Az egységes számítástudomány létrehozásának története

Gergely Tamás
Alkalmazott Logikai Laboratórium

1. Előszó.

A bonyolult jelenségek megértése, a bonyolult rendszerek megismerése, ok-okozati vagy szinkronicitási viszonyok feltárása iránt már egyetemi tanulmányaim során érdeklődést tanusítottam. Ez az érdeklődés tovább erősödött a kutatói munkám során. A kutatásaimhoz a módszereket a rendszerszemlélet, a rendszerelmélet, illetve a kibernetika biztosította, valamint a minőségi, illetve mennyiségi összefüggéseket megragadó matematikai diszciplínák.

A bonyolult rendszerek iránti érdeklődésem egyik központi objektuma a számítógép és az ezzel különböző absztrakciós szinteken foglalkozó számítógéptudomány volt. Ezt az érdeklődésemet legalább négy szempont motiválta: (i) a számítógép mint komplex jelenség, mint vizsgálati objektum, (ii) a számítógépek tudományának formálása, illetve egy új elméleti alapokra épülő komplex számítógéptudomány kifejlesztése, (iii) a számítógépek intelligensebbé tétele, új generációs számítógépek kifejlesztése és (iv) a számítógépek, mint autonóm, illetve heteronóm intelligens rendszerek összetevői.

Ahhoz, hogy érthetőbbé tegyem a fentieket , egy rövid helyzetképet adok a számítástudomány akkori állapotáról.

A 60-as évek második felétől egyre több szó esett az un. software válságról. Be nem fejezett programfejlesztések, használhatatlan rendszerek, az átfutási idők be nem tartása, rendkívül magas programfejlesztési költségek jelezték a válságot. A válság okai között szerepelt az, hogy egyre nagyobb és bonyolultabb rendszereket kellett fejleszteni egyre szigorúbb helyességi kritériumok betartása mellett. Így a bonyolult valós időben dolgozó rendszereknek, mint pl. a repülőtéri irányító, vagy a rakéta-irányító rendszereknek, igen bonyolult körülmények között kell dolgozniuk és hibáik rendkívül nagy veszteségeket jelentettek. Ismert példa, hogy az amerikaiak által fellőtt első Venus szondát egy triviális software hiba miatt jelentette elveszettnek az irányító rendszer. Természetesen a kevésbé kiélezett feladatok megoldására szolgáló programok esetében is egyre fontosabb elvárás lett a helyesség. Ennek egyik oka az volt, hogy szétvált egymástól a program fejlesztője és felhasználója. Így a felhasználó a felmerülő szoftverhibák esetén már tehetetlen volt.

A krízis tünetei alapján elsősorban tesztelési válság forgott fenn, megbízhatatlanok voltak a rendelkezésre álló módszerek. A tesztelés csak a falszifikálásra volt alkalmas, a helyesség bizonyítására azonban nem. A tesztelés csak a hibák jelenlétét mutatta ki, és nem azt, hogy az adott programban nincsenek hibák. Pl. az Apolló 14 tíznapos repülése alatt a vezérlő software 18 hibáját kellett menet közben kijavítani.

A válság felerősítette egyrészt a programfejlesztés módszereinek kutatását, másrészt az ehhez nélkülözhetetlen számítástudományi kutatásokat. Hiszen a szoftver „gyártás” technológiájának kidolgozása a műszaki tudományoknak megfelelő számítógéptudományt, vagy számítástudományt igényelt. Tehát a kutatások célja a formális eszközök és az ipari termelésre emlékeztető programmozási technológia kidolgozása volt. Itt a programok helyességének ellenőrzésére a tesztelés helyett minőségileg új közelítésre volt szükség. Ezt kívánta biztosítani a programhelyesség bizonyítás. A programhelyesség bizonyítás a specifikáció és a program

formalizált eszközökkel való összevetése. Így a tesztelés mellett, vagyis inkább helyett fontos célként jelent meg a specifikációkhoz képest helyes programok létrehozhatósága.

A formalizálás igénye a legnagyobb nehézséget a program jelentésének leírásánál okozta. Ennek elvégzéséhez a programozási nyelvek egzakt szemantikájára lett volna szükség. Azonban a programozási nyelveknek általában csak a szintaktikája volt formálisan, egzaktan leírva, szemantikája - tehát elemeinek, konstrukcióinak jelentése csak természetes nyelven, matematikailag nem formalizált alakban állt rendelkezésre. Ezért a programok tulajdonságainak bizonyításához elsősorban a programozási nyelvek formális szemantikájának kidolgozása vált szükségessé. Az alapvető probléma ezzel kapcsolatban az volt, hogy a programozási nyelvek utasításainak jelentős része állapotváltoztatást (cselekvést) kezdeményezett, azaz felszólításokat. A szemantika e felszólítások eredményezte történések kijelentő módban való leírása. A matematika addigi fejlődése során a szemantika leíró eszközöket statikus esetekre, változtathatatlan modellekre dolgozták ki, mint ezt a klasszikus logikában láthatjuk. Jelentős és mély matematikai kutatás vált szükségessé a "parancs" elemeket tartalmazó, korrekt szemantikájú nyelvek kidolgozására. Abban az időben a szemantika kezelésére különféle formális apparátussal kísérleteztek. Ezek közül akkoriban a legígéretesebbnek az automata elméleti, a logikai szemantika és a rekurzív függvény egyenleteken alapuló un. fixpont szemantika ígérkezett.

Egyúttal megjelent az igény új elvű programozási nyelvek iránt, amelyek a "parancs" típusú utasítások helyett leírást, deklarációkat használnak a programok szintaktikai egységeként. Azaz, a parancsok helyett leírást adnak a kitűzött célról, a "hogyan" helyett a "mit"- adják meg.

Tehát abban az időben, a hetvenes évek elején, a szoftver krízis megoldása érdekében egyre sürgetőbbé vált új programfejlesztési módszerek kidolgozása és alkalmazása. Ezen belül időszerűvé vált a programok helyességbizonyítása, illetve a bizonyítottan helyes módon való tervezésük és implementálásuk. Ennek a körülménynek a széleskörű felismerése magyarázza a témával kapcsolatos intenzív kutatásokat. Amikor én elkezdtem a téma iránt érdeklődni ezek a kutatási területek meglehetősen divergálóak voltak, számos új közelítésmód jött létre, de a kitűzött cél elérése meglehetősen távolinak tűnt. Mi is volt ez a cél? Erős elméleti és módszertani alapok megteremtése, amely alkalmas a programozás problémáját kellő általánossággal, újfajta módon kezelni, és így lehetőséget ad az addigiaknál hatékonyabb megoldások elérésére, különösen a gyakorlat által egyre nagyobb számban igényelt nagybonyolultságú rendszerek vonatkozásában. Azonban akkortájt a helyzet inkább Bábel tornyára hasonlított a sokféle és szerteágazó, de egymással kommunikálni képtelen megoldási javaslatok miatt. És ezek a javaslatok csak részleges megoldást kínáltak egy-egy részproblémára koncentrálva.

Gyakorlatilag ezzel a helyzettel szembesültem, amikor a számítástudomány iránt elkezdtem érdeklődni.

A számítógéptudomány az én kutatásaimban is végigjárta a maga fejlődési útját a számítógépek működésének és felhasználásának modellezésétől az intelligens partnert biztosító számítógépek kialakításáig és ennek az útnak a fontosabb állomásait szeretném bemutatni. Ez azért is érdekes mivel egy jól strukturált egységes számítástudományt ismertetek annak fontosabb fejlődési szakaszaival együtt. Ennek a számítástudománynak a legfontosabb jellemzője, hogy egységesen az első rendű matematikai logikára épül.

Ezt az utat nem egyedül jártam be, minden egyes szakaszában voltak társaim, kollégáim, akik közül néhányat meg is fogok nevezni. De a többiek is fontos szerepet játszottak, többek között a vitákban való aktív részvételükkel és az inspiráló munkahelyi légkör alakításában. Ezek nem kevés befolyással bírtak az egyes elméletek alakulására.

Tehát itt a matematikai logikára épülő egységes számítástudomány kialakításának folyamatát mutatom be. Mint minden kutatás, ez is meglehetősen szerteágazó, annál is inkább mert maga a terület, amivel foglakozunk önmagában is többrétegű és sok összetevőből formálódik egy egységes formális rendszerré. Az új, meglehetősen komplex elmélet kialakításának történetét öt szakaszra bontva mutatom be, amelyeken keresztül feltárulnak az olvasó előtt azok az ötletek, amelyek segítettek közelebb kerülni az elérendő célhoz. A szakaszok sorrendje nem jelent időbeli egymásutániságot, mivel az egyes szakaszok tematikai egységet jelentenek, amelyek művelése akár párhuzamosan is történhetett. Az első szakasz az első rendű logika formalizmusának erejét és alkalmazhatóságának határait kereste. A második szakasz a logika egy résznyelvére épülő programozási paradigmával, a logikai programozással foglakozott szigorúan az első rendű logika keretein belül keresve az egységes megalapozás formalizmusát és módszereit. A harmadik szakasz a logikai programozást is magába foglaló un. deklaratív programozás logikai megalapozásával foglalkozott. Ezen belül mutatom be a negyedik szakaszt, amely a programozás-elmélet axiomatikus halmazelméleti alapjainak kidolgozásával foglalkozott megőrizve az első rendű logikai kereteket. Ez a halmazelméleti konstrukció döntő fontosságúnak bizonyult az egységes számítástudomány kialakítása során. Az ötödik szakasz a programok specifikációjának matematikai alapjait vizsgálta és egy egységes formális elméletet és módszertant hozott létre. Az egységes számítástudomány létrehozásának utolsó állomása, a hatodik szakasz a programok és programrendszerek létrehozását, az un. szoftver mérnökséget megalapozó elméletet és erre épülő módszertant dolgozta ki. Végül összefoglalom a hat szakaszban felépített és az egységes logikai alapokra épülő integrált számítástudományt.

Az egységes matematikai logikai alapokra épülő számítástudománnyal a történet még nem ért véget. Nevezetesen az új elvekre épülő számítástechnikai megoldások kidolgozása jelentette a következő kihívást. Ez olyan számítási rendszerek kidolgozását célozta meg, amely intelligensebb támogatást nyújt a felhasználóknak, mint a kész programokat futtató számítógép rendszerek. Ezeknek az intelligens megoldásoknak a vizsgálata a mesterséges intelligencia tárgykörébe tartozott, célja pedig az un. ötödik generációs számítógép rendszerek kifejlesztése volt. Ez azonban már egy másik önálló történet.

Az egységes számítástudomány kidolgozásának történetét a könnyebb érthetőség érdekében két szinten ismertetem. Az első szinten a konceptuális leírás dominál, ami mellett néhány tudománytörténeti elemet és viszonyt is megadok. A második szinten kísérletet teszek arra, hogy az elért eredmények legalább egy kis töredékének a lényegét ismertessem.

2. Az első szakasz az egységes számítástudomány felé vezető úton

2.1. Általános megfontolások.

A számítástudomány területén az érdeklődésem középpontjába a programok és a programozás került. Pontosabban fogalmazva az elméleti számítástudománynak az az ága, amelyet hol program-elméletnek, vagy programok elméletének, hol programozás-elméletnek neveztek. Ennek a területnek a feladata a programok, illetve a programozási nyelvek vizsgálata, valamint ezekkel kapcsolatos tudományos érvelések egzakt matematikai alapon nyugvó kezelése. A számítástudomány fejlődésének kezdetén az első formális eszközök alapvetően a programokat szintaktikai szempontból vizsgálták. Nevezetesen a „hogyan" specifikációval kapcsolatos kérdésekre keresték a formális választ: arra, hogy hogyan kell egy programot megadni, hogyan épül fel egy program, mennyire optimális, mennyire komplex, stb.

2.2. Konceptuális alapok.

A programok azzal, hogy algoritmusokat valósítanak meg önmagukban is speciális formális objektumok. Nevezetesen, a szintaktikai aspektuson kívül van egy jelentésük, amelyet a végrehajtásuk határoz meg. Tehát a programok olyan speciális formális objektumok, amelyek a programok végrehajtásával, azaz realizálásával kapcsolatos jelentéssel bírnak. Az igazi programok végrehajtása igazi számítógépekben történik, ami változásokat eredményez a számítógép memóriájában, azaz a futtatott programok adatkörnyezetében. A program-elmélet azonban elvont programozási nyelvekkel és programokkal foglalkozik és ennek megfelelően a programok végrehajtása egy absztrakt adatkörnyezetben történik, ami tulajdonképpen a számítógépeket reprezentálja. Az elsődleges kérdés, amikor a program jelentését vizsgáljuk az, hogy milyen jellegű változásokat eredményez egy program futása az adatkörnyezetében. A programozási nyelvek szemantikája a „mit" típusú kérdésekre keresi a választ, úgy mint mit csinál egy adott program, vagy mik a tulajdonságai a programnak. A programozás-elmélet központi kérdésköre a programok jelentésének formális megadása, a jelentéssel kapcsolatos különböző tulajdonságok leírására alkalmas nyelvek kidolgozása, valamint e nyelvek effektív bizonyítási rendszereinek meghatározása.

A programozás-elmélet megalapozására több egymástól eltérő formális keretet dolgoztak ki. Ezek elsősorban szemléletükben és közelítésmódjukban tértek el egymástól. Így eltértek az általuk használt fogalmakban, valamint a programozási paradigma értelmezésében. Ugyanakkor különböztek a matematikai eszközeikben és módszereikben, amelyeket arra használtak, hogy a programokat és a programtulajdonságokat definiálják, leírják és vizsgálják. A különböző megközelítések a különböző formális eszközök mellett más néző-pontokat is használtak a programok és adatkörnyezetük jellemzésére. A különböző formális módszerekkel és eszközökkel kapott eredmények többnyire nem voltak összehasonlíthatóak, a nyolcvanas évekre kialakult a számítástudomány „Bábel-torony" szerű helyzete. Erre a helyzetre és a kiút szükségességére több kutató is rámutatott a nyolcvanas évek elején. Felmerült a kérdés, hogy lehetséges-e egy olyan formális keret kidolgozása, amelyen belül mindkét szempont, a szintaktikai és szemantikai is vizsgálható, és a legtöbb fogalom és elért eredmény a kereten belül értelmezhető és összehasonlítható. Amikor ez a kérdés a nemzetközi irodalomban megfogalmazódott Magyarországon már néhány éve a válaszon dolgoztunk.

2.3. A kidolgozott programozás-elmélet

A programozás-elmélet megalapozásával foglakozó munkák egy része a megoldást a matematikai logikai alapokra építve képzelte el. A programokról való gondolkodást és érvelést elősegítő logikai alapú formalizmust programozási logikának nevezték. Az irodalomban abban az időben megjelent formális konstrukciók legnagyobb része az elsőrendű klasszikus logika kiterjesztése volt, mint pl. a Floyd-Hoare féle logika, a dinamikus logika, az algoritmus logika, a temporális logika, stb.

A programozási logikáknak egyik megkülönböztetési szempontja az volt, ahogyan a programokat kezelte a szintaxis. Pl. a Floyd-Hoare nyelvben és a dinamikus logikában a programokat modalitásként kezelték, az egyik algoritmikus logikában a programok, mint atomi formulák jelentek meg, míg egy másikban termként (kifejezésként) voltak jelen. Egy másfajta közelítésmód valósult meg az temporális (idő) logikában ahol a programok egyáltalán nem jelentek meg a képletekben, végig implicitek maradtak.

A megkülönböztetés egy másik szempontja a programok szemantikájának kezelési módján alapult. A legtöbb programozási logika a jelentést a programok futási eredményével reprezentálta, amit valamilyen input-output relációval adtak meg. Ezt nevezték denotációs szemantikának. Ugyanakkor gyakorlati szempontból egyre fontosabbá vált, hogy programokat ne csak a kiszámítási eredmény szempontjából jellemezzünk, hanem a kiszámítási folyamat viselkedése szempontjából is. Ehhez a teljes kiszámítási folyamatot figyelembe vették, amit az un. futás szemantikák biztosítottak. Az ezt a szemantikát használó logikák valamilyen formában a számítógép működésének modellezése során figyelembe vették az időt is. Ez utóbbi csoportba tartozó programozási logikáknak a különböző programjellemzők megadása érdekében alkalmasnak kellett lenniük a programok adatkörnyezetének, valamint tulajdonságainak az idővel kapcsolatos jellemzésére. Erre különböző lehetőségek kínálkoztak:

(i.) csak az adatok épülnek be közvetlenül a modellbe, az idő a metanyelvben jelenik csak meg;

(ii.) az idő az adatokkal együtt megjelenik a leírónyelvben, de a modellekben csak az adatok szerepelnek;

(iii.) mindkettőt, az adatokat és az időt is kezeli a leírónyelv és mindkettő beépül a modell szerkezetébe.

Megjegyzem, hogy a fentiekben említett programozási logikák az idő logika kivételével az (i) lehetőséget valósították meg.

Az abban az időben elérhető logikai alapokra épülő közelítésmód nagy része eltért az elsőrendű klasszikus logikától, vagy azzal, hogy nem voltak elsőrendűek, vagy azzal hogy nem-klasszikus logikai eszközöket használtak. Az, hogy mégis logikát használtak érthető, hiszen a matematikai logika az az egyetlen tudományág, amely jól fejlett kultúrával rendelkezik a szintaxis és a szemantika terén. Az elsőrendű klasszikus logikától való eltérés is érthető, hiszen a klasszikus logikai rendszerek statikusak miközben az algoritmusok, illetve az ezeket implementáló programok jelentése dinamikai szemléletű közelítést igényel. Ennek érdekében olyan eszközöket kellett kialakítani, amelyek lehetővé tették a programok dinamikai szempontjainak megragadását és alkalmasak voltak a futás reprezentálására.

Az első lépéseket a különböző közelítésmóddal való ismerkedés (l. pl. Gergely et al, [1975]) jelentette, amelynek során ezek hiányosságait tártuk fel (l. Andréka-Gergely-Németi [1977], Gergely, Szöts [1978] és Gergely-Ury [1978]). A különböző közelítés vizsgálata során felmerült az az igény, hogy a programozás-elmélet az elsőrendű logika keretein belül kerüljön kialakításra. Ezzel kapcsolatos koncepciómat 1978-ban a „Matematikai logika a számítástudományban” nemzetközi

konferencián adtam elő (l. Gergely, [1978]). Ez volt az első lépés egy tisztán klasszikus első-rendű logikai alapokra épülő közelítésmód kidolgozására. A célunk ezzel a közelítésmóddal az volt, hogy új alapokra helyezzük az elméleti számítástudomány programozás-elméletét, azaz pontos matematikai logikai alapokat kívántunk nyújtani a számítógépes programokról és programnyelvekről folyó gondolkodáshoz és érveléshez. Elsősorban logikai eszközöket kívántunk nyújtani ahhoz, hogy a programokat megfelelő formális objektumként definiálhassuk és kezelhessük. Így formálisan kívántuk leírni azt, hogy milyen tulajdonságokkal rendelkezik egy program, mit csinál ez a program és ezt hogyan csinálja. Ezzel kapcsolatos első javaslatainkat Gergely, Ury, [1978]-ban foglaltuk össze.

Az általam megfogalmazott koncepció abból a megfontolásból indult ki, hogy ki lehet dolgozni egy olyan formális eszköztárat, amellyel mind a szintaktikai, mind a szemantikai aspektus vizsgálható, a fogalmak nagy része definiálható, illetve más formalizmussal elért eredmények interpretálhatóak és összehasonlíthatóak. Ugyancsak elvárásként fogalmaztam meg, hogy a platform a klasszikus elsőrendű logikára épüljön.

A programok végrehajtásának, futásának jellemzése érdekében először a számítógépet modelleztük az elsőrendű logika klasszikus modelljein belül. A végrehajtás szempontjából egy számítógép fontos jellemzője az, hogy milyen adatokat tud értelmezni, és hogy ezeket hogyan kezeli, azaz hogy milyen transzformációkat tud végrehajtani, vagy az adatok milyen tulajdonságait képes feltárni. A kiértékelés fogalma teremt kapcsolatot a változók és a modellek univerzumának elemei között. Ezt használjuk a számítógép memória elemeinek reprezentálására Ezek segítségével definiálhatjuk az állapotokat az adatkörnyezetben, azaz a modellekben.

A programozási 1ogika azon változatai, amelyek az időt csak implicit módon kezelték, azaz csak a metanyelv szintjén, az időt a program végrehajtása, illetve a futások modellezése során a természetes számok, vagy pontosabban, az aritmetika standard modelljeinek segítségével írták le. Ám ugyanakkor, az adatokat az elsőrendű elméletben modellezték, ami azt jelenti, hogy ezzel a nem-standard adatokat szintén megengedték és kezelték. A "standard" és "nem-standard" kifejezéseket az aritmetika elsőrendű logikájának értelmében használom. Tehát itt egy bizonyos asszimetria állt fent a megfelelő programozás-elméletekben, ami azt eredményezte, hogy ezen elméleteknek a bizonyítási ereje nem volt elegendő, például a kapcsolódó programozási logika nem volt teljes.

Ugyanakkor, ha megadunk egy elsőrendű elméletet, amely explicit módon kezeli az időt akkor a standard időn túlmenően, megengedjük a nem-standard időt is. Így a programozás-elmélet szimmetrikus lesz és a kapcsolódó programozási logika pedig teljes.

A nem-standard idővel kapcsolatban megjegyzem, hogy egy program bizonyos modellekben úgy terminál, hogy kívülről nézve “végtelen hosszú” ideig fut. Ebben az esetben a program egy olyan időpillanatban áll meg, amely egy nem-standard szám. Ugyanakkor ugyanez a futás végtelen és nem termináló, ha kívülről nézzük. Ezért volt nagyon fontos a program futások megállásának kapcsán a standard és nem-standard modellek közötti összefüggést vizsgálni a programozás-elméletben.

Tekintettel arra, hogy közelítésmódunk a klasszikus elsőrendű logikára épült, a kiszámítás fogalmat a természetes számok halmazáról, azaz az aritmetika standard modelljeiről ki kellett terjeszteni a nem-standard modellekre is. Így a kiszámítás a nem-standard modellekben is jól definiált lesz, ami gyakran bizonyult hasznosnak.

A mi kutatásaink alapvető sajátossága az volt, hogy a programozás-elmélet a klasszikus elsőrendű logika keretén belül lett kialakítva szemben más programozás-elméleti kutatásokkal, amelyek

különböző kérdések vizsgálatára más és más formalizmust vezettek be. Ezzel szemben mi a kidolgozott egyetlen elmélet formális keretein belül vizsgáltuk az alapvető kérdéseket. Az egységes elméleten és a kapcsolódó módszertani kereteken belül kidolgozásra kerültek azok a módszerek, amelyek lehetővé teszik a programok szemantikájának leírását bármilyen programozási paradigmáról is van szó. Egy speciális definícióelmélet adta ehhez az egyik megfelelő módszert, míg a másikat a logikai kiterjesztés módszere biztosította. Ezek segítségével bármilyen új programozási konstrukció egyértelmű definíciójára lehetőség nyílt. Ezzel együtt már a felmerülő feladatok legnagyobb része a kidolgozott programozási logika keretein belül vizsgálhatóvá vált.

Az irányzat eredményeit Ury Lászlóval a nyolcvanas évek végére egy egységes matematikai elméletbe foglaltuk. Ez egy szigorúan elsőrendű logikára épülő programozás-elmélet volt, amely hatékonyan tudta integrálni a dinamikus- és időlogikák kifejező erejét a különféle program-konstrukciók megadása érdekében. Az elsőrendű logikai keretek között felépülő nem-standard programozás-elmélet teljességéhez hozzátartozott a következő három területen megfelelő elmélet kidolgozása: (i.) a kiszámításelmélet, (ii.) a programok elmélete és (iii.) a programok tulajdonságait leíró nyelvek ill. a programozási logikák elmélete. Ezeknek az elméleteknek a közös alapját a klasszikus elsőrendű logika biztosította. Az első területen a megfelelő kiszámításelmélet kidolgozása volt a feladat, azaz egy, a klasszikus elsőrendű logika keretein belüli kiszámításelmélet megalkotása. Az (i.) és (ii.) területekkel kapcsolatos kutatások során két kérdésre kerestük a választ. Az első kérdés arra vonatkozott, hogy az intuitíven kiszámítható modellbeli relációk hogyan viszonyulnak a különböző programozási nyelvekkel kiszámítható relációkhoz. Erre a kérdésre a választ a Church tézis általánosítása adta meg, amely szerint egy első-rendű nyelv egy modelljében hatékonyan kiszámítható relációk pontosan azok, amelyek egy általunk megadott programozási nyelvvel kiszámíthatóak. A második kérdés, amely átvezetett a (ii.) területre, arra vonatkozott, hogy a különböző programozási nyelvek közül melyik kiszámítási ereje a legnagyobb. Miután meghatároztuk a programok szemantikáját és definiáltuk a programozási nyelvek kiszámíthatóságát, lehetségessé vált az egyes programozási nyelvek kiszámítási erejének vizsgálata. Így a felvetett kérdésre válaszként megadtuk a különböző programozási nyelvcsaládok összehasonlító vizsgálatát. Ennek során vizsgáltuk a különböző programkonstrukciókkal bővített nyelvek kiszámítási erejének viszonyát. Megjegyezzük, hogy a programnyelvek bővítése tisztán logikai úton is biztosítható az alapjeleket megadó szignatúra bővítésével. Itt a hasonlósági típus bővítése e célnak megfelelő módon került bevezetésre. Az egy- és kétvermes programozási nyelveket az induktív és effektíven induktív definiáló sémák segítségével tisztán logikai úton adtuk meg és jellemeztük.

A harmadik terület a programok tulajdonságainak leírására szolgáló nyelvekkel foglalkozik a klasszikus elsőrendű logikai kereteken belül. Vizsgáltuk az input/output szemantikával kapcsolatos tulajdonságok leírására szolgáló dinamikus logikát és ennek Floyd-Hoare-féle résznyelvét. Különös figyelmet szenteltünk a teljesség kérdésének. Az idő implicit kezelése okozza a dinamikus logika nem-teljességét. Ezért bevezettük és explicit módon kezeltük az időt. Először a futás-szemantikai tulajdonságok vizsgálatára alkalmas temporális logikát adtuk meg, és vizsgáltuk alapvető tulajdonságait. Kifejező erő szempontjából bevezettük a dinamikus és temporális logikákat és megállapítottuk, hogy ezek általános esetben összehasonlíthatatlanok. Az elsőrendű keretünk megengedte az idő explicit kezelését. Erre szolgált az un. időlogika. Ennek keretén belül összehasonlítottuk az irodalomban található különböző programverifikálási módszereket. Az itt összefoglalt elmélet módot nyújtott konstruktív logikai eszközök kidolgozására, pl. specifikáció nyelv kifejlesztésére.

Ez az egységes programozás-elmélet könyv formájában 1991-ben jelent meg a Springer kiadó gondozásában (l. Gergely-Ury [1991]).

2.4. *A magyar iskola*

Az általunk kidolgozásra került programozási logika a fent említett (iii.)-ik lehetőséget valósítja meg, azaz explicit módon kezeli az adatokat és az időt és a számítógépet úgy modellezi, hogy az „absztrakt gép" egyaránt reprezentálja az adatokat és az időt. Ezt a logikát a hetvenes évek végén dolgozták ki és aktívan vizsgálták elsősorban nem-standard dinamikus logika néven. E logikához kapcsolódó hazai kutatások döntő többsége Andréka, Csirmaz, Gergely, Németi, Sain és Ury nevéhez fűződött, külföldről Hajek, Makowski és Pásztor kapcsolódott be a kutatási munkákba. **Tekintettel arra, hogy e területen az alapvető eredményeket magyar kutatók érték el a programozás-elméletnek ez az irányzata kiérdemelte a világban a "magyar iskola" elnevezést.** Az iskola sok nyitott kérdést oldott meg a programozási logikák területén.

A magyarországi kutatás, amely az elsőrendű logika keretében kívánta a kérdéseket megválaszolni közös platformról indult, de két helyre koncentrálva relatíve gyenge szakmai kapcsolatok mellett folytatódott. Az egyik helyszín az MTA Matematikai Kutató Intézet, azaz akadémiai környezet, míg a másik a Számítógép Alkalmazási Kutató Intézet (SZÁMKI), azaz ipari környezet. Ezzel Magyarországon két központ alakult ki a (nem-standard) programozás-elmélet kutatására. A kutató munkákba más intézmények is bekapcsolódtak, mint pl. az SZKI, NIM IGÜSZI, stb.

A kutatások szempontjából az ipari környezet nehezebb terepnek bizonyult, mivel itt a kutatásokat rentábilisan kellett művelni. Ennek érdekében számos pályázaton vettünk részt, ahol elnyert támogatások biztosították a szükséges pénzügyi feltételeket. Ezzel egyúttal a K+F szkeptikusok irányába is bizonyítani szándékoztunk, akikből szép számmal akadt az intézmény vezetői között. A számítástudomány hazai kutatásának létjogosultságával szembeni szkepticizmus az akadémia körökben is jelen volt.

Megjegyzem, hogy a két központ kétfajta szemantikai konstrukcióval kezdett dolgozni. Az elsőt Andréka Hajnallal és Németi Istvánnal vezettük be 1977-ben folytonos futás szemantika néven (l. Andréka, Gergely, Németi, [1977]), a másodikat Ury Lászlóval 1978-ban, definiálható futás szemantika néven (l. Gergely, Ury, [1978]). Ez utóbbi szemantikai konstrukció használata jellemezte a SZÁMKI-ban folyó kutatásokat, míg a folytonos futások a másik kutatási központ eszköztárához tartoztak.

2.5. *Nemzetközi kapcsolatok*

A magyar iskola közelítésmódjának és matematikai apparátusának megjelenéséig sok különálló közelítést és többféle programozási logikát dolgoztak ki. Ezek nagy része egy-egy feladatra koncentrált és nem adott átfogó programelméletet. Fel sem merült annak igénye, hogy a kapott eredményeket összehasonlítsák. A nyolcvanas évek közepén egy európai pályázat témájának javasoltam az egységes programozás-elméleti platform kidolgozását, amely lehetőséget biztosít a különféle programozási elméletek és programozási logikák összehasonlítására. A projekt javaslat megbeszélésére Londonban került sor. Helyszíne a South Bank Politechnic, szakmai házigazdája Tom Maibaum az Imperial College tanszékvezető professzora volt. A találkozón több különböző európai kutatóközpont kutatója is megjelent. Mi Ury Lászlóval képviseltük a javasolt témát. Itt a tét az volt, hogy az összehasonlítás eredményeként melyik logika lesz a legerősebb a kifejező erő tekintetében, illetve milyen gyakorlati alkalmazás felé mutató módszertant képes megalapozni. A hosszas vita eredménye már elutazásunk után fogalmazódott meg egy elutasítás formájában és

mindenki maradt a saját kutatási irányánál. Így egy olyan európai projektjavaslat született, amely sem a javaslatunkat, sem a mi részvételünket nem igényelte.

A program-, illetve programozás-elmélet területén működő európai és amerikai kutató csoportok munkáját ismertük, vezető munkatársaikkal személyesen is többször találkoztunk. Ezek a találkozások részben konferenciákon, részben a konkrét kutatóhely meglátogatása kapcsán történtek. A találkozások során próbáltuk megismertetni a hallgatóinkat eredményeinkkel és közelítésmódunk lényegével. Ez azonban csak bizonyos mértékig sikerült és ezek a találkozások nem vezettek szorosabb együttműködéshez. Az volt a benyomásunk, hogy a saját közelítésmódjukon kívül nem igen fogadtak el más szemléletet. Pontosabban szívesen látták azokat a felvetéseket és javaslatokat, amelyek a saját paradigmájukon belül felmerült problémákra vonatkoztak, de paradigmaváltásra nem voltak nyitottak.

2.6. A következő szakasz felé.

Az elsőrendű logikára alapozott egységes számítástudomány kifejlesztésének első lépését tehát az a programozás-elmélet adta, amely hatékonyan képes integrálni a dinamikus- és időlogikák kifejező erejét a program konstrukciók megadása érdekében és amely elsőrendű elméletet biztosított a kiszámításelmélet, a programok elmélete és a programozási logikák számára. A második lépés pedig az imperatív programozási paradigma kezelése után a deklaratív programozás paradigma számára megfelelő első-rendű logikai eszközök és elméletek kidolgozása volt.

3. A második szakasz: a programozási paradigmák kezelése – a logikai programozás

3.1. Általános megfontolások.

A számítógéptudományban két egymástól lényegesen eltérő szemléletű programozási paradigma alakult ki az imperatív és a deklaratív programozás. Így a programozás-elméletnek is megfelelő eszközöket és módszereket kellet kialakítania mindkét paradigma kezelésére. Mielőtt belemennék a részletekbe, kitérek néhány programelmélettel kapcsolatos alapfogalomra. A program egy algoritmus formális leírása, amely számítógépen implementálható és végrehajtható. A programozási nyelv az egy formális nyelv, amelynek jól formázott kifejezései a programok. A programozás az egy olyan folyamat, amely programokat hoz létre egy adott input-output specifikációhoz.

Az imperatív programozás utasításokat használ az algoritmusok implementálását szolgáló programok megadására. Az utasítások egy programozási nyelv által vannak megadva. Ezek a nyelvek különféle utasításokat kezelhetnek és az általuk felépíthető programok vonatkozásában különböző kiszámítási erővel rendelkezhetnek. Az előzőekben említett programozási logikák, illetve program elméletek az imperatív programokkal foglakoztak. Az általunk kidolgozott programozás-elmélet egységes elméleti keretet adott a legkülönbözőbb programozási nyelvek utasításkészletével felépíthető programok tulajdonságainak megadására és kiszámítási erejük összehasonlítására.

A deklaratív programozás deklarációkkal operál. Pontosabban egy formula, egy formális állítás írja le az implementálandó algoritmust. Ez a leírás pedig számítógépen implementálható úgy, hogy a leírást a programnyelv értelmezője végrehajtható utasításokra transzformálja. Az egyik legismertebb változata e programozási paradigmának a logikai programozás. A logikai programozás esetében, a deklaráció, azaz a formális leírás az egy logikai formula. A formulának, mint programnak a végrehajtása, azaz futása az egy bizonyos levezetés. A futás tehát itt feltételez egy kalkulust és egy keresési stratégiát. A logikai programozási nyelv az egy bizonyítási eljárás, amely a következő komponensekből áll: (i) a logika szintaxisa, amelyen a formulák kerülnek megfogalmazásra, (ii) egy kalkulus, amely a következtetési szabályok egy adott halmaza, és (iii) egy keresési stratégia, amely egy konkrét folyamatba rendezi a végrehajtható szabályokat a levezetés megvalósítása érdekében.

Fontos megjegyezni, hogy addig ameddig az imperatív programozás szorosan kapcsolódott a kiszámítási modellekhez a deklaratív programozás a matematikai formalizmusokra és elméletekre épült, amelyek a számítógépes technikáktól függetlenül alakultak ki.

3.2. Konceptuális alapok.

A logikai programozás 1973-ban született a PROLOG nyelvvel Colmauer és munkatársai révén megelőzve saját elméleti megalapozását. A PROLOG a Robinson által bevezetett rezoluciós bizonyítási eljárásra épült, pontosabban annak SLD rezolúciónak nevezett verziójára. Ez utóbbi a definit klózókra kifejlesztett SL rezolúciót jelöli. A PROLOG nyelv szintaxisa az un. definite klózókra (nem-negatív Horn klózókra) épített. 1974-ben Kowalski bevezette a „logika, mint programozási nyelv" elvet. Ettől kezdve a PROLOG elindult hódító útjára. A logikai programozást azonosították a PROLOG nyelvvel. A kutatásfejlesztés e nyelv körül folyt, aminek során számos kiterjesztését, illetve dialektusát dolgozták ki. A Kowalski által elindított elméleti kutatások is a PROLOG szerű nyelveket vizsgálták.

Elég hamar felvetődött annak igénye, hogy kiderítsük a logikai programozás általános törvényszerűségeit, és hogy felfedezzük a PROLOG-on túli világot. Maga Kowalski is felvetette két fontos kérdés megválaszolásának szükségességét a továbblépés érdekében: (i) milyen bizonyítási eljárások tekinthetők programozási nyelvnek, (ii) mely programozási nyelvek tekinthetők logikainak. Természetesen ezek nem matematikai precizitással megfogalmazott kérdések voltak. A kérdések megválaszolása során fontos volt szem előtt tartani, hogy a logikai programozási nyelveknek konstruktívaknak kell lenniük valamint, hogy a logikai programokhoz kapcsolódik egy olyan mechanizmus, amely egy univerzális algoritmust realizál és a programot megadó formulát kielégítő értékeket állít elő. Tekintettel arra, hogy mi a klasszikus elsőrendű logikát tekintettük adekvát elméleti környezetnek a fenti kérdések korrekt megfogalmazása a következő: mely elsőrendű formulák tekinthetők programnak.

3.3. Három alapvető kérdés.

Kowalski az algoritmusokat úgy jellemezte, mint logika + ellenőrzés. A logikai programozásban az ellenőrzés a "realizáció”-nak felel meg konstruktív értelemben a logikai részben használt szintaktikai objektumoknak megfelelően. Ezért, a logikai programozást a következőképpen jellemezhetjük: program = kérdés + logika + realizáció, ahol a logika relációkat használ, amelyek vagy az eredeti hasonlósági típus elemei, vagy pozitív egzisztenciális (PE) definícióval vannak definiálva. A pozitivitás azt jelent, hogy a definiálandó reláció jelek a definiáló formulában nem lehetnek benne egy negáció jel hatáskörzetében. Továbbá a definiáló formulában az új reláció jelekben szereplő változó jelek csak egzisztenciális kvantorral lehetnek korlátozva. Ez utóbbi felel meg az egzisztencialitás követelményének. Megjegyzem, hogy a válasz a logikai program kérdésére egy kifejezés formájában áll elő az implicit definíciók halmazából adott axiómákból történő konstruktív bizonyítás eredményeként. A bizonyítási folyamat a logikai program futása, ami tulajdonképpen egy logikai alapú kiszámítási folyamat.

A logikai alapú kiszámítás természetesen megkövetel egy "közeget", amely fölött a kiszámítás megvalósítható. Mivel a logikában a modellek azok az objektumok, ahol egy formula értelmezhető, így ezek felett célszerű a kiszámítást szervezni. Ahhoz azonban, hogy a logikai programok kérdésére a kiszámítási folyamat megvalósulhasson és eredményeként valódi válaszok szülessenek, a modelleknek megfelelőeknek kell lenniük. Ez azt jelenti, hogy konstruktív modellekre van szükségünk. Ezekben a modellekben történik a bizonyítás végrehajtása, vagyis a kiszámítási folyamat.

A logikai programozás logikai alapjainak kidolgozása során először a logikai programozással összefüggő fogalmakat kívántuk tisztázni. Ennek kapcsán célunk volt, hogy (i) meghatározzuk az elsőrendű klasszikus logika azon formuláit, amelyeket logikai programnak tekinthetünk. (ii) jellemezzük a bizonyítás eljárásokat, amelyeket logikai programozási nyelvnek tekinthetünk, és (iii) jellemezzük a kiszámításnak tekinthető bizonyításokat.

A válaszok kidolgozásához szükséges matematikai eszközök egy részét Ury Lászlóval dolgoztuk ki 1978-ban (l. Gergely, Ury, 1978). A logikai programozás matematikai logikai eszköztárat a maga teljességében Szőts Miklóssal együtt hoztuk létre a nyolcvanas évek első felében (Gergely, Szöts, 1984, 1985). A kérdésekre adható válaszok keresése során egy fontos és célszerű előfeltevéssel számoltunk, nevezetesen, hogy a logikai programokat definíció formájában adhatjuk meg. Ennek a feltevésnek a megvalósítása érdekében meghatároztuk és tanulmányoztuk a „kiszámítható definíciók” osztályát. Meghatároztuk a predikátum-kalkulus egy konstruktív részlogikáját. Az induktív definíciók elmélete jellemezte azon definíciókat, amelyeknek létezik olyan legkisebb fix pontja, amelyet megszámlálható sok lépésben megkonstruálhatunk. Megmutattuk, hogy PE

definíciók kielégítik ezt az elvárást. Vizsgáltuk a PE definíciók osztályát és többek között megmutattuk, hogy minden rekurzívan felsorolható reláció megadható, mint egy PE definíció legkisebb fix pontja. Ennek alapján a fent megfogalmazott (i.) kérdésre a válasz a következő tézis volt:

1. tézis: *Tetszőleges logikai programozási nyelv logikai programjainak halmaza ekvivalens a PE definíciók halmazával.*

Tehát ahhoz, hogy egy logikai nyelv programozási nyelvként léphessen fel, szintaxisának PE definíciókból kell állnia. Ennek fontos következménye, hogy a logikai programozási nyelvek közti különbséget nem a programok halmaza hordozza, hanem a kalkulusok közti különbség. A tézisből következik, hogy a logikai programozási nyelvek közti különbséget nem a programok halmazában kell keresni, hanem a megfelelő realizáló kalkulusokban.

Természetesen ez a tézis ekvivalencia erejéig értendő. Itt az ekvivalencia nem jelenti a szokásos szemantikai ekvivalenciát, hanem a legkisebb fixpont szerinti ekvivalenciát. Nevezetesen két formula ekvivalens, mint logikai programok, ha ugyanaz a legkisebb fixpontjuk. A PE-definíciók osztálya univerzális, abban az értelemben, hogy minden kiszámítható függvényt meg lehet adni PE-definíció legkisebb fixpontjaként.

A logikai programozás általunk kifejlesztett paradigmájában PE formulákat kell bizonyítani PE definíciókból. Ehhez kapcsolódóan megvizsgáltuk, hogy a különböző kalkulusok kezelési tere hogyan fajul el, ha erre a speciális esetre alkalmazzuk. Ezzel tovább léptünk programozási szemszögből is, elvégeztük a lehetséges interpreterek és fordítók általános elvi alapjainak feltárását és ezek osztályozását . A vizsgálatot a természetes levezetés rendszerére végeztük el, de megmutattuk, hogy eredményeink érvényesek a Gentzen-féle kalkulusra, az analitikus táblázatokra és a rezolúciós kalkulusra is. Ennek alapján joggal mondhatjuk általánosnak a következő pontokban felsorolt főbb eredményeket:

(i) A „kiszámítási tér” minden esetben és/vagy fává fajul, amely különböző esetekben vagy izomorf a természetes levezetéssel kapott keresési fával, vagy ebből csomópontok (levezetési szabályok) összevonásával kapható. Ez az eredmény megmagyarázza, miért pont a választott paradigma (PE-formula, mint logikai program) ad hatékony bizonyítási eljárásokat, s úgy látszik, hogy tetszőleges kalkulusra.

(ii) Az és/vagy fa összeállítható a bizonyítandó formulák és a definiáló formulák elemzési fájából. Tehát esetünkben a formula joggal tekinthető programnak, alakja meghatározza a bizonyítás menetét. Azaz a PE definíciókhoz minden kalkulus esetén van áttekinthető operációs szemantika.

(iii) A különböző logikai programozási nyelvek bizonyítás-elméleti szempontból lényegesen a következő összetevőkben különbözhetnek:

- megkövetel-e normál alakot,
- milyen szabállyal kezeli a kvantált változókat,
- használ-e korlátos kvantort,
- használja-e a helyettesítési szabályt.

Az, hogy a fenti összetevőkből mit érdemes választani, attól függ, hogy milyen a modell, amelyben a logikai programokat értelmezzük. A kérdéses modell megszerkesztését pedig, ahol a kérdést meg

kell válaszolni a logikai programok fix-pont egyenletek formájában megadott implicit PE definíciói biztosítják.

Felmerült a kérdés, hogy milyen kalkulusok tudnak konstruktív bizonyítást adni nem PE formulákhoz. Ilyen kalkulus felépítése egyáltalán nem egyszerű. Az világos, hogy nincs univerzális algoritmus a nem-PE formulák kezelésére. Mi ennek kapcsán néhány konstruktívan bizonyítható nem-PE-formula osztályt határoztunk meg. Ezek közül a legfontosabb probléma a negáció, azaz tagadás kezelése a konstruktív realizálhatóság megőrzése mellett. Mi megmutattuk, hogy a tagadás korlátozott használata nem teszi tönkre a PE definíciók fontos tulajdonságait. Ennek a kérdésnek a vizsgálatához bevezettük a kiegészítő, vagy komplementer definíció fogalmát, amely alapvető eszköze volt a negáció kezelésének.

Megjegyzendő, hogy az összes eredményünk érvényes maradt, ha korlátos kvantorokat használtunk. A korlátos univerzális kvantor bevezetése valóban kiterjeszti a szigorúan egzisztenciális formulákat. Ez pl. az a szintaktikai tulajdonság, amellyel a mi általunk meghatározott PE-nyelv rendelkezik és ami hiányzik a PROLOG-szerű nyelvekből.

A legfontosabb következménye az 1. tézisnek az, hogy minden logikai programozás nyelv alapvetően azonos program osztállyal rendelkezik. Ez nem jelenti azt, hogy nem léteznek egymástól ténylegesen különböző logikai programozási nyelvek. Megadtuk a LOBO logikai programozási nyelv alapjait, amely nagyon fontos tulajdonságokban különbözik a PROLOG-szerű nyelvektől. Azonban szinte minden eltérő tulajdonság, még a szintaxisban rejlő különbségeket is a kalkulusok különbözősége okozza. A kidolgozott LOBO egy fontos argumentum volt a következő tézis alátámasztására.

2. tézis: *Különböző kalkulusok jelentősen különböző logikai programozási nyelveket adhatnak meg.*

Az általunk kidolgozott PE definíciók elméletére építve specifikáltuk és kifejlesztettük a PROLOG nyelvtől lényegesen eltérő LOBO logikai programozási nyelvet.

3.4. Nyitás a világ felé.

A logikai programozás megalapozása és egységes elméleti alapjainak kidolgozása során kapott eredményeket 1985 augusztusában szervezett logikai programozással foglalkozó nemzetközi nyári iskolán mutattam be Turkuban (Finnország). A nyári iskolának különleges rangot adott, hogy hárman tartottuk Robert Kowalski-val és J. Robinson-nal. Emlékeztetőül: Kowalski a PROLOG nyelv világsikerének megalapozója, Robinson pedig a logikai programozás bizonyításelméleti alapjait jelentő rezolúció módszer kitalálója volt. Tehát itt Robinson képviselte a konkrét módszertani alapokat, az alkalmazást Kowalski, míg a jövő irányába mutató egységes logikai megalapozást és fejlesztési módszertani ajánlásokat én képviseltem.

3.5. A következő szakasz felé.

A logikai programozás elméleti alapjainak a klasszikus elsőrendű logika keretein belül történő felépítése után felmerült az igény a deklaratív programozás egészének elméleti megalapozására, amely például kezelni tudja a logikai programozás mellett a funkcionális programozást is. Sőt lehetőséget biztosít a különböző programozási paradigmák összehasonlítására, vagy akár a logikai programozás funkcionális programozással való kiterjesztésére. Ennek a kiterjesztésnek a megvalósítása a harmadik lépés során realizálódott.

Egyúttal felmerült egy másik fontos kérdés a logika programozás egységes logikai alapjainak kidolgozása során. Nevezetesen, mint láttuk egy logikai program kérdésére a választ kifejezés formájában keressük. Mi van akkor, ha a választ továbbra is kifejezés formájában keressük, de ez a kifejezés egy rekurzív függvény, amely önmagában egy programot ad meg. Ez pedig a program szintézisnek felel meg. Ennek a kérdésnek a megválaszolása a negyedik szakasz során valósult meg.

4. A harmadik szakasz: a programozási paradigmák kezelése – a deklaratív programozás

4.1. Általános megfontolások.

A deklaratív programozás fontos területe volt az új lehetőségeket és a következő generációs számítógépes rendszerek új irányait kereső kutatás számára. A deklaratív programozási paradigma két legfontosabb változata a logikai vagy predikatív programozás és a funkcionális programozás. A logikai programozásnak sok közös gyökere van a funkcionális programozással. Ilyen pl. a logikai természetű formula, kifejezés és érték kezelés, a rekurzióra épülő program modularitás, valamint az induktív definíciók fontos szerepe. Természetesen a különbségek is lényegesek, mint pl. a változók radikálisan különböző kezelése, a funkcionális programozásban megengedett magasabb rendű program-entitások és a logikai programozás nem-determinisztikus program végrehajtása.

A logikai programozás által használt relációs jelölés és tételbizonyítás nagyon elegáns és hatékony képességek, amikkel a funkcionális programozás nem rendelkezik. Mivel egy logikai program nem rendelkezik arról, hogy egy reláció mely változói tekinthetők inputnak és melyek outputnak, egy definiált reláció többféleképpen használható. A funkcionális programok explicit módon rendelkeznek egy funkció bemeneteiről és kimeneteiről, azaz ezek merevek az input-output irányítottság tekintetében. Így egy logikai program több funkcionális programnak felelhet meg, amelyek ugyanazt a deklaratív információt tartalmazzák, de különböző az input-output irányítottságuk.

Az, hogy a logikai programok nem irányítottak, lehetővé teszi, hogy a program írásánál ne specifikáljunk semmilyen vezérlő információt. A vezérlő információ hiánya az adott cél elérésére irányuló kiszámítások széles változatosságához vezet. Azaz a logikai programozás eredendően nem-determinisztikus és egy lekérdezés több megoldáshoz vezethet. Ez az oka annak, hogy a logikai programok magukba foglalnak egy keresési folyamatot. A program futása során bármely kiértékelési pontban egy részcél bizonyítására több különböző út kínálkozik, melyek közül néhány sikeres lehet.

A funkcionális programok determinisztikusak, ami azzal a követelménnyel függ össze, hogy a megadható alakok csak több-egy függvényt jelölhetnek. Vagyis tetszőleges bemeneti értékhez csak egy kimeneti érték kerül kiszámításra. Továbbá megmutatható, hogy bizonyos Church-Rosser tulajdonságok fennállnak a funkcionális nyelvekre és így nincs szükség keresésre amikor egy funkcionális programot végrehajtunk. Ez azt is jelenti, hogy a funkcionális programok meghatározzák a szükséges mennyiségű vezérlési információt. Mindaddig, amíg a bal oldali legkülső redukálható részkifejezés benne marad az egyes lépések során átírt vagy újraírt kifejezések halmazában a válasz garantáltan elérhető, ha létezik.

A funkcionális nyelvek úgy vannak korlátozva, hogy outputként csak konstansok és konstruktor függvények jelenjenek meg. Így e korlátozás miatt kevésbé kifejezők, mint a logika nyelvek.

A funkcionális és logikai programozás előnyeinek egyesítése érdekében különféle programozási nyelv került kidolgozásra. Ezek vagy egyetlen működési keretben biztosítják mind a funkcionális mind a logikai programozást, pl. úgy, hogy interpretálják a függvényeket a logikai keretben vagy fordítva, avagy úgy, hogy kombinálnak egy hagyományos funkcionális nyelvet és egy hagyományos logikai programozási nyelvet egy megfelelő interface-t biztosítva a kettő között. Szinte minden ilyen kombináció szintaktikai szinten valósult meg, és nem nagyon volt olyan, amely mély elméleti

és szemantikai vizsgálatra épült volna. Mi pedig pont olyan elméleti alapokat kívántunk létrehozni, amely megalapozott integrálást tudott biztosítani.

A deklaratív programozás egységes elméletének felépítéséhez az öröklődően véges halmazok elméletét használtuk fel, amelyet az előző két szakasz megtételéhez szükséges kutatásokkal és fejlesztésekkel párhuzamosan Ury Lászlóval dolgoztunk ki. Erre az elméletre építve dolgoztuk ki a deklaratív programozás modellelméleti és bizonyításelméleti megalapozását. Ez az egységes megalapozás lehetővé tette a logikai programozás funkcionális programozássá való kiterjesztését, illetve e két deklaratív mód integrálását.

4.2. Az atomos öröklődően véges halmazok axiómarendszere- a negyedik szakasz

Az atomos öröklődően véges halmazok axiómatizálására kidolgoztuk az FSA axiómarendszert. Miért volt szükség atomokra? Mint tudjuk a Zermalo-Fraenkel axiómarendszer (ZFC) meglehetősen erős és ezért nincs szüksége atomokra. A ZFC-vel formalizált halmazelmélet egy elegáns utat biztosít a matematika megalapozására, de túlságosan erős a számítástudományhoz, pl. a deklaratív programozás elméletének kialakításához. A cél a számítástudományhoz egy megfelelő, a ZFC-nél gyengébb axiómarendszer kialakítása volt. Gyengébb a halmaz létezésének elveiben, amelyek elősegítik a formalizálást, pl. az atomok segítségével.

Az atomoknak elemi adatként van egy programozási értelmezése is, ha a reláció struktúrákra úgy tekintünk, mint a számítógép modelljeire, ahol a programok futnak. Az atomok reprezentálták a memória elemeket, regisztereket, ahol az adatokat tároljuk. Az atomok szintén elrontják a végességet, amit axiómatizálni szándékoztunk, mivel végtelenül sok atom létezhet. Az atomos öröklődően véges halmazokat axiómatizáltuk az atomos ZF axiómarendszer alapján úgy, hogy elhagytuk a hatványhalmazt és a végtelenségi axiómákat és hozzáadtunk egy új axiómát, amely leírja az öröklődően véges halmazokat. Az így kialakult új axiómarendszert FSA-nak neveztük.

A deklaratív programozás elméletének szüksége volt többek között eszközökre a végtelen objektumok kezelésére, pl. a végtelen kiszámítási folyamatok reprezentálására. Ezért a véges halmazokon, mint véges objektumokon túlmenően tudnunk kellett beszélni végtelen objektumokról is. Ennek kezelésére vezettük be az osztály fogalmát. Intuitíve, egy osztály az egy adott formulát kielégítő elemek összessége. Mivel az FSA minden egyes modelljének elemei konstruktív objektumok (atomok vagy véges halmazok), ezért az osztályokat ilyen objektumok definiált, vagy specifikált összességeként adtuk meg.

Egy adott modell esetében egy osztály nem tartalmazta az univerzum (alaphalmaz) tetszőlegesen összegyűjtött elemeit, hanem egy adott formula segítségével összeszedett elemekből állt. Így a nyelvet kibővítettük úgy hogy alkalmas legyen osztályokról szóló állítások megfogalmazására. Ennek érdekében bevezettük az osztályváltozókat. Ezek elvárt jelentése FSA egy modelljében az univerzum objektumainak egy összessége volt. Az osztály fogalmával kibővített axiómarendszert cFSA-nak neveztük. Megjegyzem, hogy nem kívántuk meg az osztályváltozók feletti kvantálást, mivel ezeket a változókat mi csak definíciós eszközként használtuk, amelyek segítségével pontosan meg tudtuk fogalmazni az egyes, az elmélet szempontjából fontos fogalmakat.

Mint azt az előző fejezetben láttuk, a logikai programozás esetében a logikai összetevő csak relációkat használt, amelyek vagy az eredeti hasonlósági típus elemei, vagy pozitív egzisztenciális definícióval vannak definiálva. Ugyanakkor a negáció kezelése fontos kérdés volt a deklaratív programozásban és így a logikai programozásban is. Ennek egyik lehetséges megoldását az intuicionista logika jelentette. Ez a logika lehetővé teszi a tagadás egy természetes kezelését. Az intuicionista logika igazi jelentősége azonban a programok szintézisében volt.

Egy logikai program futásának eredménye egy logikai kifejezés formájában megadott válasz a program által feltett kérdésre. Ez a válasz az implicit definíciók halmazából áll elő, az axiómákból történő konstruktív bizonyítás eredményeként. A logika programozás egy másik lehetséges útja amikor a választ szintén kifejezés formájában keressük, de ez a kifejezés egy rekurzív függvény, amely egy programot ad meg. Tehát ez az út a program-szintézisnek felel meg. Ennek, valamint a tagadás kezelésének megvalósításához kidolgoztuk az öröklődően véges halmazok elméletének intuicionista változatát, az IcFSA axiómarendszert. Természetesen itt a kalkulus, amelyet a kérdések bizonyítására szolgál szintén intuicionista volt.

Kidolgoztuk az absztrakt adattípusok megadására alkalmas logikai eljárást, valamint az adatmodellezésnek és adatbázis kezelésnek egy a deklaratív programozást megalapozó elméletünkhöz adekvát módszertanát.

A funkcionális programozás definíciójának különböző módjait dolgoztuk ki. Először a függvények definíciójából egy E ekvivalencia relációt adtunk meg, amelynek segítségével a konstruktív modellt faktorizálni kell azon célból, hogy lehetővé váljon a megfelelő függvények felépítése. Másodszor, a függvények definiálásának egy univerzális módját választottuk, amelynek segítségével az összes lehetséges függvény definiált és ezek közül választjuk ki a definíció által igényelt függvényeket kifejezés formájában. A lambda kalkulus és a típus-elmélet két különböző lehetőséget biztosított a funkcionális programozás második útjának megvalósításához. Mindkettőnek kidolgoztuk az elméletét a cFSA axiómarendszerre alapozva. Az általunk kidolgozott típusos nyelv tartalmazott un. öröklődően pozitív egzisztenciálisan definiált típusokat, amik analógjai voltak a típus-mentes eset pozitív egzisztenciálisan definiált objektumainak. Ennek szemantikája az FSA modelljeire épült, amelyeket kiterjesztettünk megfelelő extra struktúrákkal. Az így kidolgozott típus-elmélet konstruktív volt és így egy hatékony alternatívát jelentett Martin-Löf által kidolgozott elméletnek.

A funkcionális programozás megvalósításának említett útjai kombinálhatók a relációs (logikai) programozással. A funkcionális és logikai programozások kombinációjának egy másik útját is kidolgoztuk, amely az előbb említett intuicionista logikára épült.

4.3. A kutatás feltételei

A deklaratív programozás egységes elméleti alapjainak és módszertanának kidolgozása az OMFB-vel kötött Gl-42-044/86. sz. szerződés keretében kapott támogatást. Az általunk elért eredmények döntő többsége csak kutatási jelentések, illetve belső kiadványok formájában jelent meg. Ezek listáját az irodalomjegyzék tartalmazza.

4.4. További lépések

A deklaratív programozás sokkal inkább program-specifikáció, mint hagyományos értelemben vett programozás. Ennek megfelelően a kidolgozott halmazelmélet megfelelő megalapozást adott a számítástudomány más ágainak is, így például a programspecifikáció és program szintézis számára is. Ennek bemutatása a következő fejezetben található.

5. A specifikáció elmélet – az ötödik szakasz

5.1. Konceptuális alapok

A program, illetve a programozás elméletek számára különféle formális eszközöket fejlesztettek ki arra, hogy leírhassák, mit csinál egy program és azt hogyan teszi. Ez egy új lehetőséget is megnyitott nevezetesen, hogy előírjuk, hogy mit és hogyan kell csinálnia egy programnak. Ezen az úton az első eredmények az algebrai keretek között születtek. Ugyanerre a célra az algebrai közelítésen túlmenően egy másik közelítés is született, amely az algebrai közelítés explicit állapot fogalmával szemben a kiszámítási állapot fogalmát impliciten hagyta. Megjegyzem, hogy minden egyes közelítés először egy specifikáció elméletet kell, hogy megadjon, amely a programokkal kapcsolatos fogalmakat és tulajdonságokat képes kezelni majd erre az elméletre építve kifejleszthető egy megfelelő specifikáció nyelvet.

Mi az előzőektől lényegesen eltérő közelítésmódot dolgoztunk ki. Párhuzamosan két úton indultunk el. Az első út egy nagyon absztrakt közelítésmód kialakítását és egy erre épülő absztrakt specifikáció elmélet kifejlesztését célozta meg. Ettől az elmélettől elvártuk többek között, hogy megmutassa egy input nyelvvel szemben elvárt követelményeket. A második út egy újfajta közelítésre épülő, a gyakorlati programozást segítő szoftverfejlesztési módszertan és az ezt támogató szoftverrendszer kidolgozását tűzte ki célul

A mi legfőbb elvárásaink a specifikáció elmélettel szemben a következők voltak:

(i.) A finomíthatóság, azaz a specifikáció folyamat a különböző absztrakciós szinten megadott specifikációk egy sorozatát állítja elő, ahol az egyes specifikációk egymásból lépésenkénti finomítással állnak elő**.** Más szavakkal ez az jelenti, hogy az egyes specifikációk további részletek megadásával kiterjeszthetők legyenek.

(ii.) a modularitás, azaz, hogy a specifikáció egymástól jól elkülönülő egységekből legyen felépíthető, valamint

(iii.) az integrálhatóság, azaz, hogy a specifikációk összekapcsolhatók legyenek, vagyis egy új specifikáció legyen kialakítható több specifikáció megfelelő összekapcsolásával.

A fenti elvárásokon túlmenően szükség volt egy egyértelmű nyelvre, amelyen a specifikációk megfogalmazhatóak. A specifikáció nyelvvel szembeni legfontosabb elvárása, hogy alkalmas legyen a programokhoz és a programozáshoz kapcsolódó minden fontos fogalom megfogalmazására. Annak érdekében, hogy jellemezhessük a specifikációkat és a specifikációk formális leírását támogató nyelveket, egy magas absztrakciós szintű specifikáció elméletre volt szükség. Ezt az elméletet a kategória elmélet eszközeivel adtuk meg. Mivel a specifikáció egy leírás, ezért a specifikációk pragmatikai szempontból adekvát jellemzésére egy, a logikai formulákra és elméletekre épülő speciális kategóriát, az un. logikai kategóriát vezettük be. Ezen belül formális eszközökkel tudtuk jellemezni a fenti elvárásoknak eleget tevő specifikáció elméleteket. Egyúttal megadtuk egy specifikáció-nyelvvel szembeni elvárásokat is.

A programtervezési folyamat során, amely a felhasználó követelményeivel kezdődik és a futó programmal végződik több formális specifikáció adható meg különböző absztrakciós szinten.

Az általános specifikáció elmélet elvárja a nyelvtől, hogy az támogassa a lépésenkénti finomítási elvet. Ehhez pedig arra van szükség, hogy teljesüljön a Craig interpolációs tulajdonság. Ez azt mondja ki, hogy ha egy nyelv egy ϕ_1 állításából következik egy ϕ_2 állítás, akkor a nyelvben találhatunk egy olyan harmadik ψ állítást, hogy ϕ_1-ből következik ψ és ψ-ből következik ϕ_2. E mellett ugyanakkor célszerű megkövetelni, hogy a nyelv tegye lehetővé a programokhoz kapcsolódó összes szükséges fogalom, jelenség és tulajdonság definiálását.

Fontos kérdés még, hogy a specifikáció nyelv hogyan kezeli a definiált fogalmakat szemantikai szempontból. Nevezetesen elvárjuk, hogy a definíciók belsők legyenek, azaz az elmélet a fogalmakhoz és entitásokhoz a modell létező belső objektumait feleltesse meg. Természetesen ehhez a modelleknek is speciálisnak kell lenniük, amelyeket az elsőrendű modellek felett generálhatunk. Ezek az un. szuperstruktúrák, amelyek a program futásának megfelelően épülnek fel. Ahhoz hogy elméletünkben ezeket a szuperstruktúrákat használhassuk egy megfelelő formalizmust vezettünk be, amely lehetővé tette egy elmélet (axióma-rendszer) megadását, amelynek modelljei a kérdéses struktúrák. Ezen struktúrákról állítások fogalmazhatók meg és bizonyíthatók. Céljainknak megfelelően kidolgoztuk az elsőrendű specifikáció elméletet ahol a szuperstruktúrák axiómatizálását egy megfelelő elsőrendű nyelv segítségével adtuk meg.

Az axiómatizálást halmazelméleti közelítést követve adtuk meg. A megfelelő axiómarendszert az előző fejezetben említett cFSA axiómarendszer biztosította, amely az atomokkal és osztályokkal rendelkező véges halmazok elmélete. Ennek fontos tulajdonsága, hogy véges sok axiómával megadható. Ennek az axióma rendszernek a modelljei a fennemlített szuperstruktúrák.

Annak érdekében, hogy a bevezetett elméleti keret használható és kényelmes legyen, szükség volt egy hatékony módszerre ahhoz, hogy a programozási fogalmaknak megfelelő formális objektumokat belsővé tudjuk tenni. Az elméletünkben az osztályok voltak ezek az elvárt objektumok. Az egyik lehetséges útja ezen objektumok belsővé tételére a definícióelmélet volt. Megszokott módszer a programozási fogalmaknak megfelelő formális (matematikai) objektumokat egy megfelelő fix-pont egyenlet megoldásaként definiálni. Ezért, mi is az implicit definíciókat biztosító fix-pont egyenleteket választottuk. Ugyanakkor, azon célból, hogy a programozási fogalmakhoz belső formális objektumaink legyenek, szükségünk volt ezek explicit definíciójára a mi halmazelméleti keretünkben, azaz cFSA-ban. Ezért, egy olyan fix-pont elméletre volt szükségünk, amely FSA-ban definiálható megoldásokat biztosít. Ezért a szokásos legkisebb fix pontok helyett minket a definiálható legkisebb fix pontok érdekeltek. Ennek biztosítása érdekében a pozitív egzisztenciális (vagy konstruktív) függvényeket választottuk, amelyek felett a számunkra szükséges fix-pont elmélet kidolgozható volt. Megjegyzem, hogy ezeknek a függvényeknek az osztálya éppen a kiszámítható függvényekből állt.

A specifikáció elmélettől elvártuk, hogy alkalmas nyelvet tudjon megadni a programok specifikációjához. Ezt a nyelvet mi a Z specifikáció nyelv mintájára definiáltuk. A Z nyelv egyike volt a matematikai alapokkal rendelkező specifikáció nyelveknek. Nevezetesen ennek a nyelvnek a megalapozás a Zermalo-Fraenkel axiomatikus halmazelmélet jelentette. Maga a nyelv eszköztára egyaránt használt algebrai és logikai eszközöket. Alapvetően a Z egy, a halmazok konstruálására épülő specifikáció nyelv. Ugyanakkor a legnagyobb problémája a Z-nek, hogy nem konstruktív, ami pl. nehézségeket szül a rekurzív halmaz definíciók terén. Továbbá a halmazelméleti alap implicit maradt a Z-ben megadott specifikációk mögött, amitől ezek kétértelműek lettek. Továbbá Z-nek nem volt használható formális szemantikája, ami támogatni tudta volna a specifikáció folyamatait. Mi kidolgoztuk a HF specifikáció nyelvet, amely mentes volt a Z minden előbb említett hiányosságaitól. Ennek a nyelvnek a matematikai alapjait a cFSA adta meg.

5.2. További lépések

A specifikáció elmélettől elvártuk, hogy alkalmas nyelvet tudjon megadni minden szükséges fogalom és entitás definíciójára így az elmélet megfelelő megalapozását adja egy programozás-elméletnek, pontosabban egy program-elméletnek. Ennek a programozás-elméletnek a kialakítása teljesíti ki az egységes halmazelmélet alapra épülő számítástudomány létrehozását.

Mint fent említettem a specifikáció elmélet fejlesztésének második útja egy újfajta közelítésre épülő, a gyakorlati programozást segítő szoftverfejlesztési módszertan és az ezt támogató szoftverrendszer kidolgozását tűzte ki célul. Ezen a területen végzett munkákról szól a következő fejezet.

6. A hatodik szakasz: szoftvertechnológia

6.1. Konceptuális alapok

Az általunk kifejlesztett egységes számítástudomány lehetőséget biztosított arra is, hogy a gyakorlatban is használható szoftverfejlesztési módszertant lehessen kidolgozni Ezen belül is minket a nagyméretű szoftver rendszerek létrehozásának támogatása érdekelt. Az ezzel kapcsolatos kutató- és tényleges munka dandárját Halmay Edit végezte.

A gyakorlatorientált programozási módszertan és szoftver mérnökség előtt abban az időben a nagyméretű szoftver rendszerek létrehozásának és karbantartásának technológiai támogatottsága távolról sem volt kielégítőn megoldva. A szoftvermérnöki gyakorlatnak még mindig számos nyílt, technológiailag és módszertanilag lefedetlen problémája volt. Tipikusan ilyen problémák a tervezési döntések konformitásának bizonyítása nagyméretű szoftverek esetén, az újrafelhasználási probléma, a módosítási/karbantartási probléma, a szoftverek szövege és specifikációja közötti konzisztencia biztosítása a tárgyszoftver teljes életciklusa alatt, stb. A módszertani és technológiai támogatottság hiánya különösen szembeötlő a folyamatosan bővülő ill. változó (ún. evolúciós) rendszerek esetén. Ezek a rendszerek speciális, a *teljes életciklust lefedő* támogatást igényelnének.

A szoftverek szövege és specifikációja közötti konzisztencia biztosítása hívta életre az ún. reverse engineering eszközöket. Ezek hatóköre azonban a programok szintjére korlátozódott. A több száz, esetenként több ezer programból felépülő szoftverrendszerekben való "tájékozódás" végett nagy szükség volt a *rendszer architektúrális és funkcionális kontrolljának* és az *absztrakciós lehetőség* támogatására is, bár ilyen szoftvertechnológiai eszköz akkor még nem létezett.

A szoftvermérnöki gyakorlat egy-egy *konkrét* szoftver rendszer tervezése során beszélt ugyan a rendszer architektúrájáról, a rendszer összetevőiről (alrendszereiről), ill. ezek integrációjáról, a programok rendszerré szerveződésének *általános törvényszerűségeiről* azonban szinte semmit nem tudtunk.

Mi akkor azt vallottuk, hogy érdemi fejlődés a szoftvertechnológiában csupán egy olyan *univerzális szoftver elmélet*től várható, amely megbízható, általános érvényű, ugyanakkor a gyakorlatban is hasznosítható tudást közvetít a szoftverrendszerek világában uralkodó törvényszerűségeket illetően.

Ezért egy ilyen szoftver elméletre épülő szoftvertechnológia kidolgozását tűztük ki célul a következő főbb jellemzőkkel:

(i) Egységes formális matematikai elméleti alapokkal rendelkezik, amely

- egységes konstruktív matematikai logikai alapokra épül, amely hatékony definíció elmélettel rendelkezik tetszőleges tulajdonság, illetve konstrukció megadására,
- általános specifikáció (modell) elméletet nyújt, amely megalapozza mind a lépésenkénti finomítás mind a lépésenkénti absztrakció módszerét,
- egy új szemantika konstrukciót az un. funkcionális szemantikát használja, amely alkalmas szoftver rendszerek jelentésének megadására is,
- rendelkezik a programrendszerek formális elméletével, amely a funkcionális szemantikára épül és alkalmas tetszőleges szoftver rendszer leírására és vizsgálatára.

(ii) Konzisztens módszertannal rendelkezik, amely biztosítja a szoftvermérnökség legégetőbb feladatainak megoldását, mint amilyenek az interfész probléma, a szoftver elemek újrahasznosítása és a program módosítás és támogatás. Két fő összetevője:

- Egy hatékony reverse engineering algoritmus, amely automatikusan képes generálni egy programrendszernek és összetevőinek jelentését. A jelentés egyaránt foglalkozik architektúra, adat és funkcionális modellekkel illetve specifikációval.
- A modellezés, illetve specifikáció lépésenkénti finomítás és lépésenkénti absztrakció módszertana, amely hatékony formális eszközökkel és mérnöki elvekkel rendelkezik.

(iii) Hatékony szoftver fejlesztő környezet formájában kerül megvalósításra, amely az előre-hátra (*forth-and back)* modellre és módszertanra épül, és a teljes életciklus alatt támogatja a szoftver rendszerek fejlesztését és módosítását. Olyan modellezési technológiát biztosit amely lehetővé teszi, hogy kombináljuk a formalizmus szigorúságát a különböző absztrakciós szintű modellezési lehetőséggel, valamint az elvárások természetes nyelvű megfogalmazásaival, és mindezt vizualizáljuk egy megfelelő grafikus nyelvvel. Ez lehetőséget biztosit, hogy már a fejlesztés igen korai szakaszában hatékony ellenőrzési módszereket használhassunk fel.

6.2. A funkcionális szemantika

A kidolgozásra került szoftvertechnológia elméleti hátterének legfontosabb komponense az ú.n. *funkcionális szemantika.* (A programnyelvi szemantika formalizálása a szoftvertechnológiai eszközök számítógépes támogatása miatt elengedhetetlen.)

A funkcionális szemantika alapvetően különbözik a klasszikus programnyelvi szemantikai megközelítéstől, amelyre a számítógéppel támogatott szoftvertechnológiai eszközök támaszkodtak. A klasszikus programnyelvi szemantikai megközelítés szerint egy program szövege arról informál, hogy a program maximálisan milyen leképzéseket tud megvalósítani azon a tárterületen, amelyet a programszövegben előforduló változók szimbolikusan kijelölnek. Ebben a képben egy adott program belső változói ekvivalensek azokkal a változókkal, amelyeken keresztül a program adatokat tud fogadni az őt tartalmazó programrendszertől és/vagy a külvilágból, illetve amelyeken keresztül a program adatokat képes küldeni az őt tartalmazó programrendszernek és/vagy a külvilágnak. Ez a fajta megközelítés egyáltalán nem foglalkozik a szoftver rendszerek programnál nagyobb konfigurációs elemeinek (alrendszereinek) illetve a rendszer egészének szemantikai kezelésével.

A funkcionális szemantikai megközelítés értelmében egy-egy program szövege azt írja le, hogy a program

(1) maximum hányfajta és milyen funkciók ellátásával képes hozzájárulni az őt tartalmazó szoftver rendszerek bármelyikének működéséhez, és, hogy

(2) ezek a funkciók milyen (a programot tartalmazó rendszerből és/vagy a külvilágból fogadott) input-változó n-esekből milyen (a programot tartalmazó rendszerbe és/vagy a külvilágba küldött) output-változó m-esekbe való leképzéseken keresztül fejtik ki hatásukat.

A programoknak ezt a sajátosságát *funkcionalitás*nak neveztük.

Tekintettel arra, hogy egy adott program minden egyes aktiválásakor a program által realizált funkciók közül egy és csakis egy kerül (adott input értékekkel) kiértékelésre, a funkcionális

megközelítés képes a szoftver rendszerek programnál nagyobb konfigurációs elemeinek illetve a rendszer egészének szemantikai kezelésére.

6.3. A forth-and-back szoftverfejlesztési paradigma főbb jellemzői

A funkcionális szemantika bázisán egy új szoftverfejlesztési módszer kifejlesztését céloztuk meg, ami az ismert rendszerfejlesztési módszerekhez (pl. az akkor az USA-ban alkalmazott Yourdon-Constantine vagy a Nagy-Britanniában kifejlesztett SSADM) hasonlóan szoftverrendszerek szigorú szoftvermérnöki alapelveken nyugvó kifejlesztését célozza. Eltérően ezektől, a funkcionális szemantikai bázison kifejlesztett módszer:

- konkrét eszközöket nyújtott a módszer szerves részét képező 5 szoftvermérnöki alapelv (a világos fogalomalkotás elve, az invariancia elv, az integrálhatósági elv, az egyensúly elv és a respecifikáció elve) betartásának ellenőrzésére,
- a PE definíciókkal kifejezett, és az automatikus feldolgozásra alkalmas specifikációk grafikus reprezentációját és megjelenítését nyújtott,
- életciklus támogatást nyújtott a szoftverek számára, és
- megfelelő technológiai alapot adott egy számítógéppel támogatott, integrált szoftverfejlesztési környezet kifejlesztéséhez.

A módszer a szoftverfejlesztési folyamat egészét egyfajta specifikáció folyamatként közelítette meg, amelynek során egy amorf (menetközben pontosított és esetleg változó) követelményhalmaz fokozatosan leképződik egy formálisan definiált funkcionális specifikációba. A tárgyszoftver funkcióinak formális specifikációja tehát *nem előfeltétele, hanem végeredménye* a szoftverfejlesztési folyamatnak.
A módszer a fejlesztés alatt álló szoftverrendszert különböző absztrakciós szintű rendszer-modellek készítésével támogatta, A modellezés a tervezés kezdetén magas absztrakciós szinten támogatta az elvárások megfogalmazását, ami a rendszer specifikációját biztosította. Alacsonyabb absztrakciós szinten a modellek formális programtervet jelenítettek meg, a kód szinten pedig már az implementációt.

A legabsztraktabb leírást a tárgyszoftver konceptuális modellje nyújtotta. A konceptuális modell a tárgyszoftvert egy több funkciós "fekete doboz"-ként ábrázolta. A szoftver minden egyes funkcióját a funkció neve és lokalitása (mely input n-esekből mely output m-esekbe történik a leképzés) szimbolizálta. Ha ennél részletesebb ábrázolásra van szükség, akkor a konceptuális modellből elérhetők voltak a tárgyszoftver architektúrális és funkcionális modelljei. A legmagasabb absztrakciós szinten folyó modellezés támogatására fél-formális nyelvet adtunk meg, amely olyan mértékben formalizált modell megadására alkalmas, ami a konkretizálás irányába való továbblépéshez elengedhetetlenül szükséges.

Megkülönböztettünk funkcionális és nem funkcionális típusú modelleket. A legfontosabb nem funkcionális modellek a különböző részletezettségű architektúrális modellek, amelyek a tárgyszoftver struktúrális sajátosságáról informálnak. A funkcionális modellek a tárgyszoftver egy-egy funkcióját írták le. Egy adott funkció legfelső szintű modellje a szóban forgó funkció gráfjának legabsztraktabb definícióját nyújtotta: azt specifikálta, hogy a kérdéses funkció a tárgyszoftver legfelső szintű komponenseinek funkciói közül melyekből és hogyan épül fel.

A módszertan alapja a különböző modellek közti kapcsolatok kezelése, amely megvalósítja a lépésenkénti finomítást, és a lépésenkénti absztrakciót.

A lépésenkénti finomítás során az általános modellektől a konkrétabb modellek felé haladtunk, Ez elsősorban a tervezés eszköze volt. A lépésenkénti absztrakció során pedig a konkrétabb modellektől haladtunk az absztraktabb felé, pl. egy adott programból kiindulva megadtuk annak specifikációját. Ez utóbbi lehetett rendszer architektúra, vagy pl. funkcionális modell előállítása is.

A módszer szerves része volt az egyes tervezési döntések elfogadhatóságának bizonyítása. (Egy adott tervezési döntés akkor tekinthető elfogadhatónak, ha nem sérti a módszer által megfogalmazott szoftvermérnöki alapelvek egyikét sem.)

A módszertan biztosította a validitási vizsgálatokat is. Az egyes szintek közti átmenet során a szükséges validálás, valamint alacsonyabb absztrakciós szinten megadott modellnek az előző, magasabb absztrakciós szintre való beágyazása az alacsonyabb szintű és a felső szintű modell összevetése útján történt. Így a validálás a reverse engineering módszertannal volt végezhető.

Az elfogadható tervezési döntések funkcionális és architekturális következményeinek közvetlen retrospektív specifikációja jellemző vonása az új paradigmának: A tervezési döntés elfogadhatóságának formális bizonyítása (melléktermékeként) megadta mindazoknak a funkcionális és architekturális részleteknek a formális specifikációját (a tervezési döntés szintjétől függő absztrakciós szinten), melyeknek a célszoftver eleget tesz a vizsgált tervezési döntés következtében.

A fejlesztés végén nyert formális specifikáció

- pontosan definiálta a végtermék funkcióit,
- jól tükrözte annak architektúráját,
- támogatta a végtermék működésének megértését különféle absztrakciós szinteken.

Ha a végtermék módosításra szorult, akkor formális specifikálásának felelőssége és felhasználóbarát vonásai feljogosították ezt a specifikációt arra, hogy fő referencia pont legyen a tervezési döntésnek - mely ki van téve a szükséges változtatásoknak - a lokalizálásánál, valamint a szükségesnek tartott változtatás esetleges mellékhatásaival kapcsolatos okfejtéseknél. A célszoftver minden egyes változtatása esetén módosítani kellett az eredeti specifikációjának szövegét.

6.4. A fejlesztési környezet

Az előző pontban ismertetett módszertant megvalósító fejlesztési környezet célja az volt, hogy

1. A gyakorlati szoftvermérnöki kontextusban hatékonyan használható eszköz legyen a nagyméretű szoftver-rendszerek menedzselésénél.
2. Információ kinyerő rendszer legyen azon vizsgálatok számára, melyek a szoftvermérnökséget egy probléma megoldó diszciplínaként kezelik hasonlóan a mérnöki tervezés más ágaihoz.

A funkcionális szemantikai szabályrendszer sajátosságai lehetővé tették egy olyan szoftver (a továbbiakban R[everse] E[ngineering] S[ystem]) kifejlesztését, amely a felhasználó helyett automatikusan megkonstruálta a tárgyszoftver funkcióinak formális definícióját és az eredményt ”emészthető” formában (egy alkalmas grafikus nyelven) a felhasználó által igényelt részletezettséggel jelenítette meg.

A javasolt szoftver-fejlesztési környezet (SzFK) a következő komponensekből állt:

(1) kész és fejlesztés alatt álló szoftverek tárháza,

(2) a RES-t alapszolgáltatásként tartalmazta, és

(3) egy sor olyan, a RES-re épülő járulékos szolgáltatást biztosított, amelyek együttesen *teljes életciklust lefedő támogatást* nyújtottak az SZFK-ban megjelenő szoftverek számára az első tervezési döntés megjelenésétől a szoftver felhasználásból való kivonásáig -- azaz a szoftver "haláláig".

A kész és a fejlesztés alatt álló szoftverek megértését segítette az SZFK azon szolgáltatása, amely a tárgyszoftver egészének vagy (a felhasználó által kijelölt) tetszőleges szeletének funkcionális és architektúrális sajátosságait olyan részletezettséggel jelenítette meg, amilyenre a felhasználónak adott feladata elvégzéséhez szüksége volt.

Az SZFK négy főbb összetevőből állt:

(i) Reverse Engineerig System (RES)

Ez a rendszer egy szoftverrendszer (vagy egy szoftverrendszer részeinek) specifikálását végezte el a kódja alapján. A RES el tudott fogadni részben implementált szoftver rendszereket is, feltéve, hogy a célrendszer nem-implementált darabjait input folyambeli specifikációik képviselték. A keresett funkcionális specifikációt tároltuk, és addig marad elérhető, ameddig nem törölte a felhasználó. A keresett specifikáció formális leírása részletességének szintjét a felhasználó határozta meg.

A RES felkészíthető volt arra is, hogy - bizonyos feltételek mellett - tervezés alatt álló szoftverek értelmezésére is alkalmas legyen, hiszen a funkcionális szemantikai szabályrendszer mechanikus alkalmazása nem igényelte azt, hogy a tárgyszoftver kódja teljes egészében rendelkezésre álljon.

(ii) Konzisztencia/Teljesség Ellenőrző (KTE)

A programozási környezet ezen alrendszere elemezte a modellek (szövegek) konzisztenciáját és teljességét. Ha a célszöveg inkonzisztensnek és/vagy nem teljesnek bizonyult, kérte a felhasználót, hogy javítsa ki. A KTE segítette a szöveg kijavítását megfelelő segédinformáció biztosításával.
Ha a célszöveg a RES-ből származott, a felhasználó a specifikáció helyett a kód javítását is választhatta. Konzisztens, de nem teljes specifikáció esetén a felhasználó kérhette a specifikáció helyettesítését annak teljes magjával.

(iii) Relevancia ellenőrző (RE)

Az RE felismerte a célspecifikációval kapcsolatos azon igényeket, melyeket teljesen lefedtek más igények. A fölösleges igényeket teljesen törölte. A célspecifikációt helyettesítette az így optimalizált modellel.

(iv) Módosítás támogató részrendszer (MTR)

Az MTR lehetővé tette egy programrendszer tervezett módosítása következményeinek elemzését és a kívánt módosítások optimális megvalósítását. Felhívta a felhasználó figyelmét minden lehetséges mellékhatásra, melyet a módosítás kiválthat. Az MTR-t egy grafikus interfész támogatta, mely vizualizálta a formális specifikációkat és könnyűvé tette a rendszer használatát.

Ezen kutatások kapcsán is követtük sajátos publikálási szemléletünket., nevezetesen azt, hogy a legfontosabb eredményeket belső kiadványokban publikáltuk.

6.5. Nemzetközi kapcsolatok

Az SzFK kifejlesztését a Kijevi Kibernetikai Intézet munkatársaival közösen végeztük. Megjegyzem, hogy bár a kutatások számos eredményt hoztak, egységes alkalmazói rendszerben nem tudtak megvalósulni. Ennek alapvető oka a forráshiány volt, amire jellemző példa a

következő eset, Dines Bjørner Magyarországon járt egy szoftvertechnológiai rendezvényen és a a fehér asztal melletti borozgatást követő enyhe borgőzös állapotban azt mondta, hogy „nehogy már ti innen mondjátok meg, hogy merre fejlődjön a szoftvermérnökség tovább! A jó ötleteiket vagy meg tudjuk venni, vagy ellenkező esetben ellehetetlenítünk titeket.

6.6. Hazai fogadtatás

A hazai szakmai közönség sem értette meg a javaslataink lényegét. Ennek fényes bizonyítékát adta az MTA szakmai gréniuma Halmay Edit PhD értekezésének honosítási eljárása során. Halmay Edit az SZFK-val kapcsolatos kutatási eredményeit egy nemzetközileg is magas színvonalú PhD disszertációban foglalta össze, amelyet sikeresen védett meg 1989-ben Londonban, a South Bank Polytechnic-en. Egyik opponense Tom Maibaum, az Imperial College tanszékvezető professzora volt. Maibaum nagyon nagyra értékelte az elért eredményeket és javasolta a disszertáció könyv alakban való kiadását is. Halmay Edit ezt követően beadta az értekezését az MTA-hoz honosításra. Az MTA megfelelő osztálya úgy döntött, hogy a honosítást nyilvános védés keretében folytatja le. Ezen a védésen Halmay Edit nem kapta meg a szükséges pontszámot így a PhD cím honosítását jelentő kandidátusi cím odaítélését a bizottság elutasította. Ez is mutatta a hazai számítástudomány akkori siralmas színvonalát. Többek között ez is közre játszott abban, hogy amikor nem sokkal később szóba került, hogy vegyem át az ELTE számítástechnikai tanszékének vezetését, én ezt nem fogadtam el.

7. Az egységes számítástudomány összefoglalása.

Fő célunk egy olyan elsőrendű klasszikus logikai keret kidolgozása volt, amely alkalmas egy egységes számítástudomány kialakítására. Ez szembement azzal a széles körben elfogadott vélekedéssel, amely az elsőrendű klasszikus logika kereteit túl korlátozottnak tartotta arra, hogy alkalmas legyen kifejezni és bizonyítani a programok fontos tulajdonságait. Először is nézzük meg, miért részesítettük előnyben a klasszikus elsőrendű logikát. Matematikai eszköztára nagyon jól kidolgozott volt mind modell- mind bizonyításelméleti szempontból. Pragmatikai szempontból fontos, hogy a programozók aránylag jól ismerték a klasszikus logikát. A legfőbb érv azonban az, hogy az elsőrendű logika meglehetősen erős kifejező erővel rendelkezik ahhoz, hogy alapul szolgáljon egy rugalmas és erős programozás-elmélet kialakításához. Az is világos, hogy a változók döntő szerepet játszanak a programozásban, és az a formális nyelvnek, amelyet ki kívántunk fejleszteni a programok leírására, kezelnie kellett tudni a változókat. Ezért kellett legalább elsőrendűnek lennie. Persze vehettünk volna egy másodrendű logikát is, de ennek egy nagyon komoly hátránya van, nevezetesen ez nem teljes és így nem rekurzív. Ezért vettük az első rendű klasszikus logikát. A kidolgozott programozás-elmélet keretein belül a programváltozók szigorúan kapcsolódtak az adat-környezethez és az elmélet által kezelt egyik fontos entitást jelentették. Egy másik entitás az idő volt, amely kapcsolódott a programvégrehajtás reprezentálásához.

A klasszikus elsőrendű logika keretein belül sikeresen kidolgoztunk egy hatékony és erős kifejező erővel rendelkező programozás-elméletet. Az elmélet kidolgozása során két fő kutatás-fejlesztési irányt tartottunk szem előtt – egy tisztán logikait és egy programozás-elméletit.

A tiszta logikai megfontolások alapján a programozás-elmélet úgy lett kialakítva, hogy nagyon rugalmas legyen. Ezt a flexibilitást a következő módszerekkel biztosítottuk: a logikai kiterjesztéssel, az induktív definícióval, valamint ezt a két módszert kombináló induktív kiterjesztéssel. Az utóbbi lehetővé tette számunkra, hogy induktív definíciókat egy tiszta elsőrendű logikai keretben használjunk, ami azt is jelentette, hogy az induktív definíciók alapvető tulajdonságainak elsőrendű axiomatizálását tudtuk megadni. Ennek következtében lehetővé vált számunkra a fix-pont egyenletek használata, mégpedig úgy, hogy az egyenletek megoldása az elsőrendű kereteken belül lehetővé vált. A javasolt módszer lehetővé tette olyan klasszikus elsőrendű leíró nyelv hatékony generálását, amely nagy kifejező erővel rendelkezett és megfelelő jellemzését tudta adni gyakorlatilag bármely programozással kapcsolatos fogalomnak.

Az induktív definiálhatóságtól függetlenül az idő kezelésére egy megfelelően kidolgozott fix-pont elméletet adtunk meg. Ez az elmélet a véges sorozatok kódolhatóságára épült.

Hamarosan világossá vált, hogy pragmatikus szempontból célszerű lett volna kevesebb kódolást használni. Erre az egyik lehetőséget az alkalmazott logikai módszerek továbbfejlesztése kínálta. Nevezetesen, az időkiterjesztés helyett egy megfelelő halmazkiterjesztés alkalmazása jelenthette a továbblépést. Ez a változtatás egyszerűen végrehajtható volt, mivel programozás elméletünkben az idő függvények csak véges szeleteit használtuk fel.

A programozás-elméletünkben az alkalmazott logikai módszereknek megfelelően a definíciók is tartalmazhattak adatértékű paramétereket, de nem tartalmazhattak nem interpretált új relációkat, mint paramétereket. Egy fontos további kihívást jelentett egy új induktív kiterjesztés kidolgozása, amely lehetővé tette volna számunkra a reláció jelek paraméterként történő használatát.

A fent említett továbbfejlesztési igények párosultak még ahhoz az általános igényhez, hogy a programokat, programozási nyelveket és bizonyos mértékű programozással foglakozó elsőrendű

programozás-elméletet fejlesszük tovább úgy, hogy az a számítógéptudomány minél több területének vizsgálatára legyen alkalmas, mint pl. a különféle programozási paradigmákra (imperatív, deklaratív programozás), programok specifikációjára, szoftverek fejlesztésének támogatására, stb.

A továbblépést egy erre a célra kidolgozott halmazelmélet segítségével végeztük el. A számítástudomány számos területét lefedő új elmélet az öröklődően véges halmazok cFSA axiómarendszerére épült fel.

Erre az axiómarendszerre építve kidolgozott elmélet a következő területeket ölelte át:

a. a programozás egy új elméletét, amely alkalmas volt többek között különféle programozási módok (pl. szekvenciális, konkurens és párhuzamos programok) kezelésére,
b. a programok egy új elméletét, amely alkalmas volt többek között
    i. különféle szemantika (pl. relációs-, denotációs-, futás-szemantika) megadására és kezelésére,
    ii. adatbázis modellek megadására
    iii. absztrakt adat-típusok megadására és kezelésére
    iv. különféle programtulajdonságok (pl. parciális és totális helyesség) megadására és vizsgálatára
c. a programozási paradigmák új elméletét, amely alkalmas volt többek között
    i. az imperatív és deklaratív programozás vizsgálatára és kezelésére,
    ii. a logikai programozás megadására és kezelésére, pl. egy új logikai programozási nyelv, a LOBO is kidolgozásra került,
    iii. a funkcionális programozás megadására és kezelésére,
    iv. az imperatív, logikai és funkcionális programok együttes kezelésére.
d. a programok specifikációjának új elméletét, amely megad
    i. egy absztrakt kategória elméletre épülő specifikáció elméletet
    ii. egy konstruktív specifikáció elméletet
    iii. egy új specifikáció nyelvet, a HF nyelvet.
e. a szoftver mérnökség új elméletét és módszertanát.

Tehát a kitűzött célt, egy egységes matematikai megalapozással rendelkező számítástudomány kialakítását sikeresen elértük. A kidolgozott számítástudomány

(i) egységet tudott teremteni az egyes elméleti megoldások között legalább azzal, hogy összehasonlítási terepet biztosított számukra

(ii) a gyakorlat felől megjelenő új megoldások pl. új programozási paradigmák formális kezelésére és vizsgálatára is alkalmas eszközöket biztosított

(iii) az elméleti alapok a gyakorlat számára hatékony megoldások kidolgozását tették lehetővé, ami szoftvermérnöki módszertanban is megtestesült.

A cél elérése során matematikai szempontból négy fontos elvet követtünk:

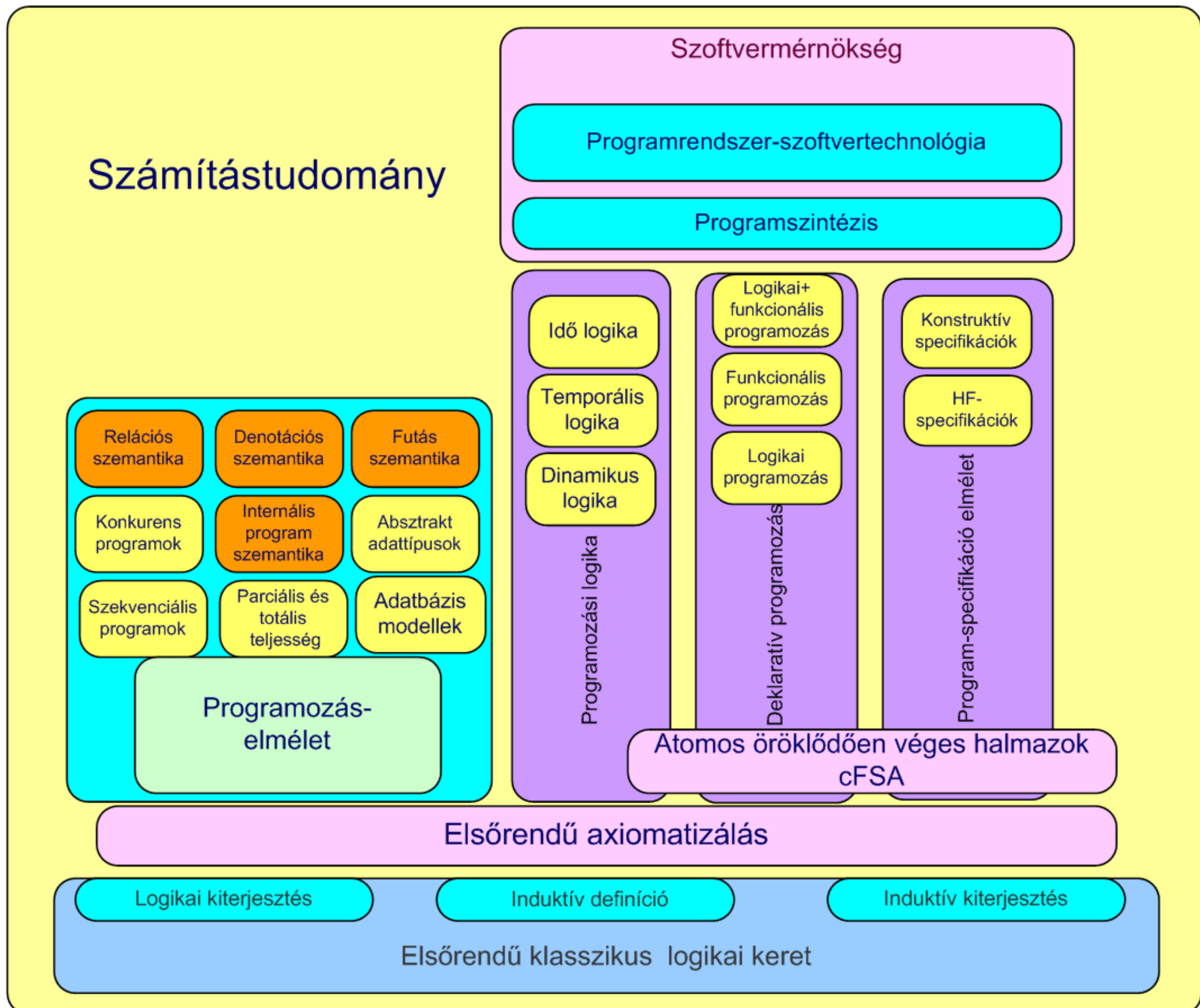
(i) megmaradni az elsőrendű logika szintjén és lehetőleg a klasszikus logika keretein belül,

(ii) a konstruktivitás szem előtt tartása,

(iii) a megfelelő definíció elmélet megadása a kiválasztott logikai keretben úgy, hogy az implicit definíciók megoldása konstruktív módon megvalósulhassanak ebben a keretben,

(iv) a számítástudományt egy megfelelő halmazelméleti alapokra épülő formális diszciplínaként állítsuk elő.

A kidolgozott számítástudomány felépítését az 1. ábra mutatja be.



1. ábra

8. Néhány Intézettörténeti megjegyzés

A kutatási munkákat számos támogatási szerződés segítette, elsősorban az OMFB részéről. A kutatások egy része szorosan kapcsolódott a Szovjetúnió által finanszírozott és általam vezetett ötödik-generációs szovjet-magyar projekthez, amely a Logikai Információs Számítási Rendszerek néven futott. Rövidítve ezt a projektet LIVSZ-nek nevezték az orosz megnevezés kezdőbetűi alapján.. Ebben a projektben a tudományos irányítást, valamint a projekt adminisztratív vezetését mi végeztük. A kutatás forrásellátottsága ellenére a SZÁMALK vezetése a kutatást igencsak mostohagyerekként kezelte és előszeretettel fordította a kutatásra befolyt pénzeket számára kedvesebb területek és emberek anyagi dotálására mindvégig hangsúlyozva, hogy milyen nagyra becsüli a kutatást. Ezt a visszás helyzetet kivédendő született meg a gondolat egy önálló kutató-fejlesztői intézmény kialakítására. 1986-ban megalakult az Alkalmazott Logikai Laboratórium Kutató-fejlesztő Kisszövetkezet, vagy "ALL" Számítástudományi Kutató Fejlesztő Kisszövetkezet, amely mind a mai napig működik, Azóta is számos hazai és nemzetközi kutatásban és kutatás-fejlesztés projektben vett és vesz részt. Ez azonban már egy másik történet…

A témakörben megjelent fontosabb publikációk a megjelenés sorrendjében:

A. Belső kiadványok:

1. Gergely,T., Szőts, M.: Programozási nyelvek szemantikájának matematikai logikai megalapozása. SZÁMKI Technical Report, Budapest, 1977..
2. Andréka, H., Gergely,T., Németi, I.: On a completeness result of program verification methods. SZÁMKI Technical Report, Budapest, 1977.
3. Gergely,T. Gondolatok a számítástudományi kutatások várható irányairól. SZÁMKI Technical Report, Budapest, 1978.
4. Gergely,T., Ury,L. Mathematical Theories of Programming, SZÁMKI Technical Report, Budapest, 1978.
5. Gergely,T. et al. A számítógépes programhelyesség-bizonyító kutatások helyzete. OMFB 16-7610-I.T, 1979 január.
6. Gergely,T. et al: A hazai számítástechnika távlati kutatások célszerű irányai. OMFB T-2222, Budapest, 1980.
7. Gergely,T., Ury,L.: A first order theory of dynamic logic. SZÁMALK Technical Report, Budapest, 1983
8. Gergely,T., Ury,L.: First order denotational semantics. SZÁMALK Technical Report, Budapest, 1983.
9. Gergely,T., Ury,L.: A uniform approach to programming logics. SZÁMALK Technical Report, Budapest, 1985.
10. Gergely, T., Szots, M. Logical Foundation of Logic Programming, SZAMALK Technical Report. Budapest, 1985.
11. Gergely, T., Ury, L. A constructive type theory, SZAMALK, Technical Report, 1985.
12. Gergely,T., Ury,L.: A uniform approach to programming logics. SZÁMALK Technical Report, Budapest, 1985.
13. Gergely,T., Ury,L.: Konstruktív programozási logika, ALL-Technical Report Series, Budapest, 1986.

14. Gergely,T. et al: Az elektronizáció számítástechnikai hírkőzlési és automatizálási K+F feladatai. G1 jel– VII. ötéves tervi OKKFT program I. és II. kötet 2. szerkesztés, OMFB, Budapest, 1986.
15. Gergely,T., Ury,L. A constructive basis for programming and specification theory, Applied Logic Laboratory, Technical Report, Budapest, 1986
16. Ury, L., Gergely, T. Hereditarily finite sets as a basis for program specifications, Applied Logic Laboratory, Technical Report, Budapest, 1987.
17. Gergely,T., Ury,L. Specification theory based on logic categories, Applied Logic Laboratory, Technical Report Series, Budapest, 1990
18. Gergely,T., Ury,L.: Logic and Functional Programming, Applied Logic Laboratory, Technical Report Series, Budapest, 1989
19. Gergely,T., Ury,L.: Constructive Programming Theory, Applied Logic Laboratory, Technical Report Series, Budapest, 1989

B. Cikkek

1. Andréka, H., Gergely,T., Németi, I.: On a direction of non-numeric application of computers. Inform ció-Elektronika 1. pp.52-57. (in Hungarian)
2. Andréka, H., Gergely,T., Németi, I.: A számítógépek nem-numerikus felhasználásának egy új irányzatáról,(On a direction of non-numeric application of computers.) Információ-Elektronika, 1. szám, 1975. 52-57.old.
3. Gergely,T., Szőts,M.: On the incompleteness of proving partial correctness. Acta Cybernetica. Fom.4. Fasc.1. 1978. pp. 45-57.
4. Gergely,T. May the theory of programming be first order? In: collection of Abstracts of the Conference on Mathematical Logic in Computer Science, Salgótarján, 1978
5. Gergely,T., Ury,L.: Nondeterministic programming within the frame of first order classical logic. Part 1 and Part 2. Acta Cybernetica, Tom.4. Fasc.4. Szeged, 1980. 334-354 and 356-375
6. Gergely,T., Ury,L.: Program behaviour specification through explicit time consideration in Lavington,S.M. (ed). Information Processing 80. North Holland, 1980. pp.107-111.
7. Gergely,T., Ury,L.: Programming in topoi, a generalized approach to program semantics, in Dittrich,G., Merzenich,W. (eds). Extended Abstracts of the 3rd Workshop Meeting on Computer Science and System Theory, Dorthmund, 1980.
8. Gergely,T.: Szemantika a számítástudományban, I. és II. rész. Információ és Elektronika 1980. 6. szám. 304-309.
9. Gergely,T., Ury,L.: Time models for programming logics, in Dömölki, B., Gergely,T. (eds). Mathematical Logic in Computer Science, North Holland, Amsterdam, 1981. pp. 359-427.
10. Gergely,T., Ury,L.: A theory of interactive programming, Acta Informatica, 17, 1982, pp. 1-20.
11. Gergely, T., Szőts, M.: About representation of semantics, in Trappl, R., Finder, N.V., Horn, W. (eds), Progress in Cybernetics and System Research, vol.11, Hemisphere P. Corp. 1982.
12. Gergely, T., Ury, L. Adequate characterization of Hoare logic, Semiotics and Informatics, Vol.22., 1983. pp. 81-101 (in Russian).
13. Gergely,T., Szőts,M. Cuttable formulas for logic programming, In: Proceedings of International Symposium on Logic Programming, IEEE Press,1984. pp. 279-310.
14. Gergely,T., Szőts,M. Some features of a new logic programming language, In Revay P. (ed.), Proceedings of the Workshop and Conference on Applied AI and Knowledge Base Expert Systems, Norsteats Trykeri A.B., Stockholm 1984. pp. 127-143.
15. Gergely, T., Ury, L.: A constructive specification theory, in: David,G., Boute, R.T., Shriver, B.D. (eds). Declarative Systems, Elsevier (North-Holland) 1990. pp. 33-82
16. Gergely, T., Ury, L.: A unique logical framework for software development, Artificial Intelligence News, Moscow, 1993 pp 62-81

17. Gergely, T., Halmay, E.: Forth-and-Back Model of Software Development, in: Magyar Informatikus I. Világtalálkozója, Gábor Dénes Műszaki Főiskola, Budapest, 1996.

C. Könyvek

1. Gergely, T., Ury, L.: First Order Programming Theories, Springer Verlag, Heidelberg, 1991

D. Tudományos értekezések

1. Szots, M. A general first order theory of logic programming, Ph.D. Thesis, Budapest, 1986.
2. Gergely T., Non-standard Programming Logic. Hungarian Academy of Science, Budapest, 1987, D.Sc. Thesis
3. E.Halmay: Formal method for the retrospective specification of the functionality of existing software systems, PhD thesis, South Bank Polytechnic, London 1989