kódja hivatkozhat ugyan másik Dynamic típusú objektumra (lusta dinamika), de itt is sok megkötés van, és a megvalósításnak több problémája is van.

További megkötést ad, hogy (tapasztalataim alapján) a dinamikus kódot a programhoz szerkesztő DynamicLinker nem működik együtt az ObjectIO könyvtár folyamataival, így ilyen folyamatokat indító programban egyelőre egyáltalán nem használhatunk dinamikus kódot.

A központi objektumunk nevének túlterhelése a Dynamics korlátozásán kívül azért sem engedélyezett, mert a hitelesítő réteg nem szűri ki az állítások közül csak azokat, amelyek a megadott nevű és típusú objektumra hivatkoznak. A bizonyított állítások szövegében később már csak a függvény vagy struktúra neve szerepel, a típusa nem, így elméletben elképzelhető, hogy a felhasználó olyan állítások alapján használja fel a dinamikus kóddal leírt objektumot, amik egy másik, azonos nevű, de más implementációjú objektumra teljesülnek csak. A korlátozás feloldásához mélyebben meg kellene vizsgálni a problémalehetőséget, és csak a veszélyes eseteket kizárni. A másik lehetőség, hogy a bizonyított állítások felsorolását bővítsük a bennük használt szimbólumok típusával. Ez viszonylag egyszerűen megtehető, mert ez az információ a szakaszban megtalálható, onnan csak át kell emelni.

Az CleanIDE környezetre vonatkozó megkötés magyarázata az, hogy a fejlesztő által használt környezet a hitelesítő gépen nem biztos, hogy létezik, ami a projektfájl értelmezése során gondot okozna. Ennek jelenlegi megoldása, hogy a használt környezetet egy olyannal helyettesítjük a csomagban, ami a hitelesítő gépen biztosan létezik. A felhasznált könyvtárakkal kapcsolatos konfliktusok elkerülése érdekében ez a környezet nem tartalmaz elérési utakat. Minden egyéb beállítása megegyezik az *StdEnv* környezet eredeti beállításaival. Szebb megoldás lenne, ha a felhasznált környezetet is hozzávennénk a hitelesítő rétegnek átküldendő csomaghoz. Ennek az az akadálya, hogy a beállítások között szerepel a felhasznált Clean eszközök (a fordító és a linker) elérési útja is. Az eszközökből a fejlesztő saját verziókat is használhat, de ezeket nyilvánvalóan nem lehet átküldeni a hitelesítő rétegnek (akkor például bármilyen vírust is küldhetnénk helyette).

A használható szakaszok számának korlátozása csak átmeneti. Ha több szakaszt engedélyeznénk, akkor beolvasásukhoz azokat előbb a hivatkozási függőségek szerint sorba kellene rendezni. Ez a Sparkle kódjára támaszkodva nem egy különösebben nehéz feladat, így később a felhasználó kényelme érdekében célszerű megvalósítani.

A forráskód fájljainak fájlrendszerbeli helye több okból is kötött. Egyrészt, a forráskód a felhasznált gépi kódú dinamikus és statikus könyvtárakkal együtt még tömörített csomagként is jelentős helyet foglalhat, így a Clean rendszer saját moduljaival nem célszerű tovább terhelni a sávszélességet. Másrészt, mint a projektstruktúra bemutatásánál leírtam, a sikeres fordításhoz szükséges fájlokat dokumentáció hiányában nem tudtam azonosítani, így a hitelesítő réteg csomagjainak összeállításához csak a projektben, illetve a használt CleanIDE környezetben megadott könyvtárakat veszem alapul. Konkrétan az itt felsorolt könyvtárakból a csomagba kerül az összes „`.dcl`” és „`.icl`” kiterjesztésű fájl, valamint a könyvtárhoz tartozó *Clean System Files* alkönyvtár teljes tartalma.

A hitelesítő rétegen való sikeres fordításhoz és ellenőrzéshez a projektfájl hivatkozásaiból következő könyvtárszerkezetet hordozható módon kell tudni megtartani. Ez úgy biztosítható a legegyszerűbben, ha kihasználjuk a CleanIDE relatív elérési út kezelésére használt technikáját. A leírt korlátozásokat betartva a készítendő csomagban az egyes fájlok a projektkönyvtárhoz viszonyított alkönyvtárukba kerülhetnek, ami a feltételeknek eleget tesz.

4.2.5. A megvalósítás részletei

A *PackFiles* eszköz kódja a CleanIDE parancssoros változata, a *BatchBuild* teljes újraírásával készült. Az eredeti kódból felhasználtam a CleanIDE belső adatszerkezeteit leíró típusokat és azok programfelületét, a *BatchBuild* főmodulját azonban csak az ezek kezelésére adott példaként tudtam hasznosítani.

Míg az interaktív CleanIDE módosított változataként készült *BatchBuild* egy ObjectIO folyamatot indít, amelyben a CleanIDE-hez hasonlóan tudja használni a fordítást és összeszerkesztést verzérlő műveleteket, a *PackFiles* lépések rögzített sorozatával oldja meg a feladatot.

Az első lépés az aktuális könyvtár elérési útjának megjegyzése, amit a későbbiekben a Clean rendszer főkönyvtáraként haszál a program. Ezután következik parancssor értelmezése, és rutinellenőrzések elvégzése. A megadott fájlok elérési útját abszolút elérési útra változtatja a program, ami a projektfájl feldolgozása szempontjából fontos lépés.

A projektfájlt az átlátszatlan `Project` típus megfelelő műveletével megnyitva a projekt belső reprezentációjához jutunk, amiből ki lehet válogatni a feladat szempontjából fontos adatokat. A művelet az „`{Application}`” illetve „`{Project}`” kezdetű relatív elérési utakat az aktuális főkönyvtár és projektkönyvtár alapján automatikusankibontja. A projektstruktúrában több helyen is található könyvtárak mindegyikét ellenőrizzük, hogy azok a Clean rendszer *Libraries* könyvtára vagy a projektkönyvtár alatt helyezkednek-e el. A vizsgálathoz az eredeti rendszer elérési utak kezelésére írt eszközeit használom, amelyeket azonban ki kellett javítani, hogy ne különböztessék meg a kis- és nagybetűket. Ha csak egy könyvtár is máshol van, a program hibaüzenettel kilép.

A további munkához egy átmeneti könyvtárra van szükség. Ezt, kihasználva, hogy egy eredeti felépítésű Clean rendszerben dolgozunk, a Clean fejlesztőkörnyezet *Temp* könyvtárán belül hozom létre véletlenszerű névvel. Ebben a könyvtárban elkészítem a leendő archívum vázát: az *Clean Files*, az *Information* és a *Theorems* könyvtárakat.

A projektleírásból összegyűjtött könyvtárak közül eldobom azokat, amelyek a Clean *Libraries* könyvtára alatt vannak, mert az ezekben lévő fájlokat nem kapja meg a hitelesítő réteg. A megmaradt könyvtárak mind a projektkönyvtáron belül vannak. Ezek megfelelőit létrehozom az átmeneti *Clean Files* könyvtáron belül, majd a bennük lévő Clean forrásfájlokat és az egész *Clean System Files* alkönyvtárukat átmásolom oda.

A forráskódból már csak a projekt fájl van hátra. Ezt a projektstruktúra alapján a `Project` típus megfelelő műveletével írom a *Clean Files* könyvtárba. Ez a függvény a beolvasó művelettel ellentétben nem foglalkozik a relatív elérési utak kezelésével, így a fájl mentése előtt a projektstruktúrában mindegyik elérési útba be kell helyettesíteni a megfelelő „`{Application}`” illetve „`{Project}`” előtagot. A felhasznált CleanIDE környezetet is le kell cserélni a fentebb leírt okok miatt. Az új környezet neve *NoPaths*, ami a nevéhez hűen nem tartalmaz elérési utakat. A fejlesztő által használt környezet elérési útjait ezért előbb át kell másolni a projektbeállítások közé. A Sparkle kedvéért az új projektfájl a főmodul nevét kapja.

A leendő archívumból már csak a szakaszfájl és az információs fájlok hiányoznak. Az előbbit a parancssorban megadott helyről másolom a Theorems könyvtárba, az utóbbiakat pedig, szintén a parancssorban megadott adatok alapján, a fejezet általános szakaszában leírt formátum szerint állítom össze. A *Names* fájlba két adat kerül: a központi objektumot exportáló definíciós modul, valamint az elmentett projektfájl neve és helye a projektkönyvtárhoz (illetve az archívumban a *Clean Files*) könyvtárhoz viszonyítva. A projektfájlban található elérési utakhoz hasonlóan a projektkönyvtár helyett itt is „`{Project}`” szerepel az utakban.

Az utolsó lépcső az átmeneti könyvtárba összegyűjtött fájlok archiválása a külső *zip* eszköz segítségével, majd az átmeneti könyvtár és tartalmának törlése.

A tömörítéshez használt *zip* program az Info-ZIP programcsomag része. Ahhoz, hogy a *PackFiles* használni tudja a programot, azt a Clean rendszer *Tools* mappája alatt létrehozott *PackFiles* mappába kell tenni.

A program hiba nélküli futása esetén az elkészült ZIP archívumot a Clean rendszer *Temp* mappájában találjuk meg véletlenszerű névvel és „`.zip`” kiterjesztéssel. A fájl pontos nevét és helyét a program a futás befejezése előtt kiírja a konzolra.

4.3. A hitelesítő réteg tervezése

A hitelesítő réteg feladatait ellátó *DynamicCertificate* eszköz a *PackFiles* programhoz hasonlóan a CleanIDE kódjára és struktúráira épül, és a feladatai egy részét más programok hívásával végzi el.

4.3.1. A megvalósítandó feladatok

A program egyetlen paramétere a kódelőállító réteg, azaz a *PackFiles* eszköz által készített ZIP fájl, ami az összes szükséges nem környezetfüggő adatot tartalmazza. A működéshez szükséges adatokat ebből a fájlból, illetve a Clean fejlesztőkörnyezet beállításaiból veszi a program.

A program fő feladata, hogy a kapott csomagban található adatok, forráskód és Sparkle szakasz alapján előállítson egy másik csomagot, amiben egy megadott objektum dinamikus kódja mellett a Sparkle szakasz állításai közül azok szerepelnek, amik a megadott kódra nézve helyesek. A készült csomagot egy tanúsítvánnyal kell ellátni, amit a jelenlegi prototípus nem tesz meg, ezt később kell implementálni.

Előre le kell szögezni, hogy a feladatra jelenleg csak félmegoldás adható. Ennek a Clean Dynamics kiforratlansága az oka: az elkészült dinamikus kód nem hordozható, csak azon a gépen használható fel, ahol előállították. A Dynamic típusú objektumokat fájlba író könyvtári művelet eredménye önállóan nem értelmes. A háttérben készül egy igen nagy méretű kódkönyvtár, és a generált fájl csupán ennek részeire hivatkozik. A fájl rögzítetten tartalmazza a kódkönyvtár abszolút elérési útvonalát, így ugyanazon a gépen szabadon mozgatható, de másik gépen nem használható. Készül már azonban a Clean SendDynamic, ami éppen a hordozhatóságra ad megoldást. Ha ennek a pontos részletei ismertek lesznek, minden bizonnyal át kell tervezni valamilyen mértékben az itt leírt prototípust. Valószínű azonban, hogy az új módszer eredménye is egy vagy több fájl lesz, és ebben az esetben

```
module DinamikusFuggveny

import
  StdEnv,
  StdDynamic

Start world
  #! (_, world)
    = writeDynamic DynamicFilename DynamicDefaultOptions
                   dynamicObject world
  = world
where
  dynamicObject = dynamic digitToInt
  DynamicFilename = "dinamikus_digitToInt.dyn"
```

4.1. ábra. Egy egyszerű függvény dinamikus kóddá alakítása

nem kell a végeredményül adott archívum felépítésén változtatni. A prototípus által a dinamikus kód előállítására használt módszer pedig valószínűleg kisebb módosításokkal alkalmas lesz az újabb dinamikus kód kezelésére.

Az állítások ellenőrzését és a helyes állítások kiszűrését egy külön programmal oldottam meg, aminek a leírása később következik. A dinamikus kód előállításához először vegyünk szemügyre egy nagyon egyszerű programot (4.1. ábra), aminek az a feladata, hogy egy Clean függvényt, az StdChar modulban definiált `digitToInt`-et dinamikus típusú objektummá alakítson, majd egy fájlba kiírjon. A program lényegi művelete a `!Char -> Int` típusú `digitToInt` függvény `Dynamic` típusú objektummá alakítása a `dynamic` kulcsszó használatával, illetve a kapott objektum fájlba írása a `writeDynamic` függvény meghívásával. Az egyes függvények típusát nem kellett megadnunk, azokat a Clean típuslevezető rendszere megtalálta.

A program a kódelőállító rétegtől kapott csomagban található információ segítségével könnyen átalakítható úgy, hogy a központi objektumunkat alakítsa dinamikus kóddá, majd mentse azt fájlba. Ehhez mindössze a `dynamic` kulcsszó után kell a `digitToInt` függvényt az objektumunk nevére cserélni, valamint az `import` kulcsszó utáni modullistába fel kell venni az azt exportáló definíciós modult is. A típuslevezető rendszer ekkor biztosan meg fogja találni az objektumunk típusát, hiszen az objektum nevének túlterhelését megtiltottuk.

A szükséges változtatásokat nem lehet a lefordított programnak futás közben paraméterként átadni, hiszen importált modulokkal ezt nem tehetjük meg. A módosításokat még fordítás előtt elvégezve, és az új kódot lefordítva viszont egy olyan programhoz jutunk, aminek az egyetlen feladata, hogy a központi objektumunkból egy dinamikus kódot tartalmazó fájlt állítson elő.

Ezt az alapötletet alkalmazva már megoldható a dinamikus kód generálásának feladata is.

4.3.2. A program használata

A *PackFiles* programhoz hasonlóan a *DynamicCertificate* is felhasználja a Clean fejlesztőkörnyezet beállításait és egyes elemeit. A helyes működéshez így a hitelesítő réteg feladatait ellátó gépen is szükség van egy olyan Clean 2.0 rendszerre, amely a hivatalos verzióhoz képest legfeljebb konfigurációs beállításokban tér el. A „`DynamicCertificate.exe`” programfájlt ennek a főkönyvtárába, a CleanIDE program mellé kell tenni, és onnan indítani.

A program egyetlen paramétert kap, a *PackFiles* eszköz által készített ZIP archívum elérési útját. Minden további információt a csomagból, illetve a Clean fejlesztőkörnyezet beállításaiból vesz.

A program tehát a következő szintaxis szerint hívható:

`DynamicCertificate` <*ZIP fájl*>

ahol a *ZIP fájl*t abszolút, vagy a „`DynamicCertificate.exe`” programfájlhoz viszonyított relatív elérési úttal kell megadni.

A program futásának eredménye egy ZIP archívum az áttekintő részben leírt belső struktúrával. Az *Information* könyvtárban két információs fájl található (amiből az egyik üres), a *Clean Files* könyvtárban a központi objektumunk dinamikus kódja van egy fájlba írva, a *Theorems* könyvtárban pedig az eredeti Sparkle szakasz mellett egy külön fájlban megtalálható a helyesnek bizonyult állítások listája. A dinamikus kód a fent leírt hiányosságok miatt csak azon a gépen használható, amelyiken azt előállították.

A program működése során külső eszközöket használ, és a fejlesztőkörnyezet speciális beállítását is igényli. A külső eszközök a Clean fejlesztőkörnyezet főkönyvtárában, illetve a Tools mappájában, azon belül is a PackFiles mappában foglalnak helyet. Ez utóbbiba kell tenni az Info-ZIP csomag két eszközét: a ZIP formátum kezeléséhez használt *zip* és *unzip* programokat, a *sparkle_check* eszközt (amelynek leírása egy későbbi szakaszban következik), és a *Sections* almappába a Sparkle részét képező, „beépített” szakaszokat tartalmazó fájlokat. Ezek a Sparkle saját *Sections* almappájában található következő fájlok: „`booleans.sec`”, „`ints.sec`”, „`lists.sec`” és „`tautology.sec`”. Szükség van még ezeken felül a prototípus részét képező *BatchBuild2* eszközre, amit a Clean rendszer főkönyvtárába, a „`DynamicCertificate.exe`” programfájl mellé kell tenni.

A programnak szüksége van továbbá egy *NoPaths* nevű CleanIDE környezetre. Ezt a következőképpen készíthetjük el. Indítsuk el a CleanIDE fejlesztőkörnyezetet, és a környezetek beállításainál („Environments") válasszuk a „Lista szerkesztése" („Edit List") menüpontot. Itt az „StdEnv" környezet alapján készítsünk egy új, „NoPaths" nevű környezetet a „Másolás" („Copy") művelet segítségével. A kapott környezet beállításai közül távolítsuk el a könyvtárakat. Ezt az „Edit" → „Paths" → „Delete" funkciók használatával tehetjük meg.

A program futása során dinamikus kódot használó programot is indít. Ehhez a programot futtató gépet fel kell készíteni dinamikus Clean kód kezelésére a kapott dokumentáció alapján. Ha a Clean fejlesztőkörnyezet tartalmazza a Dynamics támogatást, akkor a telepítéshez mindössze három dinamikus könyvtárat kell a Windows rendszermappájába másolni a fejlesztőkörnyezet „*Tools\Dynamics 0.0*" mappája alól. Ezek a „`DynamicLink.dll`", „`ServerChannel.dll`" és a „`ClientChannel.dll`" , ez utóbbi a *utilities* almappából.

A program a futása során esetlegesen előforduló hibák leírását a konzolra írja. A kódelőállító és a hitelesítő réteg közötti kommunikáció implementáláshoz ezen minden bizonnyal változtatni kell majd, mert ezeket az üzeneteket a kód írójához vissza kell juttatni. A *sparkle_check* programban alkalmazott módszer, a hibaüzenetek külön fájlba írása a feladatnak jobban megfelelne.

4.3.3. A tervezés és megvalósítás részletei

A *DynamicCertificate* eszköz a *PackFiles*hoz hasonlóan szintén a CleanIDE parancssoros változata, a *BatchBuild* főmoduljának teljes újraírásával készült, és a *BatchBuild*del ellentétben szintén ObjectIO folyamat indítása nélkül oldja meg a feladatát. A CleanIDE forrásából felhasznált modulok ugyanazt a javítást tartalmazzák, mint a *PackFiles* esetében (kis- és nagybetűk helyes kezelése Windows alatti elérési utak összehasonlításakor).

A program első lépésben lekérdezi az aktuális könyvtár nevét. Ezt a könyvtárat a továbbiakban a Clean fejlesztőrendszer főkönyvtáraként használja.

A munka folytatásához szükség van egy átmeneti könyvtárra. Ezt a Clean fejlesztőkörnyezet *Temp* könyvtárában hozza létre a program, véletlenszerű névvel. A kapott ZIP archívumot ebbe a munkakönyvtárba az *unzip* program segítségével kibontva elérhetővé válnak a csomagban található fájlok. Az elkészítendő csomag a megegyező felépítés mellett sok közös elemet tartalmaz a kapott csomaggal, így az új archívum összeállításához is a most kibontott struktúra fog alalpul szolgálni.

A kapott fájlokból először ki kell olvasni az ott tárolt, paraméterként szolgáló adatokat. Ehhez az *Information* könyvtár két fájlját, a *Names*t és a *Target*et kell beolvasni és a tartalmukat értelmezni.

A következő lépcső az egyik érdemi feladat elvégzése, a dinamikus kód elkészítése. Ehhez a fent vázolt ötlet alapján mintákat használ a prototípus, amit megfelelően paraméterezve egy lefordítható programhoz jutunk. Az új, „`.icl`” kiterjesztésű főmodul szövegét egy mintából a központi objektum nevének, az azt exportáló definíciós modul nevének és a készítendő dinamikus kódot tartalmazó fájl elérési útjának megadásával készíthetjük el. A modul neve véletlenszerűen választott lesz. Ahhoz, hogy a modulból futtatható kódot készíthessünk, egy projektfájlra is szükségünk lesz. Ezt a főmodulhoz hasonló módon, egy minta alapján hozzuk létre. Ehhez a főmodul elérési útját, a Clean rendszer főkönyvtárát, és a projektkönyvtárat (jelen esetben a *Clean Files* könyvtár) kell megadni.

Az elkészített projektet a Clean fejlesztői környezet részét képező *BatchBuild* program módosított változatával, a *BatchBuild2* eszközzel fordíthatjuk le és szerkeszthetjük össze. Ha a paraméterként kapott ZIP archívum helyes kódot tartalmazott, a frissen generált projekt sikeresen fordítható. A fordítás eredménye egy olyan program, ami elindítása után a központi objektumunk dinamikus kódját egy fájlba írja.

A készített program dinamikus kódot használ, így a Clean Dynamics jelenlegi implementációja alapján nem egy EXE fájl keletkezik, hanem több könyvtár és egy DOS batch fájl. A program futtatásához a batch fájlt kell lefuttatni. Ez sajnos Clean programból nem lehetséges (legalább is nem sikerült megoldani). Szerencsére a batch fájl egyetlen utasítást tartalmaz, ami a Clean fejlesztőkörnyezet dinamikus linkerének, a „`DynamicLinker11.exe`” programfájlnak a megfelelő paraméterekkel való indítása. Az EXE fájl már meghívható Clean programból is, így az elkészült programot a batch fájlból kiolvasott parancs segítségével futtatjuk le. A Clean Dynamics implementációja minden bizonnyal változni fog a jövőben. Ha a dinamikus kódot használó programokat egy későbbi verzióban másképpen kell majd indítani, elképzelhető, hogy a prototípusban ehhez használt módszer helyett egy újat kell kitalálni és megvalósítani.

Az eddig leírt lépések sikeres végrehajtása után a Clean fejlesztőrendszer *Temp* könyvtárában megtalálható a központi objektumunk dinamikus kódját tartalmazó fájl, így hozzáláthatunk a következő érdemi feladat, az állítások ellenőrzésének elvégzéséhez. A Sparkle csak olyan szakaszfájlokat tud megnyitni, amelyek a saját *Sections* mappáján belül találhatóak. Ez a megszorítás érvényes a Sparkle kódját felhasználó *sparkle_check* eszközre is. Az ellenőrzés első lépése tehát, hogy a *sparkle_check* programhoz tartozó *Sections* könyvtárban létrehozzunk egy átmeneti alkönyvtárat véletlenszerű névvel, és a ZIP archívumban érkezett szakaszfájlt átmásoljuk oda.

A *sparkle_check* program indításakor a szakaszfájlon kívül meg kell még adni a ZIP archívumban kapott projektfájl helyét, továbbá hogy az eredményt és a hibaüzeneteket tartalmazó fájlokat melyik könyvtárban szeretnénk látni. Ha a program nem ad hibaüzenetet, akkor az azokat tartalmazó fájl sem létezik. A végeredményt (helyes állítások) tartalmazó fájl viszont akkor is létrejön, ha az adott szakaszban a forráskódra nézve egyetlen helyes állítás sincs, persze ekkor a fájl üres. Ezek alapján könnyen ellenőrizhető, hogy sikerült-e egyáltalán elindítani a *sparkle_check* programot, és hogy annak futása során volt-e valami probléma. Ha program hibák nélkül lefut, az azt jelenti, hogy a kapott forráskód, valamint a rájuk hivatkozó állítások szintaxisa helyes volt. Ha hibaüzenetet adott az eszköz, akkor azokat a konzolra írjuk, és kilépünk. Ellenkező esetben az eszköz eredményfájlját már csak a megfelelő helyre, a Theorems könyvtárba kell másolni, mert a fájl formátuma az archívumstruktúra bemutatásánál leírt előírásoknak megfelel.

A további műveletekhez sem a ZIP archívumban érkezett, sem az általunk generált forráskódra nem lesz már szükség, így az archívumstruktúra újrahasznosításához azokat törölnünk kell. Ezt legegyszerűbben a teljes *Clean Files* könyvtár rekurzív törlésével, majd egy új, üres könyvtár létrehozásával tehetjük meg. A jelenlegi implementáció egy hibája miatt (egyes Windows verziókon a megfelelő Clean könyvtári függvény nem tud mappákat törölni) előfordulhat, hogy a *Clean Files* mappából csak a fájlokat tudjuk eltávolítani, a könyvtárakat nem, így az eredeti forráskód struktúrájának megfelelő üres könyvtárakból álló rendszert az eredményként előálló új ZIP archívum *Clean Files* mappája is tartalmazni fogja.

Az archívum struktúrájának véglegesítéséhez a generált dinamikus kódot tartalmazó fájlt be kell még másolni az új, elvileg üres *Clean Files* mappába, valamint törölni kell az *Information* mappabeli *Names* fájl tartalmát. Ez utóbbit a fájl törlésével és újbóli létrehozásával tehetjük meg.

Utolsó lépésként a *zip* program használatával a munkakönyvtárban összeállított struktúrából előállítjuk a ZIP archívumot, majd töröljük a feleslegessé vált átmeneti fájlokat (a munkakönyvtárat, illetve a *Sections* mappában létrehozott átmeneti könyvtárat).

Az eredményül kapott ZIP archívum helyét és nevét a program a konzolra írja.

4.4. Az állítások ellenőrzése

A hitelesítő réteg egyik feladata, hogy a kódelőállító rétegtől kapott állításokat és bizonyításait a szintén onnan kapott forráskódra nézve ellenőrizze. Mivel a bizonyítás a Sparkle tételbizonyító segítségével készült, a bizonyítási lépések csak abban érvényesek, a Sparkle tudását kell használnunk a bizonyítás ellenőrzéséhez is.

A Sparkle, grafikus eszköz lévén, nem alkalmas a feladat automatikus megoldására, így a tudása csak a forráskódjának felhasználásával hasznosítható. A feladat megoldásához először át kell tekintenünk a Sparkle, és a állításokat és bizonyításokat tartalmazó szakaszok felépítését, majd az így szerzett információk elemzése után láthatunk a konkrét lépések megfogalmazásához, amelyek a feladatot megoldó *sparkle_check* eszköz elkészítéséhez szükségesek.

4.4.1. A Sparkle felépítése

A Sparkle egy többablakos program, ahol az ablakok száma és funkciója rögzített. Minden egyes ablaknak saját neve is van.

- A *Project Center* szolgál a Clean forrásfájlok kezelésére. Egy modullistát tartalmaz, amit egyesével is bővíthetünk, és egy projekt összes modulját is felvehetjük a projektfájl megnyitásával. Egy modul megnyitásakor a Sparkle azt rögtön lefordítja, és az esetleges hibákról hibaüzenetet ad. A modulokról többek között megtudhatjuk, hogy milyen függvényeket illetve struktúrákat tartalmaznak, pontos típussal és definícióval.

- A *Section Center* feladata a Sparkle szakaszainak kezelése. Az itt található szakaszok listáját a lemezre mentett szakaszok közül bővíthetjük, és az egyes szakaszokat el is menthetjük. A lista valamelyik elemére kattintva megkapjuk az abban definiált állításokat, majd innen továbbléphetünk az egyes állítások bizonyításainak részletezéséhez. Az állítások listájában az egyes állítások mellett egy pipa jelzi, ha annak a bizonyítása teljes és helyes. A bizonyítások félbehagyhatók, és később folytathatók. Ha egy szakasz más szakaszok állításaira hivatkozik, a Sparkle azokat automatikusan betölti. Betöltés közben a Sparkle mindegyik állítást ellenőrzi, és ha az állításban vagy a bizonyításában a Project Centerben éppen megnyitott forrásfájlokra nézve valamilyen hiba van, azt azonnal jelzi. Ekkor megszakíthatjuk a beolvasást, de dönthetünk úgy is, hogy a problémás állítást figyelmen kívül hagyjuk, és tovább folytatjuk a műveletet.

- A *Tactic window* a bizonyítások soránfelhasználható taktikák rögzített listáját tartalmazza. Ha folyamatban van egy bizonyítás, valamelyik taktikára kattintva azt a bizonyítás következő lépéseként haszálhatjuk. Ha a taktika a jelen állapotban nem alkalmazható, hibaüzenetet kapunk, ellenkező esetben pedig egy új ablakban állíthatjuk be a taktika alkalmazásának paramétereit.

- A *Proof window* mutatja az éppen futó bizonyítás pillanatnyi állapotát. Itt megtalálhatjuk a bevezetett változókat, a hipotéziseinket, az ezek alapján bizonyítandó részcélt, és a későbbi lépésekben bizonyítandó részcélokat.

A további ablakok, mint a *Tactic Suggestions Window*, a jelen feladat megoldásának szempontjából nem érdekesek.

A felület és a program szerkezete alapján a feladat megoldhatónak látszik, hiszen ahhoz be kell azonosítani a projektek betöltéséért felelős kódot, és a szakaszok betöltéséért és ellenőrzéséért felelős kódot, majd a megadott projektre és szakaszra a két műveletet egymás után alkalmazni kell. Ha sikeres volt a művelet, akkor meg kell keresni és ki kell listázni a bizonyított állításokat. A műveletek végrehajtása közben a hibaüzeneteket a párbeszédablakok helyett egy fájlba kell írni.

A Sparkle felületének felépítését valamilyen szinten a program forráskódja is követi, és jól beazonosíthatók a főbb fogalmakat leíró programstruktúrák. A feladat szempontjából fontosabb típusok a *Project*, a *Section* és a *Theorem*. A *Project* és a *Section* típusok kezelését egy-egy programmodul segíti, amelyekben, többek között, az adott struktúrák fájlból való beolvasásának művelete is megtalálható.

A kód tüzetesebb vizsgálata során kiderül, hogy a grafikus elemek (általában párbeszédablakok, illetve az éppen futó művelet részleteit mutató státuszablak) viszonylag mélyen be vannak ágyazva az adott struktúrák típusműveleteinek kódjába, amelyet így feltétlenül át kell írni.

4.4.2. A szakaszok szerkezete

Egy Sparkle szakasz állításokat, és azok teljes vagy részleges bizonyításait tartalmazza. Ahhoz, hogy az állításokat és a bizonyításaikat értelmezzük, az ott megadottakon kívül további információra van szükség. Mint már volt róla szó, egy állítás bizonyításához más szakaszokban definiált állításokat is felhasználhatunk (olyanokat is, amelyeket még nem bizonyítottunk). Ezeket a taktikák felhasználása során csak a nevük azonosítja, de ez nem elég pontos információ az adott tételek beazonosításához. Hasonló a helyzet az állításokban felhasznált programszimbólumokkal, amelyeket szintén fontos a nevükön kívül pontosabban azonosítani. A problémák Sparkle általi megoldását jól illusztrálja az elmentett szakaszt tartalmazó fájl formátuma.

Egy szakasz fájl három rögzített részből áll. Az első rész a szakasz külső függőségeit írja le. Az előbb leírtak alapján itt a másik szakaszokból felhasznált állítások, valamint a hivatkozott programszimbólumok listája szerepel. A hivatkozások nem különösebben pontosak, aminek vannak egyaránt előnyei és hátrányai is. Az állítások nevén kívül itt csak az azt tartalmazó szakasz nevét találjuk, a szimbólumoknál pedig mindössze a típusuk szerepel. Ennek az az előnye, hogy az implementációt, illetve az egyes felhasznált tételek bizonyításait szabadon változtathatjuk. Ha az új implementációval vagy állítással valamelyik tétel ugyanazokkal a bizonyítási lépésekkel már nem bizonyítható, az állításunk érvénytelen lesz, de megmarad, így az implementációs hibát korrigálhatjuk, vagy ahhoz új bizonyítást készíthetünk. A megoldás hátránya ugyanebből következik. Könnyen előfordulhat, hogy csupán véletlenségből nyitjuk meg a szakaszt más forráskóddal, mint amivel a bizonyításokat végeztük, például ha a kódból több verziónk is van. Ekkor előfordulhat, hogy a nehezen megalkotott bizonyításainkat elvesztjük, ha a szakaszt az így kapott érvénytelen állításokkal újból elmentjük.

A szakaszfájl második része az ebben a szakaszban definiált állításokat tartalmazza. Egy állítás leírása a `THEOREM` kulcsszó után az állítás nevét, majd magát az állítást tartalmazza. Az állítás leírásában a felhasznált szimbólumok mellett az előző rész valamelyik sorára mutató referencia szerepel. Néhány függvény illetve reláció Sparkle-beli reprezentációja egy kódtábla, ami a szakaszfájlban is felváltja az eredeti szimbólumot. A leírásnak ez a két tulajdonsága elég ahhoz, hogy az állítások az emberi olvasó számára nehezen követhetővé váljanak.

A szakaszfájl harmadik, utolsó részében találjuk a bizonyításokat. Egy bizonyítás leírása a tétel nevével kezdődik, majd a bizonyítás során felhasznált tételek fájlbeli sorszámai következnek (az első és a második részben felsorolt állításokat egy folyamatos listaként kezelve). Maga a bizonyítás bizonyítási lépések számozott, hierarchikus listájaként szerepel. Egy bizonyítási lépés az alkalmazott taktika nevét, és az alkalmazásához megadott paramétereket tartalmazza. A hierarchiában akkor lépünk beljebb, ha valamelyik bizonyítási lépés az aktuális célt több részcélra bontja, amelyek mindegyikét bizonyítani kell ahhoz, hogy az eredeti célt bizonyítottnak nyilváníthassuk. A részcélok (és velük a bizonyítási lépések) így egy fát adnak. A Sparkle megköti az egyes részcélok bizonyításának sorrendjét, azokat csak a fát preorder módon bejárva bizonyíthatjuk. Ez lehetővé teszi azt, hogy egyértelműen azonosíthassuk azt a pontot, ahol egy bizonyítást félbehagytunk. Egy félbehagyott bizonyításnál következő, meg nem adott bizonyítási lépés helyén egy „*" karakter szerepel.

A bizonyítási lépésekként megadott taktikák között az interaktív bizonyítás során felhasználható taktikákon kívül szerepelhet egy *Axiom* nevű taktika is. Ezt azonban nem lehet rosszhiszeműen kihasználni, mert a taktikákhoz hasonlóan az axiómák is egy rögzített lista elemei lehetnek csak. A taktikák és az axiómák listáját a Sparkle nem külön fájlokból veszi, hanem azok a forráskódban lettek rögzítve. Ez a megoldás biztonságos ugyan, ám nem túl rugalmas.

4.4.3. Az elkészítendő eszköz specifikációja

A feladatot ellátó eszköznek más, parancssoros programokkal kell együttműködnie, ezért maga sem tartalmazhat interaktív elemeket. A paramétereit a parancssoron keresztül kell fogadnia, a végeredményt pedig fájlokba kell írnia, hogy azt az őt meghívó programok egyszerűen feldolgozhassák.

A Sparkle elvárja, hogy az elmentett szakaszokat a programfájlhoz viszonyítva a *Sections* könyvtárban találja. Mivel az eszköz a Sparkle kódjára épül, ezt itt is meg kell követelni.

Az eszköz tehát a következő szintaxis szerint indítható:

`sparkle_check` <*projektfájl*> <*szakasz*> [<*opciók*>]

ahol a *projektfájl* lehetőség szerint egy abszolút elérési út, a szakasz pedig egy olyan szakaszfájlra mutat, amely az eszköz saját *Sections* könyvtárában található. A szakaszfájl nevéből el kell hagyni a „`.sec`” kiterjesztést, az elérési utat pedig a *Sections* könyvtárhoz relatívan kell megadni.

Az opciók „opciónév=érték” formájúak lehetnek. Jelenleg a következő opciók adhatóak meg:

- *IgnoreErrors*, amelynek az értéke *Y* vagy *N* lehet, az alapértelmezett a *N*. Ha ez *N*, akkor hiba esetén a program futása megszakad. Ha *Y*, akkor hiba esetén a hibaüzenetet megjelenítése után a program tovább dolgozik, így esetleg még például további helyes állításokat találhat.

- *Silent*, amelynek az értéke szintén *Y* vagy *N* lehet, az alapértelmezett itt is a *N*. Ha a paraméter értéke *Y*, akkor nem jelennek meg a hibaüzenetek.

- *OutputDir*, aminek értéke tetszőleges létező és írható könyvtár lehet. Az eszköz ebbe a könyvtárba írja a végeredményt és a hibaüzeneteket tartalmazó fájlokat. Ha nincs megadva, akkor a program az aktuális könyvtárba dolgozik.

Az *OutputDir* kivételével az opciók többször is megadhatók. Ekkor a legutolsóként megadott érték lesz érvényben.

Az eredményfájl neve „`sparkle_check.ok`”, és ezt mindig létrehozza a program. A fájl a megadott szakaszfájl azon állításait tartalmazza, amelyek bizonyítása az adott projekthez tartozó forráskódra nézve helyes. A fájl formátuma megegyezik a két réteg közötti információcseréhez használt archívumban a *Theorems* könyvtár alatt található *Proved* fájl formátumával.

A hibaüzeneteket tartalmazó fájl neve „`sparkle_check.err`”, és ez csak akkor jön létre, ha van hibaüzenet. Az első sora egy angol mondatban leírja, hogy hány hibát találtunk, majd a megadott számú hibaüzenet következik.

4.4.4. A megvalósítás részletei

A Sparkle kódjának felépítése az ObjectIO csomag által sugallt módon a műveleteket egy többablakos (MDI) ObjectIO folyamat köré szervezi. Az inicializációs fázis után a felületi elemek eseményei határozzák meg, hogy milyen kód fusson. A műveletek során nagyon sok információt kell egyes függvényeknek megkapniuk. Ezeknek az adatoknak a legnagyobb részét a Sparkle az ObjectIO programállapot lokális komponensében tartja, így legtöbbször elegendő a függvényeknek programállapoton kívül csak néhány paramétert átadni.

Annak érdekében, hogy az új program struktúrája minél inkább hasonlítson az eredeti kódéra, a *sparkle_check* eszköz is egy ObjectIO folyamatot indít, de a Sparkle-lal ellentétben NDI módút, azaz nem engedélyezi a dokumentumablakok használatát. Párbeszédablakok azonban még így is megjelenhetnek. Ennek elkerülésére a több belső típusműveletnek is el kellett készíteni az eredeti alapján egy új verzióját, amely státuszüzeneteket egyáltalán nem közöl, a hibaüzeneteket pedig megjelenítés helyett átadja a hívó függvénynek.

Az így kapott műveletekre alapozva már nekiláthatunk a program lépéseinek megadásához. Ezek a lépések jól meghatározott sorrendben hajthatók végre. Az ObjectIO folyamattal ezt úgy egyeztethetjük össze, hogy a lépéseket az incializáló függvényben adjuk meg, és az inicializáció végén, a folyamat végtelen ciklusának beindítása előtt leállítjuk a folyamatunkat.

A parancssor értelmezése után az első lépés, hogy az ott megadott könyvtárban létrehozzunk egy üres eredményfájlt, és töröljük a hibafájlt, ha az esetleg létezik. Ez azért fontos, mert a hívó program a fájlok megléte alapján tudja eldönteni, hogy sikerült-e a programot elindítani, és hogy voltak-e hibák a működés során.

Következő lépésként egy kezdeti programállapot létrehozása után beolvashatjuk a megadott projektfájlt és a szakaszt. Ezeknél a lépéseknél dől el, hogy a forráskód maga helyes-e, illetve az állítások és a bizonyítások arra nézve értelmezhetők-e. Az erre készült műveletek átírt változatának köszönhetően ha ez nem teljesül, vagy a műveletek során más jellegű hibába ütközünk, akkor az okot hibaüzenetek formájában megkapjuk, amit később a hibafájlba írhatunk.

Ha a műveletek sikeresen voltak, akkor nincs más dolgunk, mint hogy a megnyitott szakaszok listájából, amit a program lokális állapotában találhatunk, kikeressük azt a szakaszt, amit paraméterként megkaptunk, majd abból kiválogassuk azokat az állításokat, amelyek bizonyítása helyes. Ezt az információt az állítást leíró típus tartalmazza, nem kell tehát a bizonyítási lépéseket végigkövetni. Ahhoz, hogy az így kapott állításokat a megfelelő formátumban az eredményfájlba írjuk, szükségünk van az állítás szövegére, amit szerencsére a `Theorem` típus szintén tartalmaz. Az állítások sikeres kiírása után a hibákat is a hibafájlba írjuk, ha ezt a megadott parancssori opció engedélyezi.

4.5. A BatchBuild2 eszköz

A *BatchBuild2* eszköz a hitelesítő réteg munkájában játszik szerepet, ott dinamikus kódot generáló program futási időben történő előállításához használjuk. Az eszköz a Clean rendszer részét képező *BatchBuild* eszközből apró módosítással keletkezett.

A *BatchBuild* eszköz a CleanIDE parancssoros változata. Feladata az, hogy egy projektfájlban tárolt információk és a forráskód alapján automatikusan elkészítse a megfelelő programot. A program az eredeti koncepció szerint szabadon hordozható, ezért nem használja a Clean fejlesztőkörnyezet beállításait. Ahhoz viszont, hogy a program a prototípusban elláthassa a feladatát, annak használnia kell a fejlesztőkörnyezetben definiált CleanIDE környezeteket (egészen pontosan azok közül egyet).

A *BatchBuild* eszköz kezel ugyan CleanIDE környezeteket, de az azt leíró fájlt az aktuális könyvtárban keresi. A programot tehát igen egyszerű volt úgy módosítani, hogy az a „beépített” környezetekkel dolgozzon.

A *BatchBuild2* eszközt a sikeres használathoz a Clean fejlesztőkörnyezet főkönyvtárába kell tenni. A programnak egyetlen paramétere van, a projektfájl. A végeredményként előállított programot az eszköz a projektfájlban megadott helyre teszi.

4.6. Különbségek az általános architektúrához képest

A prototípus felépítése és az általános architektúra közötti leglényegesebb különbség, hogy míg az általános architektúrában az absztrakt gépi kód a kódelőállító rétegben születik, a prototípus azt a hitelesítő rétegben generálja.

Ennek egyik oka az, hogy az általános architektúra szerint a hitelesítő réteg egyik feladata, hogy ellenőrizze, hogy az absztrakt gépi kód valóban a forráskód helyes fordításával állt elő. A feladat teljes megoldása két módon képzelhető el. Az egyik hogy az absztrakt gépi kódból visszakövetkeztetünk a forráskódra. Habár a gépi kódból kiindulva sokmindent ellenőrizhetünk (erre jó példa a Java nyelv bájtkód ellenőrző eszköze), teljes biztonságot adó ellenőrzés így általánosságban nem valósítható meg. Az vagy nagyon speciális nyelv és gépi kód esetén képzelhető el, vagy úgy, ha a gépi kód jegyzetekként tartalmaz a forráskódra utaló információt. Ez utóbbi eset viszont nem sokban különbözik a másik irányú, a forráskódból kiinduló megközelítéstől.

Ha a forráskódból indulunk ki, akkor a legegyszerűbben megvalósítható módon úgy ellenőrizhetjük a forráskód és a gépi kód kapcsolatát, ha a forráskódot lefordítjuk, és az így kapott gépi kódot a kívülről érkezett gépi kóddal összehasonlítjuk. Más megoldások megvalósítása lényegesen körülményesebb lenne, ellenben valószínűleg nem lenne sokkal hatékonyabb.

Így tehát az egyik leglogikusabb megoldás, ha lefordítjuk a forráskódot. Mivel a jelen esetben megkapjuk a fordítási paramétereket is, a két módon kapott absztrakt kód összehasonlításakor csak a teljes egyezés lenne elfogadható. Így tehát két teljesen megegyező fájlhoz jutnánk, amelyek közül a kódelőállítótól kapott változatot akár meg is spórolhatjuk.

Ha lenne hatékony általános módszer a forráskód és a generált absztrakt kód összehasonlítására, a dinamikus Clean kód jelenlegi, kísérleti változatára azt dokumentáció híján nehéz lenne implementálni, ráadásul az ezt megvalósító kódot sűrűn kellene újraírni a Clean Dynamics változásai miatt. Az is akadálya lenne ennek a megoldásnak, hogy a Sparkle jelenleg nem kezel dinamikus kódot, így arra nem tudnánk állításokat kimondani és bizonyítani.

További különbség az általános architektúrához képest, hogy a prototípus az ott előírt típusellenőrzést sem végzi el. Ez a prototípus egyik hiányossága. Az ellenőrzés megvalósításához a megadott definíciós modulból ki kellene keresni az objektum deklarációját vagy definícióját, és az ott szereplő típust kellene összevetni a megadottal. Ha a modulban a típus nem szerepel, akkor a Clean fordító típuslevezető kódjára támaszkodva azt meg lehetne találni.

A prototípus egy másik, már említett hiányossága, hogy a hitelesítő réteg a készített archívumot nem látja el tanúsítvánnyal. „Mentségként" felhozható, hogy ez a funkció inkább a hitelesítő réteg és a kód felhasználója közötti kapcsolatban játszik szerepet.

Egy további különbség, hogy az eredeti elképzelés szerint az állítások nem külön fájlban, hanem a forráskódhoz csatolt jegyzetekként lennének megadva. Habár ennek a megoldásnak rengeteg előnye lenne, ahhoz a fejlesztőeszközök, a fordító és a tételbizonyító támogatására (vagy teljes újraírására és integrációjára) lenne szükség. Ennek a különbségnek a hátterében tehát a felhasznált nyelv és eszközök tulajdonságai vannak.

5. fejezet

Korlátozások és további lehetőségek

A fejezetben összefoglalom a feladatra adott megoldás elméleti korlátait, és a prototípus hiányosságait, valamint megjelölök néhány irányt a továbblépéshez.

5.1. Korlátozások és hiányosságok

Az elkészített prototípus sok hiányossággal és korlátozással bír. Ezek egy része a felhasznált eszközök korlátaiból következik, más részük mögött elméleti akadályok vannak, jónéhány viszont még megvalósításra, illetve javításra vár.

Az egyik legsúlyosabb korlátozás, hogy az előállított dinamikus kód csak ugyanazon a gépen használható fel, amelyiken létrehozták, ami a jelen megvalósításban a hitelesítő réteg eszközeit futtató számítógép. Ez a korlátozás a Clean nyelv új lehetőségének a dinamikus kódnak a hiányos, kísérleti szakaszban lévő megvalósításából következik. A dinamikus kód hordozhatóságának megoldásán már dolgoznak a nyelv fejlesztői.

A Clean nyelv néhány fontos elemét nem támogatja a megvalósítás. Többek között nem lehet unique típusú és dinamikus típusú objektumokra, valamint az ObjectIO csomagot használó programokra állításokat megfogalmazni. Ez ugyan a Sparkle korlátozásának következménye, a háttérben azonban elméleti problémák találhatóak. Ahhoz, hogy a Sparkle támogassa a unique típusokat, a rendszert további taktikákkal kellene bővíteni, amelyek pontos meghatározása alapos átgondolást igényel. A dinamikus típus támogatásához vagy alapjaiban meg kellene változtatni a Sparkle forráskód-kezelését, vagy a dinamikus kód mellé kellene tudni csatolni a Sparkle információit. Az ObjectIO csomagot használó programok jellemzéséhez pedig nem elég az elsőrendű logika, a Sparkle-t ehhez temporális logikai tudással kellene felvértezni.

A pótolandó hiányosságok között található például a tanúsítvány hozzáadása a hitelesítő réteg eredményéhez, a forráskódból a dinamikus kód előállításához és az állítások ellenőrzéséhez feltétlenül szükséges fájlok kiszűrése, és csak azok továbbítása a kódelőállító rétegből, vagy a típusellenőrzés elvégzése a hitelesítő rétegben.

5.2. További lehetőségek

A CPPCC architektúra Clean nyelvhez illesztésének tovább finomítására, és a prototípus korlátainak feloldására, illetve további tudás hozzáadására bizonyos feltételek és feladatok mellett sok lehetőség van.

A Clean Dynamics tökéletesedő implementációjának köszönhetően várhatóan sok innen eredő korlátozás feloldhatóvá válik. Az egyes korlátozások mindegyikénél át kell gondolni azonban, hogy milyen hatással lenne a feloldása az architektúrára, illetve azt milyen feltételek mellett lehet biztonságossá tenni.

A CPPCC architektúra azon elképzelése, hogy az állításokat a forráskódban jegyzetekként szerepeltessük, sok előnnyel járna, ám a felhasznált eszközök támogatása és azok integrációja lenne hozzá szükséges. Még jobb lenne, ha dinamikus kód is támogatná a kódhoz fűzött jegyzeteket, és oda az állításokat úgy lehetne beilleszteni, hogy azok direkt a hivatkozott szimbólumok kódbeli definíciójára mutathassanak. Ekkor többek között fel lehetne oldani a nevek túlterhelésének tiltását, és ez nagyban hozzájárulna ahhoz, hogy a Sparkle is támogassa a dinamikus típusokat.

Az ObjectIO elemei által felvetett problémákra példa lehet az ott definiált, általam is használt véletlenszámfüggvény, a `rand`. Erre a paraméter nélküli függvényre nem teljesül pédául az egyik legalapvetőbb tulajdonság, a hivatkozási átláthatóság, hiszen az egy paraméter nélküli függvény esetében azt jelentené, hogy annak az értéke konstans. Észrevehetjük viszont, hogy a függvénynek van egy implicit paramétere: az időpillanat (illetve kettő is, az idő és a futtató szál azonosítója, de a Clean jelenleg nem támogat konkurrenciát), amelynek felhasználásával már a hivatkozási átláthatóság is teljesül. Ez a probléma egyenesen mutat a Sparkle temporális logikai tudással való kiterjesztése felé. Az itt bemutatott architektúra ilyen irányú bővítéséhez mást nem kell tenni, csak a megfelelő eszközt a leírt eljárás alapján újra előállítani, hiszen az továbbra is Sparkle állításokat és tételeket kezelne a Sparkle tudásának felhasználásával.

6. fejezet

Zárszó

A kitűzött feladatot sikerült megoldani: elkészült egy prototípus, amely a CPPCC architektúra két rétegének legtöbb feladatát elvégzi, és a közöttük lévő információcserét egy jól használható adatformátumon keresztül segíti.

A megoldandó feladat elemzése során sok lépés szinte magától értetődően következett a felhasznált eszközök tulajdonságaiból, míg máshol sok energiát kellett abba fektetni, hogy egyes korlátozásokat elkerülhessünk.

Habár a prototípus sok korlátozással és hiányossággal rendelkezik, amelyek egy részéről már a feladat kitűzésekor tudni lehetett, mégis próbáltam a megoldást a lehető legrugalmasabban megtervezni, és az általános architektúrát a lehető legkevesebb megszorítással a konkrét eszközökhöz alakítani. Törekedtem például a körülményekhez képest legnagyobb hordozhatóság elérésére annak ellenére, hogy a felhasznált eszközök korlátozásai miatt a megoldás jelenleg nem lehet hordozható.

A feladat megoldása során felvetődött néhány előre mutató kérdés is, amelyek megválaszolásához tovább kell lépni a mind a nyelv, mind a bizonyítórendszer tudásának fejlesztésében.

Irodalomjegyzék

[1] Rinus Plasmeijer, Marko van Eekelen: *Concurrent Clean Version 2.0 Language Report,* Technical Report, Computing Science Institute, University of Nijmegen 2001. december, `http://www.cs.kun.nl/~clean/`

[2] Paul de Mast, Jan-Marten Jansen, Dick Bruin, Jeroen Fokker, Pieter Koopman, Sjaak Smetsers, Marko van Eekelen, Rinus Plasmeijer: *Functional Programming In Clean.* Computing Science Institute, University of Nijmegen 1999. július

[3] Erik Barendsen, Erik, Sjask Smetsers: Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science* 6, 579-612. o., 1996.

[4] Pil, M.R.C.: Dynamic types and type dependent functions. *Proc. of Implementation of Functional Languages (IFL '98)*, Springer-Verlag, Lecture Notes in Computer Science 1595, 169-185. o., 1999.

[5] Maarten de Mol, Marko van Eekelen, Rinus Plasmeijer: Theorem Proving for Functional Programmers - SPARKLE: A Functional Theorem Prover. *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers*, Springer-Verlag, LNCS 2312, 55-71. o., 2001. szeptember

[6] L. C. Paulson: *The Isabelle Reference Manual*, Cambridge, 2001. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`

[7] A COQ fejlesztői: *The Coq Proof Assistant Reference Manual (version 7.3)*, Inria, 1998. `http://pauillac.inria.fr/coq/doc/main.html`

[8] Maarten de Mol: Verified proofs concerning functional programs. *Proceedings of the Nijmegen Student Conference on Computing Science*, 69-82. o., 1998.

[9] Maarten de Mol: *Clean Prover System*, diplomamunka, University of Nijmegen, 1998.

[10] Horváth Zoltán, Peter Achten, Kozsik Tamás, Rinus Plasmeijer: Proving the Temporal Properties of the Unique World. *Software Technology, Fenno-Ugric Symposium FUSST'99 Proceedings, Technical Report CS 104/99*, 113-125. o., 1999.

[11] Horváth Zoltán: *Párhuzamos programok relációs programozási modellje*, doktori disszertáció, Eötvös Loránd Tudományegyetem, Budapest, 1996.

[12] Horváth Zoltán, Kozsik Tamás: *Safe Mobile Code - CPPCC: Certified Proved-Property-Carrying Code*, position paper, Eötvös Loránd Tudományegyetem, Budapest, 2001.

[13] *JAR File Specification*, `http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html`

[14] George C. Necula: *Compiling with Proofs*, doktori disszertáció, Carnegie Mellon University, 1998.

[15] Christopher Colby, Peter Lee, George C. Necula: A Proof-Carrying Code Architecture for Java. *Computer Aided Verification*, 557-560. o., 2000.

[16] Manfred Broy, Ursula Hinkel, Tobias Nipkow, Christian Prehofer, Birgit Schieder: Interpreter Verification for a Functional Language. *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer-Verlag, 77-88. o., 1994.

[17] Tobias Nipkow, David von Oheimb, Cornelia Pusch: μJava: Embedding a Programming Language in a Theorem Prover. Technische Universität München., 2000