Konputagarritasunerako Oinarrizko Teknikak

Arantza Irastorza, Ana Sánchez, Jesús Ibáñez

UPV/EHU / LSI / TR 01-2017

KONPUTAGARRITASUNERAKO OINARRIZKO TEKNIKAK

Arantza Irastorza

Ana Sánchez

Jesús Ibáñez

Aurkibidea

1. Sarrera

Konputagarritasunaren Teoria Informatika Teorikoaren barnean sartzen den arloa da, sistema informatikoek algoritmoen diseinuaren bitartez problemak ebaztean dituzten muga logikoak ezartzea helburua duena, hain zuzen ere. Egunez egun konputagailuen aplikagarritasun praktikoa zabaltzen doazen diziplina eta tekniken aurrean, informazioaren prozesamendurako edozein teknologia digitalek gainditu ezingo duen muga-taldea ezartzen du teoria honek. Oinarrizko Lege moduan ezartzen ditu, informatikaren existentziaren baldintza propioak zuzentzen dituzten legeak bezala.

Konputagarritasunaren Teoriaren berezko metodoak izugarri konplexuak izan daitezke, eta gehien bat alderdi teknologikora bideratutatako formakuntzarentzat arraro samarrak suerta daitezke, batez ere. Bere emaitza aurreratuenak oso formakuntza tekniko ona duten informatikari esperientziadunentzat ere ulertzeko zailak dira. Hala ere, oinarrizko emaitzen nukleo bat badago, eskura dauden tekniken bitartez ekin daitezkeenak, eta erabakiezintazun konputazionalaren kontzeptu nagusia isladatzeko bertutea badutenak. Esan bezala emaitza horiek zenbait teknikei esker erator daitezke, Konputagarritasunaren Teoriaren bere-berezkoak ez izan arren, arlo honetan guztiz emankorrak suertatu direnak.

Txosten honetan Teoriaren oinarrizko nukleo hori osatzen duten kontzeptu eta tekniken deskribapena sartzen da. Informatikan ezagunak eta nabarmenak diren zenbait problemen konputaezintasunarekin erlazionatutako emaitzen lehenengo bateria ematea da bere helburua. Emaitzak aurkeztean while-programena izango da erabiliko den programazio estandarra, aurreko txosten batean [IIS 98] deskribatu genuena, eta Diagonalizazio teknikaren azalpen zehaztua eta sistematikoa ere sartuko da.

UPV/EHUko Informatika Fakultatean ematen den Ingeniaritza Informatikoko Ikasketa Planaren *Konputazioaren Eredu Abstraktuak II* irakasgaiaren ikasleentzat irakaskuntza laguntza moduan idatzia izan den arren, egileen asmoa edo nahia, lehen aipaturiko emaitza eta tekniken deskribapen ulergarria ematea da. Txosten hau aurreko paragrafoan aipaturikoaren jarraipena den arren, azken hori irakurtzea ekidin nahi duenak, oinarrizko kontzeptu eta definizioak bigarren kapituluan aurkituko ditu, eta while-programen lengoaiaren despribapen zehaztua eta testuingurua osatzeko beste zenbait informazio, berriz, eranskinetan.

Hala ere, lan honetara [IIS 98] txostenaren ondoren iristen den horrek, 2. kapituluan definizio ezagun asko aurkituko ditu (nahiz eta, bertan sartu ez ziren zenbait definizio ere badauden), eta ziurrenik eranskinen edukia azalekotzat hartuko

du. Hemen dokumentu hartan azaldu ziren emaitzak hartu eta bertan ezarri ziren hitzarmen guztiak mantenduko ditugu, makroprogramen goiburukoari buruzko hura izan ezik. Aipatu txosteneko 3.5 atalean aurretik inplementaturiko funtzioen inportazioaren nozioa sartu genuen, **package** KONPUTAGARRIAK notazioaren bitartez eta konputagarritasun frogapenen prozesu inkrementalaren zentzu hierarkikoa barneratzeko bide bezala; baina gauden puntu honetan hau ez da beharrezkoa eta, aldiz, neketsua gertatzen da zehaztasun hori sistematikoki adieraztea.

3. kapituluan lan honen gai nagusiari ekiten diogu, formala ez den ikuspegi batetik oraindik. Konputaezintasunaren kontzeptua, sistema informatikoei buruz uneko ezagueran oinarritutako ausazko interpretazioetatik banatzea garrantzitsua da. Hori egiteko era egoki bat bereizketa egitea da: funtzio konputaezinen eta konputagarritasun eskuragaitza duten funtzioen artean bereiztea. Era berean, horrek Konputagarritasunaren Teoriaren analisia guztia oinarritzen den funtsezko premisa ezartzera garamatza: konputaezintasunaren ezaugarriak aukeratutako programazio sistemaren independenteak dira.

5. kapituluan, funtzioen konputaezintasuna eta predikatuen erabakiezintasuna frogatzeko funtsezkoa den diagonalizazio teknika nahiko zehaztasunez azaltzen da. Bere aurretik, berriz, gure helburuetarako guztiz erabilgarriak diren zenbait emaitza eta teknika biltzen duen kapitulua ere sartu da. Aipatu emaitza horiek zenbait egite nabarmenen froga dira, esaterako, helburu orokorreko ordenadoreak egon behar direla edo prozesu paraleloen ereduen eta sekuentzialen arteko baliokidetasuna, baina, batez ere, funtzio nabarmen askoren konputagarritasuna zein konputaezintasuna frogatzeko erabiliko diren tresna garrantzitsuak ematen dituzte.

Behin diagonalizazioa eta bere aplikazioak aztertu ondoren, erabakitze-problematan zentratutako azken kapitulu bat sartu da. Honek kontzeptu berria azaltzen du, sasierabakigarritasuna, zenbait problemen konputaezintasunaren inguruan ikuspegi posibilistagoa edukitzeko argigarria gertatzen dena.

2. Oinarrizko nozioak eta notazioa

ALFABETOA ETA HITZA: Informazioaren prozesamenduaren nozioa, ikurren konbinaziotik eratortzen diren objektuen eraldaketa kontrolatuaren ideiara lotuta dago. Horregatik, ***alfabetoa*** ikurren edozein $\sum$ multzo finituri esango diogu. Edozein $\sum$ alfabeto emanik, bere gainean definituriko ***hitzen*** edo ***kateen*** multzoa $\sum^*$ definitzen dugu. Hitz hutsaren (ε) eta hitzen kateamendu eta alderanzketaren nozio klasikoak erabiliko ditugu. Hauek eta ikurrak maneiatzeko beste funtzio batzuk C eranskinean (***a*** atalean) zerrendan ageri dira.

Hitzak dira domeinuko objektuak errepresentatzeko aukera ematen diguten elementuak, eta beraz, informazioaren oinarria osatzen dutenak. Horrela, programen eginkizuna, beste hitz (emaitza) batzuk lortzeko zenbait hitz (datu) maneiatzea izango da.

FUNTZIOAK: Programen portaera sarrera/irteera moduan deskribatzeko, hitzen arteko ***funtzio partzialak*** erabiliko ditugu. Hauek, bere balizko sarreretako zenbaitentzat indefinituta egon daitezke. $\psi : \sum^* \longrightarrow \sum^*$ funtzioak, sarrerako **x** hitzaren gainean aplikaturik, ***konbergitzen*** badu (irudia badauka) $\psi(\mathbf{x})\downarrow$ ikurraren bitartez adieraziko dugu. Aldiz, puntu horretan funtzioa indefinituta badago, $\psi(\mathbf{x})\uparrow$ idatziko dugu eta ***dibergitu*** egiten duela esango dugu.

Argumentu bat baino gehiago duten baina betiere emaitza bakarra itzultzen duten funtzioak ere erabiliko ditugu, $\psi : \sum^{*\mathbf{k}} \longrightarrow \sum^*$ erakoak, hain zuzen ere. Kasu horretan $\psi(\mathbf{x_1},...,\mathbf{x_k})\downarrow$ edo $\psi(\mathbf{x_1},...,\mathbf{x_k})\uparrow$ adieraziko dugu.

PROGRAMAK: Programa ikurrak maneiatzen dituen prozesu baten espezifikazio ez-anbiguoa da, eta prozesu horrek sarrerako katea (datu) bat edo batzuk eraldatzen ditu, irteerako beste bat (emaitza) lortzeko. Programa, programazio-lengoaia definitzen duten arau sintaktiko batzuen arabera eraikita egon behar da. Gure kasuan erabilitako lengoaia while-programena da, bere sintaxi zehatza A eranskinean deskribatzen da baina deskribapen zehatzagoa [IIS 98]-n kontsulta daiteke.

P programa baten portaera, funtzio partzial baten bitartez deskribatua izango da. Funtzio hori $\varphi_{\mathbf{P}} : \sum^* \longrightarrow \sum^*$ eran idatziko dugu eta **P**-k jasotzen duen sarrera-katearen eta emaitza moduan sortzen duenaren artean dagoen erlazioa deskribatuko du. Emaitza bat itzultzeko, programak bere egikaritzapena bukatu egin behar duenez, **x** sarreraren gainean **P**-ren portaera indefinituki ziklatzea denean, hori $\varphi_{\mathbf{P}}(\mathbf{x})\uparrow$ eraren bitartez islatuko da, programaren emaitza indefinitua delako. $\varphi_{\mathbf{P}}$ funtzioa ***osoa*** bada (hots, bere balizko argumentu guztientzat definituta badago) orduan **P** programa edozein baldintzen pean bukatzen da. Beste muturrean

φ_P funtzio hutsa ($\perp\!\!\!\perp$) deneko kasua dago: hau beti indefinituta dago eta ondorioz, **P**-k edozein sarrerarekin ziklatu egiten duela adierazten du.

Kontzeptua era naturalean hedatzen da, datu bakarraren ordez **P** programak sarreran **k** datu jasotzen dituenean. Orduan bere portaera deskribatzen duen funtzioa honakoa da: $\varphi_P^k : \sum^{*k} \longrightarrow \sum^*$.

KONPUTAGARRITASUNA: $\psi : \sum^{*k} \longrightarrow \sum^*$ funtzioa ***while-konputagarria*** da **P** while-programa existitzen bada bera konputatzen duena ($\varphi_P^k \cong \psi$), hots, bere argumentuen edozein baliotarako sistematikoki funtzioaren emaitzak lortzen ditu. ***Konputagarritasuna*** baino ez esaten badugu, ψ funtzioa konputatzeari buruz ari gara baina irudikatu dezakegun edozein lengoaia edo programazio sistemarekin erabiliz eta ez while-programen lengoaiarekin ezinbestez. While-konputagarritasuna eta konputagarritasunaren artean dagoen erlazioa 3.2 atalean aztertuko dugu.

Txosten honen helburua funtzio konputagarri eta konputaezinen arteko bereizte mekanismoak ezartzea da.

PREDIKATUAK: Konputagarritasunaren Teorian garrantzia berezia duten funtzioen klase berezi bat dago: predikatuak, edo funtzio boolear osoak (ikus 6. kapitulua). Funtzioak while-konputagarriak edo konputaezinak bezala sailkatzen ditugu funtzioaren balioak kalkulatuko dituen while-programa dagoenaren arabera, eta era berean predikatuak ere sailka ditzakegu, bere egiazko kasuak eta kasu faltsuak bereiztuko dituen programa dagoenaren arabera. Horrela, $\mathbf{S} : \sum^* \rightarrow \mathbb{B}$ predikatua (non $\mathbb{B}$ {*true*,*false*} balio boolearren multzoa den) ***while-erabakigarria*** dela esaten dugu, baldin eta **P** while-programa existitzen bada $\mathbf{S}(\mathbf{x})$ betetzen duten balio guztietarako $\mathbf{r_1}$ emaitza sortzen duena eta beste emaitza deberdin bat $\mathbf{r_2}$, $\neg\mathbf{S}(\mathbf{x})$ betetzen den balioetarako. Bere izenak berak adierazten duen bezala, predikatu bat while-erabakigarria da bere egiazkotasuna while-programa baten bitartez erabaki badaiteke.

Ideia hori **k**-tarrak diren predikatuetara era naturalean hedatzen da, honela: $\mathbf{S} : \sum^{*k} \rightarrow \mathbb{B}$.

MAKROPROGRAMAK: funtzio konplexuen while-konputagarritasuna frogatzeko lana erraztearren while-programen gainean ***makro*** lengoaia bat definitzen da, horrela programen idazketa laburtzea eta beren esanahia hobeto ulertzea ahalbidetzen dute. Ondorioz sortzen diren laburdurei makroprograma deitzen diegu, eta horrela, lehendik beren konputagarritasuna frogatu zaien funtzioak erabil ahal izango dira, lengoaiaren parte izango balira bezala. B eranskinean makroen klase garrantzitsuenak deskribatzen dira, makro horien

erabilera justifikatuta geratzen da benetan while-programen lengoaiaren berezko egituren baliokideak direla (egitura horietara ***hedatu*** daitezkeela) frogatuz.

Makroprogrametan edozein funtzio while-konputagarriri edo predikatu while-erabakigarriri deiak egiteko aukera ematen denez, programazio-lengoaian gehituko bagenitu bezala, C eranskinean hainbat funtzio eta predikaturen zerrenda ematen da, horien while-konputagarritasuna [IIS 98] erreferentzian frogatu zen eta hemendik aurrera idatziko diren programetan erabiliko dira. Konbentzio edo hitzarmen hauei guztiei esker, lanerako edukiko dugun lengoaia while-programena baino askoz aberatsagoa eta maneiatzeko erosoagoa izango da.

DATU-MOTAK: while-programak karaktere-kateen (hitzen) gainean soilik lan egiteko definiturik dauden arren, $\sum^*$-ko elementuen eta beste datu multzo batzuen arteko elkarrekikotasun errazak definitzea posible da, eta horrela kate bakoitzak multzo berriko elementu bat errepresentatuko duelarik eta hitzen gaineko konputazioa mota berriko elementuen errepresentazioarekin bat etorriko delarik. Horrela, errazak eta baliogarriak diren datu-mota batzuen ***inplementazioa*** lortzen dugu, hala nola, ***boolearrak*** $\mathbb{B}$, edo ***zenbaki naturalak*** $\mathbb{N}$. Era berean mota egituratuak inplementatzen dira, esaterako ***pilak*** $\mathbb{P}$ eta ***bektore dinamikoak*** $\mathbb{V}$. Azkenik ***while-programak*** $\mathbb{W}$ ere datu-mota bezala inplementa daitezkeela frogatzen da. Honek guztiak $\sum^*$–ren desberdinak diren domeinuetan funtzio konputagarriekin lan egitea ahalbidetzen digu, eta gure makroprogrametan mota horien elementuak eta eragiketak erabiltzea ere.

C eranskineko ***c***, ***d*** eta ***e*** ataletan datu-mota horiekin definituriko funtzioen eta predikatuen zerrendak ematen dira, beraien while-konputagarritasuna [IIS 98] erreferentzian frogatua izan zen eta hemendik aurrera eraikiko ditugun makroprogrametan erabiliko dira. Berriz ere abantaila bezala programazio-lengoaiaren aberasketa lortzen dugu, gure helburuetarako oso baliogarriak diren datu-moten berezko objektuak (konstanteak, funtzioak eta predikatuak) gehituz.

HITZEN ZERRENDATZEA: boolearren kasuan izan ezik (arrazoi nabariengatik) inplementazioak bijektibo moduan defini daitezke beti. Honek esan nahi du, adibidez, zenbaki naturalen eta horiek errepresentatzeko erabiltzen diren $\sum^*$–ko hitzen artean korrespondentzia biunibokoa badagoela, eta hori aukeratutako $\sum$ alfabetoaren guztiz independientea gertatzen dela. Horregatik, edozein hitz multzo, $\sum^*$, honela ***zerrenda*** dezakegu, $\sum^* = \{\mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3, ...\}$, non $\mathbf{w}_i$ hitza i zenbakia errepresentatzeko balio duena den. Behin ordena definituta, sarrera-datu bezala jasotako hitzaren ondoren datorren hitza lortzen duen funtzioa programa dezakegu (*hur* funtzioa), edo aurreko hitza lortzen duena (*aurre* funtzioa).

Gure kasuan aukeratutako ordena hitzak luzeraren arabera ordenatzen dituena da, eta luzera berdinarekin orden lexikografikoa aplikatzen duena. Horrela, {**a**, **b**, **c**} alfabetoarekin hitzen gainean ezarritako ordena honakoa litzateke: **ε - a - b - c - aa - ab - ac - ba - bb - bc - ca - cb - cc - aaa - aab - aac** ..., non $\mathbf{w_0} = \varepsilon$ eta $\mathbf{w_9} = \mathbf{bc}$, hur(**abc**) = **aca** eta aurre(**aaaa**) = **ccc** izango diren.

KODEKETA-FUNTZIOAK: funtzio konputagarrien beste bilduma bat ere bereiz dezakegu, kodeketa-funtzioak (*kod*^{k} funtzioak) deitzen ditugunak eta hitz segidak bakar batean trinkotzea edo taldekatzea ahalbidetzen dutenak. Horrela, *deskod* funtzioen bitartez, modu konputagarrian ere, kodeketatik abiatuz jatorrizko seriea berreskuratzea posible izango da. **k**=2 kasurako, hitz-bikoteei kodeak lotzeko modua 2.1 irudiko taulan adierazten da.

	$\mathbf{w_0}$	$\mathbf{w_1}$	$\mathbf{w_2}$	$\mathbf{w_3}$	$\mathbf{w_4}$	...
$\mathbf{w_0}$	w_0	w_2	w_5	w_9	w_{14}	
$\mathbf{w_1}$	w_1	w_4	w_8	w_{13}	w_{19}	
$\mathbf{w_2}$	w_3	w_7	w_{12}	w_{18}	w_{25}	
$\mathbf{w_3}$	w_6	w_{11}	w_{17}	w_{24}	w_{32}	
$\mathbf{w_4}$	w_{10}	w_{16}	w_{23}	w_{31}	w_{40}	
...						...

2.1 irudia: hitz-bikote baten kodea zehaztea ahalbidetzen duen elkarrekikotasun taula. Horrela, $\text{kod}^2(\mathbf{w_0}, \mathbf{w_3}) = \mathbf{w_9}$ da ($\sum = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ alfabetoaren adibide zehatzean $\text{kod}^2(\varepsilon, \mathbf{c}) = \mathbf{bc}$ dela adieraz dezakegu). Bere kodetik abiatuz, bikote baten elementuak berreskuratzen dituzten funtzioak $\text{deskod}_{2,1}(\mathbf{w_9}) = \mathbf{w_0}$ eta $\text{deskod}_{2,2}(\mathbf{w_9}) = \mathbf{w_3}$ dira.

C eranskineko ***b*** atalean funtzio hauen deskribapen zehatz eta xeheagoa ematen da.

PROGRAMEN ETA FUNTZIOEN ZERRENDATZEA: Hitzen zerrendatzea inplementaturiko edozein datu-motara iragan daiteke, eta while-programen kasua oso nabarmentzekoa da. Horiek honela $\mathbb{W} = \{\mathbf{P_0}, \mathbf{P_1}, \mathbf{P_2}, \mathbf{P_3}, ...\}$ zerrenda daitezke, non $\mathbf{P_i}$ programa $\mathbf{w_i}$ hitzak errepresentatzen duen programa den. Orduan $\mathbf{w_i}$ hitza (eta hedapenez, **i** zenbakia) $\mathbf{P_i}$ programaren ***kodea*** dela esaten da.

Zerrendatze horretan balizko programa guzti-guztiak daudenez, funtzio konputagarri guztien klasea ere zerrenda dezakegu, honela $\{\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots\}$, non φ_i funtzioa $\mathbf{P_i}$ programak kalkulatzen duen funtzioa den, programa hori $\mathbf{w_i}$ hitzak errepresentatzen duelarik. Horrela, $\mathbf{w_i}$ hitza (eta hedapenez, erlazionaturiko **i** zenbakia ere) φ_i funtzioaren ***indizea*** dela esango dugu.

Ondorioz, eta ψ funtzioa konputagarria dela adieraztea hori konputatzen duen **P** programa existitzen dela frogatzearen baliokidea denez, funtzioarentzako indize bat badagoela baieztatzearen baliokidea ere ba da, hots, indize hori **e** balioa da non $\psi \equiv \varphi_e$ den. Alabaina, kode bakarra duten programetan ez bezala, funtzio konputagarri guztiek infinitu indize dauzkatela azpimarratu behar dugu, baliokideak diren infinitu programa aurki baititzakegu. Programa baten testua eraldatzeko infinitu aukera daude, betiere bere semantika aldatu gabe.

Azkenik notaziorako beste hitzarmen bat azpimarratuko dugu: φ_i funtzioaren ***domeinua*** $\mathbf{W_i}$ bezala adieraziko dugu, eta bere ***heina*** $\mathbf{R_i}$[1] moduan. Multzo hauek programen portaera deskribatzeko orduan garrantzia dute, $\mathbf{W_i}$ multzoak $\mathbf{P_i}$ programak baliozkotzat jotzen dituen datuen multzoa errepresentatzen baitu, eta $\mathbf{R_i}$-k berriz, programa horrek berak sor ditzakeen emaitzen multzoari erreferentzia egiten dio.

[1] Zentzu hertsian, programa bera hainbat eratara erabil daiteke, sarreran ematen zaizkion datuen kopuruaren arabera. Lehenago deskribatu dugun zerrendatzeak argumentu bakarreko funtzioak bakarrik kontuan hartzen ditu. Argumentuen kopuru arbitrarioa duten funtzioak interesatzen bazaizkigu, $\mathbf{k}>0$ bakoitzerako $\{\varphi_0^k, \varphi_1^k, \varphi_2^k, \varphi_3^k \ldots\}$ zerrenda defini dezakegu, non φ_i^k funtzioa sarreran **k** datu ematen zaizkionean $\mathbf{P_i}$ programak konputatzen duen funtzioa den. Argumentu bakar batentzat esandakoa modu naturalean **k** argumentuetara zabaltzen da, eta bereziki $\mathbf{W_i^k}$ eta $\mathbf{R_i^k}$ notazioa, φ_i^k funtzioaren domeinurako eta heinerako, hurrenez hurren.

3. Konputagarritasuna versus konputaezintasuna

Funtzio while-konputagarritzat ulertzen duguna deskribatu dugu: funtzio bat zeinak bera konputatzen duen while-programa bat existitzea onartzen duen. Nozio hori, pentsa dezakegun edozein problematara heda daiteke era naturalean. Problema bat konputagarria izango da funtzio partzial baten bitartez adieraz badaiteke eta hau konputatu ahal bada (funtzio horren datuak eta emaitzak karaktere-kateak izango dira edo bere inplementazioa frogatuta duen datu-motaren batekoak). Adibidez, hauteskundetan hautagai-zerrenda bakoitzak lortutako boto kopurutik abiatuz eta D'Hont legearen arabera parlamentuko aulkien banaketa zehaztearen problema while-konputagarria da. Problema hori honako funtzioaren bitartez adieraz daiteke:

$$\text{banaketa: } \mathbb{N} \times \mathbb{V} \longrightarrow \mathbb{V}$$

non, baldin eta **n** banatu beharreko aulki kopurua eta **V** kandidatura desberdinen botoak gordetzen dituen bektorea badira, orduan banaketa(**n**,**V**) beste bektore bat den, **V**-ren dimentsio bera duena eta hautagaitzen aulkiak gordetzen dituena, hain zuzen ere. Funtzio hau ondo definituta dago (**n**,**V**) bikote zehatz batentzat aulkien balizko banaketa bakarra dagoelako[2]. Gainera badakigu konputagarria dela, informatikariek beste programazio-ingurune batzuetan aspaldi ebatzi dutelako (zalantzaren bat geratuz gero, egiaztatzea oso erraza izango litzateke, zuzenean makroprograma txiki bat eginez).

Oraingo honetan galderak beste batzuk dira, problema bat while-konputagarria ez dela baieztatzeak zer esan nahiko luke? Eta, behin hau argi edukita, problema konputaezinak ba al daude? Eta baldin badaude, nola aurki ditzakegu eta nola antzeman?

Funtzioa while-konputaezina dela diogu hori konputatuko duen while-programarik ez dagoenean. Eta problema konputaezina dela ulertzen dugu karaktere-kateen arteko funtzio moduan adieraz daitekeenean baina funtzio hori konputaezina suertatzen denean. Beste era batera esanda, problema funtzioren baten bitartez adieraztea lortzen ez badugu (ikus orri-oineko oharra), edo bere datuak edo emaitzak ikur-kateen bitartez adierazteko zailtasunak baditugu, orduan problema espezifikatzen zaila (edo ezinezkoa) da, baina konputagarria denetz eztabaidatzen

[2] Berez, hori egia izango da soilik baldin eta boto-kopuru bera lortuko luketen hautagaitzen arteko berdinketak hausteko arau zehatz eta determinista badago. Araudiak zozketa-motaren bat aurreikusten badu, orduan *banaketa* ez da ondo definitutako funtzioa izango, emaitza ez baita datuen menpekoa bakarrik izango. Kasu horretan problema ezin da funtzionalki adierazi eta while-konputagarria denetz galdetzeak ez du zentzurik.

hasteak ez dauka zentzurik. While-konputagarritasuna, beraz, problemaren espezifikazio funtzionala egin eta gero planteatzen da, hori dela eta, orokorrean funtzio (eta ez problema) konputagarriei eta konputaezinei buruz hitz egitera mugatuko gara.

> *Beren espezifikazioa oso zaila den problema pila dago bizitza errealean. Adimen Artifizialean ohiko kontua da. Adibidez, problema bezala "xakean ondo jokatu" planteatzen badugu, nola jar dezakegu hori sarrera/irteera moduan, problema hori ebatziko duen programa eraikitzeko? Ahal den gehiena zehaztuta ere (adibidez, xakean ondo jokatzea ELO puntuaketan 7.500 baino gehiago lortzea dela esanez) problemak funtzio erako planteamendua onartu gabe jarraituko du. Arlo honetan ezin ditugu ez konputagarritasunaren ezta konputaezintasunaren kontzeptuak aplikatu.*
> *Jakina, honek ez du esan nahi Informatikak problema hauek alboratzen dituenik, baina beren ebazpideari ekin ordez (zentzurik ez duena problemaren "ebazpenaz" hitz egin ezin dugulako) beren hurbilketarako teknikak erabiltzen ditu (hala nola, heuristikoak, probabilistikak, etab.).*

Zein arlotan mugituko garen argitu ondoren, funtzio while-konputaezinak badauden edo ez galdetzea dator. Zalantza egokia da: informatikariak konputagarritasunaren definizioa lehenengo aldiz aurkitzen duenean oso arraroa da programatzea ezinezkoak diren problemak badaudela aurretik planteatu izana. Bere esperientziaren arabera, batzuk beste zenbaitzuk baino zailagoak direla jakingo du, baina oro har nahiko ahaltsuak diren tresnen bidez konputazioaren munduan dena posible izaten bukatzen dela. Alabaina, hori faltsua da: problema konputaezinak egon ez ezik, beraien kopurua konputagarriena baino askoz handiagoa da, aurrerago ikusiko dugun bezala.

Demagun informatikari bat honako problemaren aurrean aurkitzen dela: edozein programa emanda, bere eraiketan erabilitako espezifikaziora egokitzen dela egiaztatu, hau da, balizko edozein sarreratarako itxaroten den huraxe egiten duela egiaztatu. Problema hori funtzionalki defini daiteke, lehendabizi espezifikazioen $\mathbb{E}$ datu-mota definitu eta ikur-kateen bitartez inplementatzen badugu (adibidez, pre-post espezifikazioak erabil ditzakegu, programen egiaztapenerako Hoare-ren kalkuluan erabiltzen direnen antzekoak). Orduan, problema honako funtzioaren (predikatua, egia esan) bitartez definituta geratuko litzateke:

$$\text{espezifikazioa\_bete?}: \mathbb{W} \times \mathbb{E} \longrightarrow \mathbb{B}$$

Hala ere, funtzio hau konputatuko duen programa bat eraikitzen saiatzen bagara, lan erraza ez daukagula berehala konturatuko gara. Egiaz, bibliografia kontsultatuz gero, inolako programazio-ingurunetan inork ez duela programa hori

eraikitzea lortu ikusiko dugu, ondorioz while-konputagarria ere agian ez dela izango susmatzen has gaitezke. Zer egingo dugu orduan? Ondoko hauek bezalako argumentuak erabil ditzakegu: "nik ez dakit hori egiten", "egingo duen norbait kontratatzerik ez dut lortu", "Fakultatean ez didate horrelakorik irakatsi", "arazo hori ebazteko utilitateak eskaintzen dituen software enpresarik ez dut aurkitu amaraunean edo web-ean". Agian programa aurkitzeko ideia ederren eskukada bat behar dugu, agian eraikitzeko hemendik hogeita hamar urte barruko teknologia itxaron beharko dugu. Baina konputaezintasuna askoz serioagoa da: *espezifikazioa_bete?* funtzioa while-konputagarria ez bada, dagokion programa garatzeko inork inoiz ezta 5 minutu emateak ere ez du merezi, helburu hori besterik gabe ezinezkoa delako. Baina nola lor dezakegu egite horrentzat argumentu sinesgarriren bat? hau da, ezintasun horren inguruan mundu guztia ados jarriko duena. Nola pasa gaitezke "oraindik ez da lortu" bezalako inpotentziatik "ezinezkoa da" erako zorigaitzera? Logika matematikoan bakarrik aurkituko dugu arazo honentzat irtenbidea: programa existitzea, egia direla jadanik badakigun beste zenbait egiterekin kontraesanean egongo litzatekeela frogatuz.

3.1 Konputagarritasuna eta konputagarritasun eskuragaitza

Funtzio batzuen konputaezintasuna frogatzearen problemari ekin baino lehen, kapituluaren hasieran aipatu dugun "ezin dugu" eta "ezin da" esamoldeen arteko oinarrizko desberdintasun horren inguruan xehetasun gehiagorekin hausnartzea komeni da. Oraingoz, funtzioaren konputagarritasuna frogatzen saiatzen garenean hori kalkulatuko duen programa bat aurkitu behar dugu. Baina zenbaitetan funtzioa while-konputagarria dela frogatuko lukeen while-programa topatzea ezinezkoa da, eta, edozelan ere, funtzioa while-konputagarria da!. Baina nola? while-programak deskribatzera orrialde mordoa eskaini eta orain errepresentatzen duten eredu hori osoa ez dela aurkitu?, hau da, funtzio konputagarri guztiak biltzeko ez duela balio.

Urduritasunean murgildu eta beste eredu alternatiboren bat bilatzen hasi baino lehen, funtzio while-konputagarriaren definizioak "bera konputatuko duen programa bat existitzea" exijitzen duela beste behin azpimarratzea komeni da. Eta *programa existitzea* ez da *programa hori aurkitzeko gai izatea*-ren berdina. Jakina bigarrenak lehenengoa ondorioztatzen du, baina bien artekoa elkarrekiko erlazioa ez dela ikusiko dugu: zenbaitetan programa ziur existitzen da, alabaina zein ote den ezin dugu jakin. Orduan zalantza honakoa da: nola demontre dakigu existitzen dela? Adibide moduan honako predikatua har dezagun:

$$\mathbf{f}(\mathbf{x}) = \begin{cases} \text{true} & \text{Fidelen bizarrak zehazki } 10.253 \text{ ile ditu} \\ \text{false} & \text{bestela} \end{cases}$$

Ebaluatu beharreko predikatua absurdua dela eman dezake, bere balioa zehazteko zailtasun handiak daudelako. Fidel ezagutzen dugun, bere bizarra ukitzen uzten digun, edo bizkartzaina ote duenaren menpe dago. Bere bizarrera iristeko inolako mugarik ez dugula suposatuz, bere ileak kontatuko dizkiogun une zehatzaren inguruan ere zailtasunak daude: berriren bat ahaztea gerta daiteke, edo baten bat erori, bizarra kentzea erabaki dezake, ustekabean hil eta bere gorpua erraustua izan daiteke. Arrazoizkoa da pentsatzea Fidelen bizarrak zenbat ile dauzkan ez dugula inoiz jakingo (CIA-k horri buruz ezkutuko txostenen bat ez badauka behintzat), eta beraz, 10.253 direnentz ez dugula jakingo, eta ondorioz, **f** predikatua ezingo dugula inoiz konputatu.

Baina, zenbakia jakitea egiazki behar al dugu? Eraiki beharko dugun programaren egituran pentsa dezagun. Programa horrek itzuli beharreko balioa zeinen menpekoa da? Funtzioaren definizioan arreta gehiago jarriz gero, irteerak **x** sarrerarekin inolako zerikusirik ez duela ikusten dugu, hots, predikatuaren ebaluazioan bi aukera bakarrik daudela: Fidelen bizarrak 10.253 ile ditu (eta **f**-k edozein **x**-rako *true* emaitza itzultzen du) ala ez ditu (eta **f**-k edozein **x**-rako *false* emaitza itzultzen du). Hau da, **f** predikatua beti egiazkoa da ala beti faltsua (beste aukerarik ez dago). Hortaz, **f** funtzioa kalkulatzeko bi programa hautagai ditugu:

Ala

```
X0 := true;
```

ala

```
X0 := false;
```

Predikatua konputatzen duena bietatik zehazki zein den ezin dugu erabaki, baina bietako batek egiten duenaren ziurtasun osoa dugu. Honek **f** konputatzen duen programa bat existitzen dela frogatzen du, eta beraz erabakigarria dela. Erabili dugun argumentua polizi nobela batzuetan erakutsitakoaren antzekoa da: hiltzailea hilketaren unean etxean zegoenetako bat dela ziur dakigu, baina ez zehazki zein den, alabaina etxeko biztanle horiek zeintzuk diren jakina denez, hiltzailea gizonezkoa dela edo bere helburua lapurreta ez zela ondoriozta dezakegu.

Era berean, gure kasuan **f** predikatuaren definizioa ez dago nahiko doituta zehazki zein funtzio den jakin dezagun, baina konputagarria izan behar dela zehazteko adina bai.

Funtzioa konstantea dela deduzitzea ez da, programa idatzi beharrik gabe, bere konputagarritasuna ondorioztatzeko metodo bakarra. Bedi, adibidez, honako predikatua:

$$\mathbf{g(x)} = \begin{cases} \text{true} & \text{Fidelen bizarrak zehazki } \mathbf{x} \text{ ile ditu} \\ \text{false} & \text{bestela} \end{cases}$$

Noski, orain ezin dugu **g** konstantea dela esan, kasuetako batean (**x**-ren menpekoa) egiazkoa delako eta gainontzekoetan faltsua. Fidelek bizarrean zenbat ile dituen jakitera behartuta al gaude orain? **g** konputatzen duen programa zehatza idatzi nahiko bagenu, hori derrigorrezkoa litzateke. Alabaina, hori ez da gure asmoa. Ez ahaztu programa mahaiaren gainean jartzea ez dugula behar: bere existentzia frogatzea nahikoa izango da.

Nolakoa da **g** orain, eta programak zer egin beharko luke bera konputatzeko? Aipatu dugun bezala, programak *true* kasu bakar batean itzuli behar du (sarrerak Fidelen bizarraren ile kopuruarekin bat egiten duenean) eta *false* gainontzeko guztietan. Fidelen bizarraren M ile kopuruaren berri bagenu, honako makroprograma idazterik edukiko genuke:

```
X0 := X1 = M;
```

Baina, berriz ere, programaren existentzia frogatzeko M-ren balioa hutsala, gutxienekoa, da. Itxura horretako programa guztiekin, hots, M-ren balizko balio bakoitzeko programa bat hartuz, zerrenda presta dezakegu:

```
X0 := X1 = 0;
```

ala

```
X0 := X1 = 1;
```

ala

```
X0 := X1 = 2;
```

ala

```
X0 := X1 = 10.253;
```

Ona zein zen erabaki ezin genuelarik, **f** predikatuarentzat bi programa (edo beraien baliokideak) genituela esaten genuen bezala, **g**-rentzat infinitu hautagai ditugu, **M**-ren balio bakoitzeko bana, eta zehatz bat aukeratzea ezinezkoa zaigu. Baina programa horietako batek **g** konputatzen duenaren ziurtasun osoa badaukagu eta ondorioz **g** ere konputagarria badela.

Pixka bat bihurriagoa den azken adibide bat azter dezagun. Honako multzoa defini dezakegu: **A** = {**z**: existitu daiteke izaki bizidunen bat zeinaren genomaren luzera zehazki **z** den}. Azaleko azterketa batekin ere definizio honek alderdi korapilatsuak biltzen dituela antzeman daiteke.

Alde batetik, izaki bizidun gutxi batzuen genomen luzera zehatzak bakarrik ezagutzen ditugu. Bestetik, espezie ezezagun asko dago eta egunero beste batzuk galtzen dira, eta zer esanik ez, eboluzioak oraindik eman ez dituenak, ingeniaritza genetikoak sortu ez dituenak edo, besterik gabe, "existitu daitezkeenak" baina inoiz existituko ez direnak. Hori guztia kontuan hartuz honako predikatua defini dezakegu:

$$\mathbf{h}(\mathbf{x}) = \begin{cases} \text{true} & \exists \mathbf{y}\ (\mathbf{x} + \mathbf{y}) \in \mathbf{A} \\ \text{false} & \text{bestela} \end{cases}$$

Eta gure buruari galde diezaiokegu, konputagarria al da, bai ala ez? Baietz frogatzeko genetikari buruzko inolako jakintzarik edukitzea ez dugula behar ikusiko dugu. Argi dago **h(x)**-ren balioa *true* izango dela bere genomaren luzera **x**-ren berdina edo handiagoa duen izakiren bat existitu litekeenean. Beraz, **h(x)** betetzen denean **x**-ren azpiko sarreretarako ere **h**-ren emaitza egiazkoa izango da.

Hasiera batean, **A** multzoa infinitua izatearen aukera badago (hau da, bizitzeko gai diren izakien genomen luzeraren inguruan goi-mugarik ez egotearen aukera). Kasu horretan **h** funtzioa *true* funtzio konstantea izango zela edukiko genuke, edozein **x** hartuta ere, **A** multzoan **x** hori baino handiagoko **z** balioak egongo liratekeelako.

Baina **A** finitua izan liteke: agian genomarako **L** luzera maximoa egon daiteke, inolako izaki bizidunik gainditu ezin duena. Kasu horretan **h** predikatua **L**-ren berdinak edo txikiagoak diren balioetarako egiazkoa izango litzateke, eta gainontzekoetarako faltsua. Eta ez dago beste aukerarik (**A** finitua ala infinitua izan daiteke bakarrik), ondorioz honako programetatik baten batek nahitaez **h** konputatu behar du:

```
X0 := true;
```

ala

```
X0 := X1 <= 0;
```

ala

```
X0 := X1 <= 1;
```

ala

```
X0 := X1 <= 100.893.873.343.157.2245.234.388.253;
```

Aurreko kasuetan bezala, zein den erabakitzeko modurik ez daukagun arren (agian Biologia Teorikoaren arloan egindako ikerketek etorkizunean hori argitu ezean).

Funtzio while-konputagarrien artean ikusi dugu badaudela batzuk ezin zaiena dagokien programa aurkitu. Orduan, beraien ***konputagarritasuna eskuragaitza*** dela esaten dugu, programa zehatza eraikitzea, hots, eskuratzea, posible dutenetatik bereizteko. Lehenengoen interes praktikoa mugatua den arren, horrelako funtzioak existitzeak konputaezintasunaren kontzeptuaren inguruan sakonkiago hausnartzeko bidea zabaltzen digu. Kontzeptu horren izaera absolutua dela, hots, problemaren espezifikazioan jarritako zehaztasuna bezalako faktoreen menpe ez dagoela hausnar dezakegu. Aitzitik, funtzio konputaezinak argi eta garbi espezifikatzen direla, hots, beraien esanahiari edo interpretazioari buruz zalantzarik ez dagoela ikusiko dugu.

3.2 Church-Turing-en Tesia

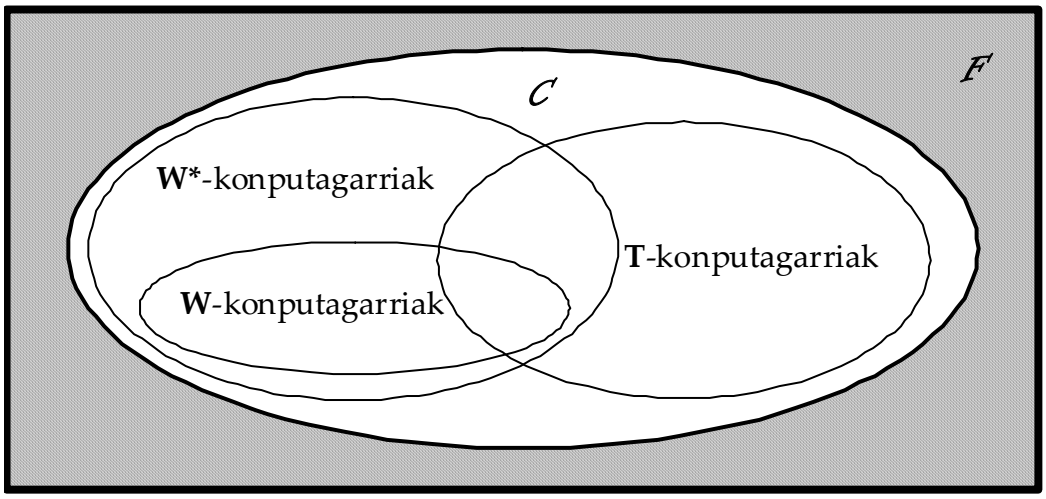
Beste aztergai garrantzitsua aukeratutako programazio-lengoaiaren egokitasunarena da: while-programek zein mailataraino osatzen dute lengoaia adierazkor bat? Zenbait problema while-konputagarriak direla ikusi dugu eta laster while-konputaezinak diren beste batzuk aurkitzen hasiko gara. Alabaina, zer gertatuko litzateke beste programazio-lengoaia desberdin bat, esaterako, ahaltsuagoren bat, erabiltzen hasiko bagina? Primitiba gehiago gehituz gure lengoaia zabaltzen badugu, lengoaiaren bertsio sinplerako konputagarriak ez ziren funtzio batzuk konputagarriak bihurtzen direla ez al genuke aurkituko agian?

Gai hau guztiz funtsezkoa da, berez, guretzat while-programek ez baitute inolako interesik. While-konputagarritasunaren kontzeptua lengoaia honekin bakarrik aplika daitekeela aurkituko bagenu, gure emaitza edo ondorio guztiek ez lukete inolako interesik edukiko, while-programak ez baititugu praktikan erabiliko. Aitzitik, problema bakoitzerako zein lengoaia zehatzetan izango den programagarria jakiteak garrantziren bat eduki dezakeen arren, konputaezintasunak kontzeptu absolutu bezala du interesa: funtzio bati konputaezina esan beharko genioke bakarrik bere balioak kalkulatzeko prozedurarik ezartzea ezinezkoa balitz, edozein metodo, lengoaia edo sistema konputazional erabilita ere. Problema zehatz bat **S** sisteman konputagarria ez den arren, aldiz, **T**-n hala dela frogatzera mugatzeak ez dauka zentzu gehiegirik. Horrek, gaitasun adierazkor gutxiago daukalako, **S** sistema kalitate okerragokoa dela eta agian baztertu, utzi, egin beharko genukeela besterik ez digu adieraziko.

Edozein programazio-sistemari bere bidez konputagarriak diren funtzioen klasea dagokio. Ikur-kateen gaineko funtzio guztien klaseari $\mathcal{F}$ deitzen badiogu, **S** programazio-sistema bakoitzak **S**-konputagarriak diren funtzioak **S-**konputaezinak direnetatik bereizteko marra bat ezartzen du. Konputagarritasunaren nozio orokorra programazio-sistema zehatzaren menpekoa

ezin denez izan, funtzio bat besterik gabe ***konputagarria*** dela esango dugu, **S**-konputagarrian bihurtuko duen **S** programazio-sistema existitzen bada. Funtzio konputagarri guztien klaseari $\mathcal{C}$ deituko diogu, eta beste klaseekiko erlazioa 3.1 irudiko diagraman adierazten dugu.

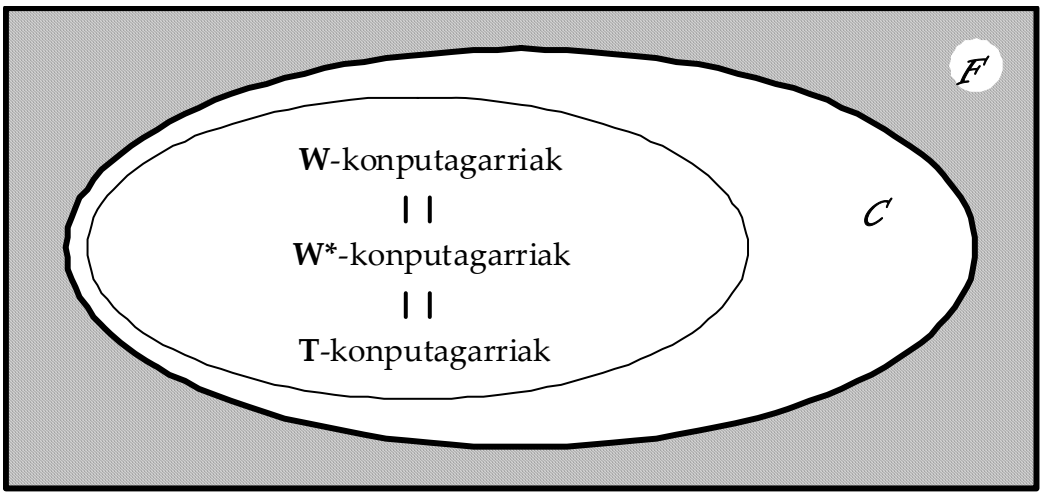
Adierazitako hori egoera erreala izango balitz arazo larria edukiko genuke: funtzioa konputagarria ez dela frogatzeko, existitzen diren programazio-sistema guztiak banan-banan probatu beharko genituzke, funtzio horrekin ezin dutela egiaztatzeko. Zorionez 3.1 irudian erakutsitako panorama ez dator bat errealitatearekin eta programazio-lengoaiak beraien artean oso antzekoak dira, susma genezakeena baino askoz gehiago. Adibidez, [IIS 98]-n makroekin egin dugun lan guztiak, while-programen lengoaian erraztasunen baten hutsunea (hala nola, kabiatze-adierazpenak, aurreprogramaturiko azpierrutina gehiago, datu-mota gehiago, egitura konplexuagoak, erabiltzaile-objektuak, etab.) aurkitu dugun bakoitzean hori lengoaiari gehitzea ez zela beharrezkoa erakutsi digu. Eta ez zen beharrezkoa lengoaian jadanik definituta zeuden elementuen bitartez simulatu zitekeelako. Beste era batera esanda, while-programek desiratu beharko luketen eta beste programazio-lengoaiaren batek daukan ezaugarririk ez dugu topatu.



3.1 irudia: **W** programazio-sistema edukiko bagenu, zeinarekin **W**-konputagarriak diren funtzioen klasea konputatzera iritsiko litzatekeen, sistemaren bertsio hobetua **W*** lortzea gerta liteke eta berarekin, lehen aipatutako klasea **W**-rentzat konputaezinak ziren funtzioekin zabaltzea. Irudian **T** deitu duguna bezalako sistema independenteak gara litezke, funtzio berriak konputatzea ahalbidetu lezaketenak, baina **W**-konputagarriak diren funtzio batzuk programatzea ezinezkoa egingo luketenak. Funtzio konputagarrien $\mathcal{C}$ klasea (lodiagoa den marra) sistema partikular guztien bitartez konputatutako guztien bildura izango litzateke. Eta konputaezinak, balizko edozein metodo hartuta ere ebazpenik onartuko ez luketenak izango lirateke (eremu grisa).

Alabaina, honen inguruan eduki dezakegun gure esperientzia oso argigarria izan arren, Konputazio Zientziek dituzten ia 70 urteetan zehar sortutako erabateko unanimitate sendoa are gehiago da: beraien artean oso urruti dauden ereduak eraiki

diren arren (horietako batzuk esoterismoaren ingurukoak ere esan dezakegu), while-programak baino perfektuagoa den programazio-lengoaiarik ez da sekula lortu. Oraindik adierazgarriagoa dena, perfekzio gutxiagoko ereduak ere ez dira eman: orain arte asmatutako **S** sistema guztiek **S**-konputagarriak diren funtzioen klase bera sortu dute zehatz-mehatz[3]. Egoera 3.2 irudian erakutsitakoa izango litzateke.



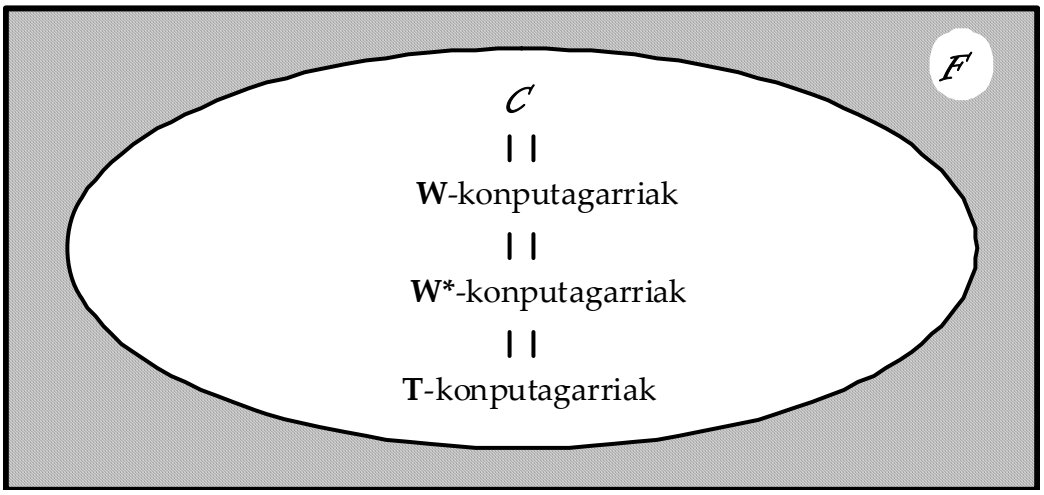
3.2 irudia: Orain arte aurkitutako programazio-sistema guztiek, eta beraien aldakuntzek, funtzio berdin-berdinak konputatzen dituzte. Hala ere, gaur egun konputaezinak suertatzen diren funtzioetako batzuk konputatzea ahalbidetuko luketen programazio-sistema berriak agertzeko atea irekita gera zitekeen. Hori horrela balitz, egungo sistemek lortu ez duten funtzio konputagarrien eremu bat egongo litzateke.

Jokaleku honetan zenbait gauza hobetu dira baina beste batzuk ez. Alde batetik, funtzio bat while-konputagarria ez dela frogatzen badugu, aldi berean ezaguna den beste edozein lengoaiarekin ere ezin dela konputatu frogatzen ariko ginateke, eta hori ez da gutxi. Alabaina, "benetan" konputaezina dela egiaztatzeko inolako modurik ez genuke izango, horretarako oraindik asmatu ez diren, eta agian inoiz izango ez diren, programazio-sistemekin duten erlazioa aztertu beharko genukeelako.

Hala ere, programazio-sistema guztiak behin eta berriz bat etortzeak, konputagarritasunaren nozio berdina sortuz, informatikari teorikoak ondorio zorrotz batera eraman zituen oso goiztik: konputagarritasunaren nozio absolutua existitzen diren sistema konputazionaletatik baino haratago ez da joango, hau da, ezagunak direnak baino hobea den programazio-sistemarik ez da existitzen. Eta hori izango litzateke, hain zuzen ere, Informatikan burutu diren berrikuntzetatik bakar

[3] Baieztapen honi ñabardurak egin behar dizkiogu. Programazio-lengoaia behar adina murrizten bada, zenbait funtzio konputatzeko gaitasuna gal daiteke. Esaterako, while-programekin egin genuenaren antzera, baina begizta orokortuak for motakoak diren begizta kontrolatuengatik ordezkatuz, *for-programak* definitzen baditugu, lengoaia berriak gaitasun adierazkorra galtzen du eta jatorrizko lengoaiarekin baino funtzio gutxiago konputatzen dira.

batek ere hasierako konputazio eredu abstraktuek konputatzen ez zituzten funtzioak konputatzea ez lortu izanaren arrazoia. Teoria hau ***Church-Turing-en tesia*** bezala ezaguna da eta era zabalean onartutzat hartzen da. Badira urteak Konputazio Zientziatan hori horrela dela inolako erreserbarik gabe kontuan hartuz lan egiten dela. Bere ondorioak 3.3 irudian adierazten dira.



3.3 irudia: *Church-Turing-en Tesiaren* ondorioa: programazio-sistema desberdinetarako lortutako konputagarritasun-nozioa erabatekoa da, eta etorkizunean inolako sistemarik ez du aldatuko.

Church-Turing-en Tesiaren berehalako ondorioa da while-konputagarritasun eta while-konputaezintasunaren nozioak egungo eta etorkizuneko programazio-sistema guztietarako ere baliozkoak direla. Hortaz, funtzio jakin bat konputatzen duen while-programarik ez dagoela frogatzea lortzen dugunean, askoz sakonagoa den zerbait frogatzen arituko gara: Informatikak funtzio hori konputatzeko bitartekorik ez daukala, ez orain ezta inoiz ere. Lege Unibertsalaren mailako muga aurkitua izango dugu.

Konputagarritasuna, konputaezintasuna, erabakigarritasuna edo erabakiezintasuna hitzen aurretik hemendik aurrera *while* aurrizkia ez erabiltzearen arrazoia arestian aipatutako horixe izango da, hain zuzen ere. Funtzio bat konputagarria dela frogatzeko bera konputatuko duen while-programa aurkituz jarraituko dugu, edo konputaezina dela frogatzeko inolako while-programarik ez duela horrelakorik egiten egiaztatuz. Hala ere, badakigu frogapen horien ondorioak euskarria besterik ez den erabilitako lengoaia baino haratago doazela, eta Konputazioaren Legeen barnean funtzio horrek duen estatutuari eragiten diotela.

4. Funtzio Unibertsala

Konputagarritasun eta konputaezintasunaren nozioen hedapena argitzen duten zenbait funtsezko kontzeptu ezarri ondoren, lehenengo funtzio konputaezinen bilaketari ekiteko prest gaude. Alabaina, hurrengo kapituluan ikusiko dugun lez, lan hori ez da batere arrunta eta arrakastaz praktikan jartzeko kasu batzuetan tresna gehigarriak beharko ditugu. Konputaezintasun frogapenen ezaugarri bitxienetako bat programen eraikuntzan oinarritzen direla da, askotan nahiko konplexuak diren programak eraiki behar izaten direlarik.

Funtzio konputagarri gehiago aztertzeko beste arrazoi bat konputagarritasunaren azken muga arakatzearen aukeran datza. Kapitulu honetan landuko ditugun funtzioak eta predikatuak, nolabait konputagarritasunaren barnean plantea ditzakegun bihurrienak dira. Pixka bat urrutirago joaten saiatzen bagara, lehenengo konputaezinekin aurkituko gara (ikus 5. kapitulua).

Badago datu-mota bat, zeinen interesa aurretik ezaguna izan dezakegun beste edozeinena baino haratago doan, $\mathbb{W}$ datu-motari buruz ari gara. Karaktere-kate, balio logiko, zenbaki edo, pilak edo zuhaitzak bezalako egituren kasuan era praktikoan planteatzen zaizkigun problema gehien-gehienak konputagarriak dira eta aspaldidanik (zenbait kasutan zibilizazio egiptiarretik) ebatzita daude, baina programak (eta bereziki, while-programak) askoz konplexuagoak diren objektuak dira eta hauetan konputaezintasuna ia-ia agerian geratzen da.

Beste inplementazio batzuekin gertatu den bezala, programen gainean definitutako eragiketa zehatzen konputagarritasuna plantea dezakegu eta ondoren eragiketa horiek (hobeto esanda, horien artetik konputagarriak direnak) beste problema batzuk ebazteko tresna lagungarri bezala erabil ditzakegu. Era berean programa-multzoak definitu eta gure definiziotan erabil ditzakegu:

$$\mathrm{VAC} = \{ \mathbf{x}\in\mathbb{W}: \forall \mathbf{y}\in\sum^* \varphi_{\mathbf{x}}(\mathbf{y})\uparrow \} = \{ \mathbf{x}\in\mathbb{W}: \mathbf{W}_{\mathbf{x}} = \varnothing \}$$

$$\mathrm{COF} = \{ \mathbf{x}\in\mathbb{W}: \mathbf{W}_{\mathbf{x}} \text{ kofinitua da} \} = \{ \mathbf{x}\in\mathbb{W}: \overline{\mathbf{W}_{\mathbf{x}}} \text{ finitua da} \}$$

$$\mathrm{SUP} = \{ \mathbf{x}\in\mathbb{W}: \varphi_{\mathbf{x}} \text{ supraiektiboa da} \} = \{ \mathbf{x}\in\mathbb{W}: \mathbf{R}_{\mathbf{x}} = \sum^* \}$$

$$\mathrm{INJ} = \{ \mathbf{x}\in\mathbb{W}: \varphi_{\mathbf{x}} \text{ injektiboa da} \} = \{ \mathbf{x}\in\mathbb{W}: \neg\exists \mathbf{y}\exists \mathbf{z}\ (\mathbf{y}\neq\mathbf{z} \wedge \varphi_{\mathbf{x}}(\mathbf{y})=\varphi_{\mathbf{x}}(\mathbf{z})) \}$$

$$\mathrm{K} = \{ \mathbf{x}\in\mathbb{W}: \varphi_{\mathbf{x}}(\mathbf{x})\downarrow \}$$

SIC = { $\mathbf{x} \in \mathbb{W}$: **x**–n ez dago if_then erako azpiprogramarik }

CER = { $\mathbf{x} \in \mathbb{W}$: **x**-ren egikaritzapenik ez dago X0-ren edukia erabiltzen duenik }

Horrela, VAC multzoa sarrera guztiekin begizta baten barnean geratzen diren programen multzoa da, COF ia datu guztietarako konbergitzen dutenena da (hots, portaera dibergenteen kopurua finitua dutenena), SUP edozein emaitza onartzen dutenena da, eta INJ datu desberdinetarako emaitzak errepikatzen ez dituztenen multzoa da. Era berean, K bere kodearen gainean konbergitzen duten programen multzoa da, SIC bere testuan baldintzazko agindurik ez daukatenena da eta CER multzoa X0-ren edukia erabiltzen ez dutenena da (hots, bere eskuin aldean X0 aldagaia daukan esleipenen batetik pasatzen ez direnena). Ikusiko dugun bezala, programa-multzo hauek guztiak (bat ezik) erabakiezinak dira.

Azaldu ditugun arrazoi horiek direla eta, datozen ataletan ahalegina egingo dugu while-programak datuak bezala hartzen dituen programazioaren inguruan trebetasuna garatzeko. Horrela, zenbait funtzio konputagarri eta predikatu erabakigarri lortuko dugu ere, beste funtzio eta predikatu batzuk hala ez direla frogatzeko laguntza handikoak izango direnak hain zuzen ere.

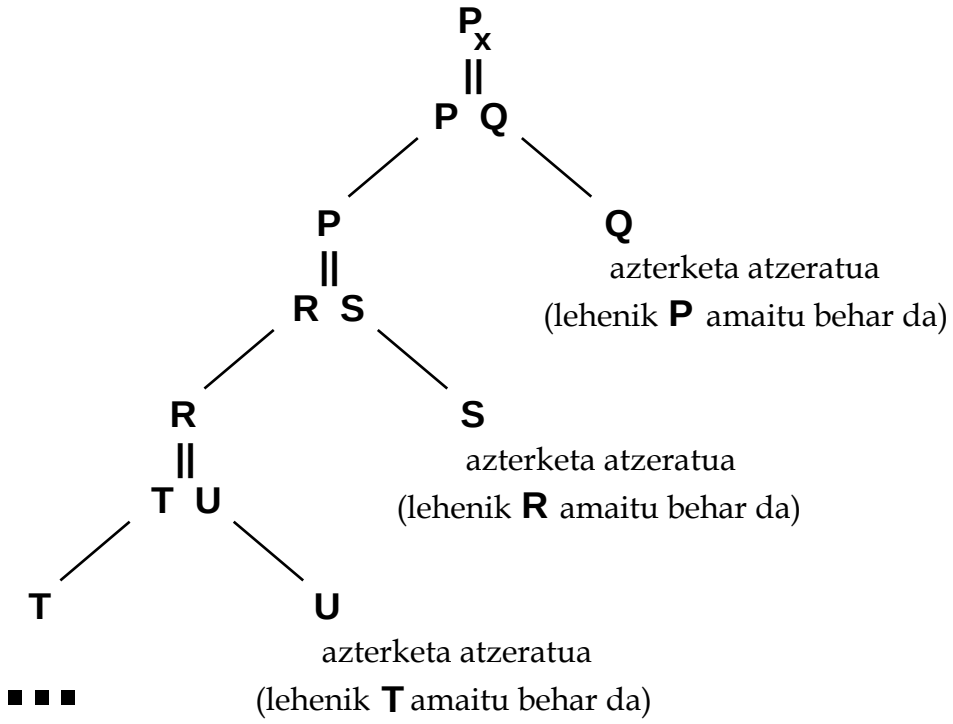
4.1 While-programen gaineko eragiketa sintaktikoak

While-programak beste edozein datu arrunt bezala erabiltzea planteatzen badugu, defini dezakegun maneiatze-maila errazena sintaktikoarena da: programa baten testua azter dezakegu, hots, bere eragiketak edo egitura arakatu, eta horren inguruan galdera errazak ebatzi ditzakegu. Maila honetan, oro har, argumentu bezala while-programa bat hartu eta emaitza sinple bat (zenbakizkoa edo boolearra) itzultzen duten funtzioak aurkituko ditugu.

Adibidez, honako erantzunak ematen dituzten funtzioak defini ditzakegu: programa batek dituen aldagaien edo aginduen kopurua, habiaratuta dauzkan begizten kopuru maximoa, ezaugarri bereziak dituen egituraren bat (esaterako, bere barnean while-aren kontrol aldagaia aldatzen ez duen begizta susmagarriren bat) ba ote duen, etab. Programen ***ezaugarri estatikoekin***, hots, konpilatze-unean zehaztuak izan daitezkeenekin, bakarrik lan egiten duten funtzioen konputagarritasuna frogatzen joatea ez da zaila. Hori ikusteko honako funtzioa adibide moduan hartuko dugu:

$$\text{azken\_aldagaia}: \mathbb{W} \to \mathbb{N}$$

edozein programa harturik, bertan agertzen den aldagai-indize handiena adierazten duena. Bere konputagarritasuna frogatzeko makroprograma bat idatziko dugu. Honek, *azken_aldagaia*(**x**) kalkulatzeko, $\mathbb{W}$ datu-motaren berezko eragiketak erabiliko ditu eta $\mathbf{P_x}$ programa arakatuko du, bere oinarrizko aginduen (esleipenen eta baldintzen) bila, aldagaiak bertan erreferentziatzen baitira. Analisi-prozesu hau kasu gehienetan lineala da. Adibidez, $\mathbf{P_x}$ esleipena bada, bere aldagaien indizeak atera eta handienarekin geratzea nahikoa izango da. Baldintza edo iterazio agindua bada, baldintzan parte hartzen duen aldagaia aztertuko dugu, barneko agindua atera eta prozesua honekin jarraituko dugu. Alabaina, $\mathbf{P_x}$ agindua **P Q** motako konposaketa sekuentziala bada, lehenik **P** eta gero **Q** (edo alderantziz) arakatu behar ditugula aurkitzen dugu. Baina **P**, era berean, **R S** motako konposaketa sekuentziala izan daitekeenez, bere analisiak **R** aztertzen dugun bitartean **S** baztertuta uztea eska dezake. Jakina, hau amaierarik gabeko egoera izan daiteke, beraz $\mathbf{P_x}$-ren azpiprograma diren eta aztertzeko zain dauden aginduen kopuru mugagabea eduki dezakegula aurreikusi behar dugu (ikus 4.1 irudia)



4.1 irudia: Zuhaitz egitura duen while-programa bat analizatzean, azterketaren zain dauden azpiprogramen zerrenda mantentzea beharrezkoa izan daiteke. Zerrenda horren luzera arbitrarioa izango da.

Arazo hau konpontzearren oraindik analizatu gabe dauden while-programak dituen pila bat mantenduko dugu une oro. Pila honek hasiera batean $\mathbf{P_x}$ programa edukiko du eta begizta batek kudeatuko du. Honek, iterazio bakoitzean, pilaren

tontorrean aurkitutako while-programa prozesatuko du eta bere azpiprogramak pilan berriro sartuko ditu, ondoren eta komeni denean prozesa daitezen. Pila husteak lanarekin amaitu dela adieraziko du.

While-programak maneiatzeko datu-motaren eragiketak erabiliko ditugu, 3. Eranskinaren ***e*** atalean deskribatuta daudenak hain zuzen ere.

```
PILA :=  <];
PILA :=  pilaratu (X1, PILA);
X0 :=  0;
-- X0-n orain arte aurkitutako indize maximoa mantenduko dugu
while not  P_hutsa? (PILA)  loop
-- Une honetan analizatu beharreko azpiprograma pilatik ateratzen da
      PROG :=  tontorra (PILA);
      PILA :=  despilatu (PILA);
-- Azpiprograma hori bere motaren arabera prozesatzen da
      if  esleipen_hutsa? (PROG)  then
            if aldagai_indize (PROG) > X0  then
                  X0 := aldagai_indize (PROG);
            end if;
      elsif esleipen_cons? (PROG) or esleipen_cdr? (PROG)  then
            if aldagai_indize (PROG) > X0  then
                  X0 := aldagai_indize (PROG);
            end if;
            if 2_aldagai_indize (PROG) > X0  then
                  X0 := 2_aldagai_indize (PROG);
            end if;
      elsif baldintzazkoa? (PROG) or iterazioa? (PROG)  then
            if aldagai_indize (PROG) > X0  then
                  X0 := aldagai_indize (PROG);
            end if;
            PILA :=  pilaratu (barne_agindu (PROG), PILA);
      else  PILA := pilaratu (2_ barne_agindu (PROG), PILA);
            PILA := pilaratu (barne_agindu (PROG), PILA);
      end if;
end loop;
```

4.2 Zerrendatze Teorema

While-programen beste ezaugarri estatiko batzuk ere antzeko eran tratatzen dira. Alabaina, programen *portaera dinamikoaren* menpekoak diren, hots, egikaritzapen-unean egiaztatzen diren egoeren menpekoak diren beste ezaugarri batzuetan murgiltzen garenean emaitza interesgarriagoak lortzen dira.

Horrela, ikusiko dugun bezala konputagarria izango den funtzio berezi batean jarriko dugu gure arreta: ***funtzio unibertsala***. Honek argumentu bezala $\mathbf{P_x}$ while-

programa eta **y** datua (hitza) hartzen ditu, eta aipatu datu hori sarrera bezala emanda $\mathbf{P_x}$ programak sortuko lukeen emaitzarekin erlazionatzen du bikotea. Beraz, funtzio unibertsala edozein konputazioren portaeraren emaitza deskribatzeko gai izango da, eta horregatik bere hantusteko izena. Funtzioa Φ hizkiaren bitartez adierazten da eta honela definitzen da:

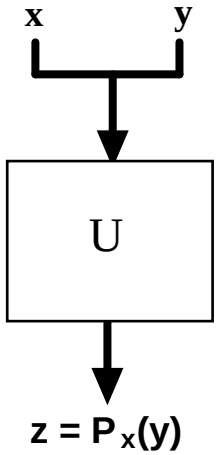
$$\Phi(\mathbf{x},\mathbf{y}) \cong \varphi_{\mathbf{x}}(\mathbf{y})$$

Funtzio honen garrantzia azpimarratu behar dugu, sarrera bezala, kalkulatu beharreko funtzioaren indizea eta sarrera-datua besterik eman gabe, argumentu bakarreko *edozein* funtzio konputagarriren portaera simulatzen duelako. Funtzio unibertsala konputatzen duen programa, nolabait, gainontzeko programa guztiak duintasunez errepresentatzeko gai da, bere barnean konputazioaren mamia bilduz.

TEOREMA (ZERRENDATZEARENA): $\Phi : \mathbb{W} \times \sum^* \longrightarrow \sum^*$ funtzioa konputagarria da:

$$\Phi(\mathbf{x},\mathbf{y}) \cong \begin{cases} \varphi_{\mathbf{x}}(\mathbf{y}) & \varphi_{\mathbf{x}}(\mathbf{y}) \downarrow \\ \perp & \text{bestela} \end{cases}$$

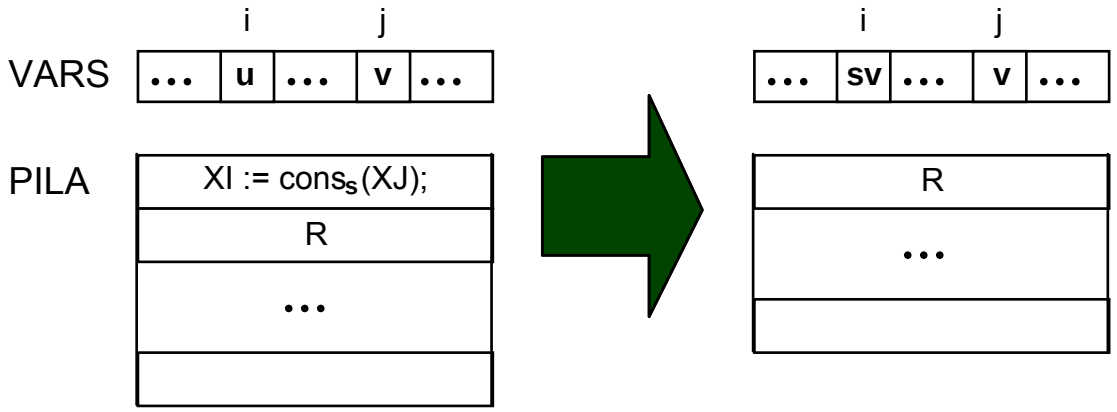
$\boxtimes$ Φ konputagarria dela frogatzeko idatzi behar dugun **U** programak (***programa unibertsalak***) sarrera bezala edozein **x** programa-kodea eta **y** datua hartuko ditu, eta emaitza bezala $\mathbf{P_x}$-k **y**-ren gainean sortuko lukeen irteera (egonez gero) lortu beharko du (ikus 4.2 irudia).



4.2 irudia: Diseinatu behar dugun **U** programaren funtzionamenduaren eskema.

U programak $\mathbf{P_x}$-k **y** datuaren gainean daukan portaera (hots, bere konputazioa) simulatzea izango da hau egiteko era naturala. Beraz, **U** while-programen *interpretatzailea* izango da: egokia den egoera-bektorea hasieratu eta mantendu beharko du, $\mathbf{P_x}$ programaren aginduek egoeran burutzen dituzten eraldaketak kontu handiz jarraituz. Hala ere, **U** idazteko unean egoera-bektore

horrek izango duen tamaina ez dakigu, zein $\mathbf{P_x}$ programa zehatza simulatu beharko dugun ez baitakigu. Egitez, egoera-bektore horrentzat ezin dugu ezta **k** tamaina maximoa finkatu, espazio gehiago behar duen eta aurreikuspen horiek hausten dituen programaren bat beti existituko delako. Irtenbidea egoera-bektore osoa edukiko duen VARS bektore dinamikoa erabiltzea da, eta honela une oro VARS(**i**) balioa $\mathbf{P_x}$ programaren XI aldagaiak daukan edukia izango da. VARS-en hasieratzea, lehenengoan ezik gainontzeko bere posizio guztietan ε balioa jarriz egingo da, eta VARS(1)-ek **y** datua eduki beharko du. Era berean, behin $\mathbf{P_x}$-ren egikaritzapenaren simulazioa bukatzen denean, bere emaitza bektorearen 0-garren posizioan bilatzera joko dugu.

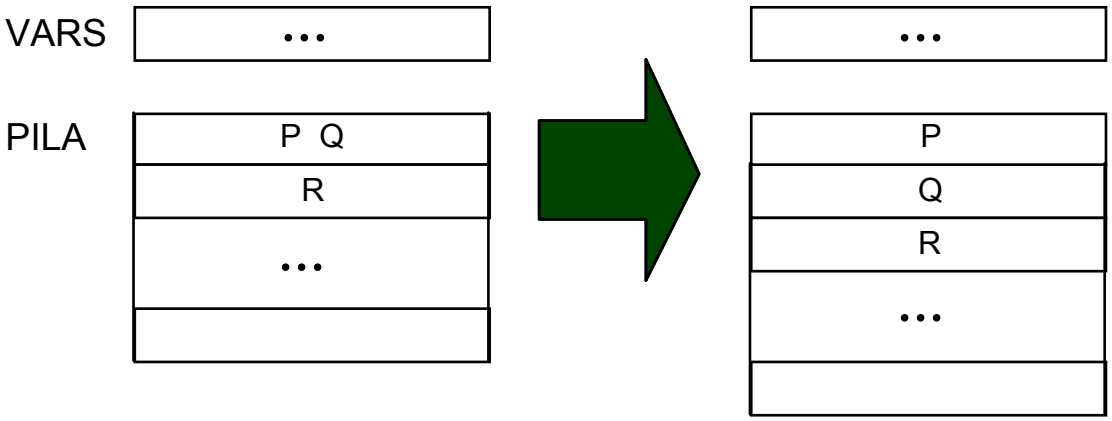


4.3 irudia: **U**-ren egikaritzapen-zikloa, egikaritu beharreko hurrengo programa esleipena denean.

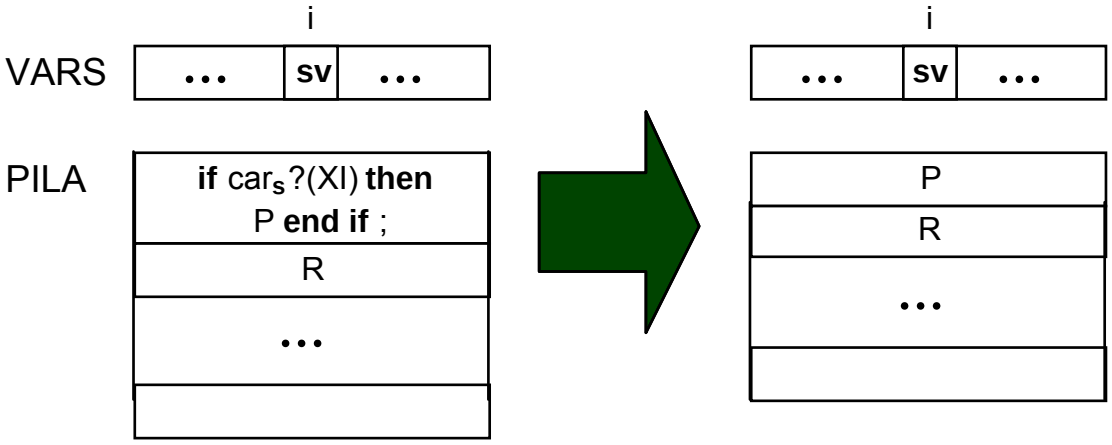
Bestaldetik, $\mathbf{P_x}$-ren egikaritzapenak berak *azken_aldagaia* funtzioarekin genuenaren (hots, 4.1 irudian deskribatzen genuenaren) antzeko arazoa agertzen digu. Modu berean ebatziko dugu, baina abagune honetan pilak, egikaritzeko zain dauden programak edukiko ditu, oraindik analizatu gabe daudenen ordez. Ondoren zuzena izango den egikaritzapen-ordena sor dadin, zain dauden programak pilan sartzean kontuz ibili beharko dugu, besterik ez. Hasiera batean PILA-k $\mathbf{P_x}$ programa bakarrik edukiko du, eta egikaritzapenaren oinarrizko zikloa PILA-tik programa bat atera eta hau egikaritzen saiatzea izango da. Esleipena bada, ondorioz VARS konputazio-egoeran aldaketa egingo da (4.3 irudia).

Konposaketa sekuentziala izanez gero, programa sinpleagoak izango diren bere bi osagaietan zatitu eta pilan sartzearekin konformatuko gara (4.4 irudia). Baldintzazko konposaketa bada, baldintza VARS konputazio-egoeran ebaluatuko dugu eta **if**-aren barneko agindua egikaritzea egokitzen bada, hori pilan sartuko dugu (4.5 irudia) hurrengo iterazioan egikaritua izan dadin. Azkenik, iteraziozko programa bada, bere baldintza ebaluatuko dugu ere, baina kasu honetan begiztan sartzea egokitzen bada, pilan bere barneko agindua ez ezik begizta osoaren kopia

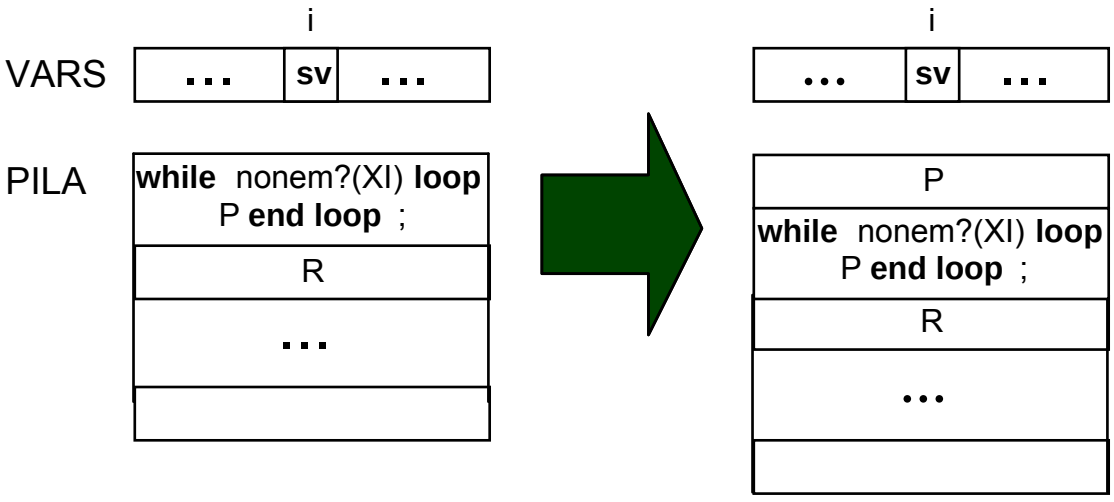
ere sartuko dugu, uneko iterazioa bukatu ondoren berriz kontuan hartua izan dadin (4.6 irudia).



4.4 irudia: U-ren egikaritzapen-zikloa, egikaritu beharreko hurrengo programa konposaketa sekuentziala denean.



4.5 irudia: U-ren egikaritzapen-zikloa, egikaritu beharreko hurrengo programa baldintzazko konposaketa denean, bere baldintza betetzen delarik.



4.6 irudia: U-ren egikaritzapen-zikloa, egikaritu beharreko hurrengo programa iteraziozkoa denean, eta gainera bere baldintza betetzen delarik: orduan gutxienez bere lehen iterazioa burutuko da.

Lehen itxura batean *azken_aldagaia* eta Φ funtzioen programek oinarri berdinak jarraitzen dituztela eta antzeko portaerak dituztela eman dezake, baina beraien artean badago oso diferentzia funtsezkoa, pilaren erabilerari eragiten diona. Lehenengoa analisi-prozesu bati dagokio, eta pilatik ateratzen den programa ez da han berriro sartuko (seguraski bere pusketak sartu beharko diren arren). Alabaina **U** simulazio-prozesu bati dagokio eta begizten egikaritzapenak programa bera behin eta berriz pilaratzea exiji dezake.

Aurreko paragrafo eta irudietan zehaztutako estrategiak honako **U** programara garamatza:

```
VARS(azken_aldagaia (X1)) := ε;
VARS(1) := X2;
-- y kargatu baino lehen, Px-ren egoera-bektorearen aldagai guztiak ε-ekin hasieratzen ditugu
PILA := pilaratu (X1, <]);
while not P_hutsa? (PILA) loop
    PROG := tontorra (PILA); PILA := despilatu (PILA);
    if esleipen_hutsa? (PROG) then
        VARS(aldagai_indize(PROG)) := ε;
    elsif esleipen_cons? (PROG) then
        VARS(aldagai_indize(PROG)) :=
            ikur_erabili (PROG) & VARS(2_aldagai_indize(PROG));
    elsif esleipen_cdr? (PROG) then
        VARS(aldagai_indize(PROG)) :=
                        cdr (VARS(2_aldagai_indize (PROG)));
    elsif konposaketa?(PROG) then
        PILA := pilaratu (barne_agindu (PROG),
                        pilaratu (2_barne_agindu (PROG), PILA));
    elsif baldintzazkoa? (PROG) then
        if ikur_erabili(PROG) = lehena (VARS(aldagai_indize(PROG))) then
            PILA := pilaratu (barne_agindu (PROG), PILA);
        end if;
    else
        -- iterazioa da
        if nonem? (VARS(aldagai_indize (PROG))) then
            PILA := pilaratu (barne_agindu (PROG), pilaratu (PROG, PILA));
        end if;
    end if;
end loop;
-- Px-ren egikaritzapena bukatzen bada, emaitza deskargatzen dugu
X0 := VARS (0); ⌫
```

Makroprogramen erabiltzen diren datu-motekin burututako eragiketen inguruan argibide gehiago behar izanez gero C Eranskinera jotzea gomendatzen dugu.

OROKORTZEA: Aurreko emaitza orokortzen da, **k** argumentudun funtzio konputagarrientzat den funtzio unibertsala ere konputagarria dela frogatzeko. Horrela, $\mathbf{k}\geq 0$ guztietarako $\Phi^{k+1} : \mathbb{W} \times \sum^{*k} \longrightarrow \sum^{*}$ funtzioa konputagarria dela daukagu:

$$\Phi^{k+1}(x, y_1, \ldots, y_k) = \begin{cases} \varphi_x(y_1, y_2, \ldots, y_k) & \varphi_x(y_1, y_2, \ldots, y_k)\downarrow \\ \perp & \text{bestela} \end{cases}$$

⌦ Aurretik deskribaturiko programa berbera nahikoa izango da, baina bigarren lerroaren ordez honakoa jarriz:

```
VARS(1):= X2;
VARS(2):= X3;
VARS(3):= X4;
 ...
VARS(K):= X[K+1]; ⌫
```

> *Historikoki, Zerrendatze Teoremaren frogapenak honek agerian uzten duena baino aurrerakuntza kontzeptual handiagoa suposatu zuen. Kontuan hartu behar da Konputagarritasun Teoriaren hastapenetan (gogoratu, oraindik konputagailuak existitzen ez zirenean) programa bakoitza sistema konputazional zehatz baten portaera operatiboaren deskribapentzat hartzen zela. Hau da, algoritmo bat existitzeak ataza zehatz hori egiteko gai zen makina bat eraikitzea posible zela frogatzen zuen. Funtzio Unibertsalaren konputagarritasunak helburu orokorreko konputagailuen existentzia frogatzea suposatu zuen, gailu bakar bat (**U** programa konputatzeko gai den hori), existitzen den beste edozein gailu fisikoren portaera eraginkortasunez simulatzearren, bere portaera modulatzeko gai delako. Honek hardware/software dualtasuna behin-betikoz ezartzea eta programagarritasunaren kontzeptua formalizatzea ekarri zituen. Programagarritasunarekin, gailu konputazionalek beren egitura fisikoa aldatu behar izan gabe beren funtzionalitatea errotik aldatzeko duten gaitasuna adierazten dugu.*

Funtzio unibertsala, frogatu dugun moduan, konputagarria denez, beste edozein programaren makroadierazpenetan erabil ahal izango dugu, eta horrela beste funtzio batzuen konputagarritasuna frogatu ahal izango dugu, esaterako, ondoren definitzen dugun $\chi : \mathbb{W} \times \sum^{*} \longrightarrow \sum^{*}$ funtzioarena:

$$\chi(x, y) \cong \begin{cases} \varphi_x(y) & \text{car}_a?(y) \\ y^R & \text{car}_b?(y) \wedge \varphi_x(x \bullet x)\downarrow \\ \perp & \text{bestela} \end{cases}$$

Hau konputatuko duen makroprograma hau bezain sinplea izango da:

if car_a?(X2) **then**
    X0 := Φ(X1, X2);
elsif car_b?(X2) **then**
    R := Φ(X1, X1&X1);
    X0 := $X2^R$;
else X0 := $\perp\!\!\!\perp$;
end if;

Kontuan hartu R := Φ(X1, X1&X1); makroaginduaren helburua filtro-lana egitea dela. Baldin eta $\varphi_x(\mathbf{x}\bullet\mathbf{x})$-k dibergitzen badu, itxaroten dugun portaera lortzen dugu, $\chi(\mathbf{x},\mathbf{y})$ ere indefinituta geratuko delako. Beste kasuan, programa hurrengo agindura pasako da, eta horrela kalkulatu eta $\mathbf{y}^R$ itzuli ahal izango du.

4.3 Etendako egikaritzapena

Zerrendatze teoremarekin edozein while-programaren funtzionamendua amaieraraino simula dezakegula egiaztatzen dugu, baina zenbaitetan hau ez da egokia, programa horrek ziklatzen badu bere konputazio infinituan atzetik eramango gaituelako.

Noizean behin, programa baten egikaritzapena neurrizko denbora baten ondoren etetea komenigarriagoa izango da, nahiz eta horrela eginez konputazio horrek azkenean bukatuko ote zuen ziurtasunez jakingo ez dugun. Horretarako **T** predikatua oso onuragarria gertatuko zaigu, honek, pausoetarako goi-muga maximoa hartuz, programa batek datu baten gainean konbergitzen duenetz behatzen duelako. Konputazio-pausoa $\mathbf{P_x}$ programaren ekintza bat burutzeari, hots, esleipen bat egikaritzeari edo baldintza baten egiaztapenari, dagokiola ulertzen dugu.

PROPOSIZIOA: Ondoren definitzen dugun $\mathbf{T}: \mathbb{W} \times \sum^* \times \mathbb{N} \to \mathbb{B}$ predikatua erabakigarria da:

$\mathbf{T}(\mathbf{x},\mathbf{y},\mathbf{p}) \Leftrightarrow$ $\mathbf{P_x}$-k **y**-ren gainean konbergitzen du **p** pausotan edo gutxiagotan

⌦ **T** predikatua konputatzen duen programa Φ-renaren oso antzekoa izango da, baina egikaritzapena eteteko pauso-kontagailu bat erabiliz:

VARS(azken_aldagaia (X1)) := ε; VARS(1) := X2;
PAUSOAK := 0; PILA := pilaratu (X1, <]);
while not P_hutsa? (PILA) **and** PAUSOAK<X3 **loop**
    PAUSOAK := PAUSOAK+1;
    PROG := tontorra (PILA); PILA := despilatu (PILA);
    if esleipen_hutsa? (PROG) **then**

```
                VARS(aldagai_indize(PROG)) := ε;
        elsif esleipen_cons? (PROG)  then
                VARS (aldagai_indize(PROG)) :=
                        ikur_erabili (PROG) • VARS (2_aldagai_indize (PROG));
        elsif  esleipen_cdr? (PROG)  then
                VARS (aldagai_indize(PROG)) :=
                                cdr (VARS (2_aldagai_indize(PROG)));
        elsif  konposaketa?(PROG)  then
                PILA:= pilaratu(barne_agindu(PROG),
                                pilaratu(2_barne_agindu (PROG), PILA));
                PAUSOAK := PAUSOAK–1;
                -- inolako ekintzarik burutzen ez den kasu bakarra
        elsif  baldintzazkoa? (PROG)  then
                if ikur_erabili(PROG) = lehena(VARS(aldagai_indize(PROG)))  then
                        PILA := pilaratu (barne_agindu (PROG), PILA);
                end if;
        else
                if  nonem? (VARS (aldagai_indize (PROG)))  then
                        PILA := pilaratu (barne_agindu(PROG), pilaratu (PROG, PILA));
                end if;
        end if;
    end loop;
  -- predikatua egiazkoa izango da Px-ren egikaritzapena bukatu bada
    X0 := P_hutsa? (PILA);  ⌫
```

Programa baten egikaritzapenak gutxienez beti pauso bat eman beharko duela kontuan hartu beharrekoa da, hortaz, T(**x**,**y**,0) beti faltsua izango dela.

Jakina, **T**(**x**,**y**,**p**)-ren emaitza faltsua denean ez dugu arrazoiari buruzko informazioa jasotzen. $\mathbf{P_x}$-k **y**-ren gainean ez konbergitzeagatik izan daiteke edo bestela konputazioa bukatzeko **p** pauso-kopurua nahikoa ez izateagatik.

Batzuetan konputazioa bukatzen denentz ez ezik, zein emaitzarekin egiten duen galdetzea erabilgarria gerta daiteke. Horregatik beste predikatu bat definitzen dugu, **E** predikatua, zeinek argumentu gehigarria hartzen duen, bilatu nahi dugun emaitza hain zuzen ere.

PROPOSIZIOA: Honako $\mathbf{E}: \mathbb{W} \times \sum^* \times \mathbb{N} \times \sum^* \rightarrow \mathbb{B}$ predikatua erabakigarria da:

$\mathbf{E}(\mathbf{x},\mathbf{y},\mathbf{p},\mathbf{z}) \Leftrightarrow$ $\mathbf{P_x}$-k **y**-ren gainean konbergitzen du **p** pausotan edo gutxiagotan, eta irteera bezala **z** sortuz

⌦ **E**-ren programa lortzeko **T** konputatzen duenaren gainean ukitu txiki bat bakarrik da beharrezkoa. Bere azken aginduaren ordez honakoa jartzea nahikoa izango da:

X0 := P_hutsa? (PILA) **and** VARS(0) = X4; ⌫

E(**x**,**y**,**p**,**z**)-rentzat emaitza negatiboa edukitzea zio desberdinengatik izan daitekeelako aurkitzen gara berriz ere. $\mathbf{P_x}$-k **y**-ren gainean ez konbergitzeagatik izan daiteke, edo konputazioa bukatzeko **p** pauso kopurua nahikoa ez izateagatik, edo bestela konputazioa **z**-ren desberdina den emaitza sortuz bukatzeagatik.

OROKORTZEA: **k** sarrerekin erabilitako while-programetarako **T** eta **E** predikatuak ere erabakigarriak direla frogatzeko aurreko emaitzak orokortzen dira. Horrela, **k**≥0 denetarako, ondoren definitutako $\mathbf{T}^{k+2} : \mathbb{W} \times \sum^{*k} \times \mathbb{N} \to \mathbb{B}$ eta $\mathbf{E}^{k+3} : \mathbb{W} \times \sum^{*\mathbf{k}} \times \mathbb{N} \times \sum^{*} \to \mathbb{B}$ predikatuak erabakigarriak direla daukagu

$\mathbf{T}^{k+2}(\mathbf{x},\mathbf{y_1}, \ldots,\mathbf{y_k},\mathbf{p})$ ⇔ $\mathbf{P_x}$-k ($\mathbf{y_1}$, …,$\mathbf{y_k}$)-ren gainean konbergitzen du **p** pauso edo gutxiagotan

$\mathbf{E}^{k+3}(\mathbf{x},\mathbf{y_1}, \ldots,\mathbf{y_k},\mathbf{p},\mathbf{z})$ ⇔ $\mathbf{P_x}$-k ($\mathbf{y_1}$, …,$\mathbf{y_k}$)-ren gainean konbergitzen du **p** pauso edo gutxiagotan eta irteera bezala **z** sortuz

⌦ $\mathbf{T}^{k+2}$-ren kasuan **T**-ren programa aldatu egin beharko da, bere lehen lerroa hauengatik ordezkatzeko:

```
VARS (1) := X2;
VARS (2) := X3;
VARS (3) := X4;
 ...
VARS (K) := X[K+1];
```

eta begiztaren baldintza, honengatik:

while not P_hutsa? (PILA) **and** PAUSOAK < X[K+2] **loop**

$\mathbf{E}^{k+3}$-rentzat, aurrekoaz gainera, bukaerako esleipena, honengatik ordezkatzea beharrezkoa izango da ere:

X0 := P_hutsa? (PILA) **and** VARS(0) = X[K+3]; ⌫

Hasieran definitutako **T** eta **E** predikatuak, jakina, $\mathbf{T}^3$ eta $\mathbf{E}^4$ kasu partikularrak besterik ez dira.

4.4 Prozesu tartekatzea

Funtzioren baten konputagarritasuna frogatzeko algoritmo bat programatzen dugunean, gure helburua lortzeko, datu bat edo gehiago behar duten programa bat edo gehiago egikaritzeko beharrarekin aurki gaitezke gure prozesuaren barnean. Adibidez, ondorengoko $\psi : \mathbb{W} \times \mathbb{N} \dashrightarrow \mathbb{N}$ funtzioa konputagarria dela frogatu nahi badugu:

$$\psi(\mathbf{x},\mathbf{y}) \cong \begin{cases} 12 & \varphi_{\mathbf{x}}(\mathbf{y})\downarrow \wedge \varphi_{\mathbf{x}}(\mathbf{y}+1)\downarrow \\ \perp & \text{bestela} \end{cases}$$

bi prozesu egikaritu behar dugu: **y** datuarekin **x** programari dagokiona ($\varphi_{\mathbf{x}}(\mathbf{y})$) eta **y**+1 datuarekin **x** programari dagokiona ($\varphi_{\mathbf{x}}(\mathbf{y}+1)$). Kasu honetan, biak konbergenteak direnentz egiaztatu nahi dugunez, era sekuentzialean egikari ditzakegu. Adibidez, honako programak ψ konputatuko luke:

```
R := Φ(X1, X2);
R := Φ(X1, X2+1);
X0 := 12;
```

Hiru egoera eman daitezke:

a) $\varphi_{\mathbf{x}}(\mathbf{y})\uparrow$, beraz, gure makroprogramaren lehen esleipena ez da bukatzen; portaera egokia da, kasu horretan $\varphi_{\mathbf{x}}(\mathbf{y})\downarrow \wedge \varphi_{\mathbf{x}}(\mathbf{y}+1)\downarrow$ baldintza ez baita betetzen, eta hortaz $\psi(\mathbf{x},\mathbf{y})$ indefinitua da.

b) $\varphi_{\mathbf{x}}(\mathbf{y})\downarrow$ baina $\varphi_{\mathbf{x}}(\mathbf{y}+1)\uparrow$, beraz, gure makroprogramaren lehen esleipena bukatzen da baina bigarrena ez; portaera egokia da kasu horretan ere $\varphi_{\mathbf{x}}(\mathbf{y})\downarrow \wedge \varphi_{\mathbf{x}}(\mathbf{y}+1)\downarrow$ baldintza ez baita betetzen, eta hortaz $\psi(\mathbf{x},\mathbf{y})$ indefinitua da.

c) $\varphi_{\mathbf{x}}(\mathbf{y})\downarrow$ eta $\varphi_{\mathbf{x}}(\mathbf{y}+1)\downarrow$, beraz, gure makroprogramaren lehen bi esleipenak bukatzen dira eta 12 emaitza sortzen dugu; portaera egokia da, kasu horretan $\varphi_{\mathbf{x}}(\mathbf{y})\downarrow \wedge \varphi_{\mathbf{x}}(\mathbf{y}+1)\downarrow$ baldintza bete egiten delako, eta hortaz $\psi(\mathbf{x},\mathbf{y})$-ren balioa guk emandako horixe da.

Alabaina, honako $\chi : \mathbb{W}^2 \times \mathbb{N} \longrightarrow \mathbb{N}$ funtzioaren konputagarritasuna frogatzeko beharrean aurkituko bagina:

$$\chi(\mathbf{x},\mathbf{y},\mathbf{z}) \cong \begin{cases} \mathbf{z}^2 & \varphi_{\mathbf{x}}(\mathbf{z})\downarrow \vee \varphi_{\mathbf{y}}(2*\mathbf{z})\downarrow \\ \perp & \text{bestela} \end{cases}$$

oso egoera desberdinean egongo ginateke. Lehen bezala aztertu beharreko bi prozesu ditugu: **z** datuarekin **x** programari dagokiona ($\varphi_{\mathbf{x}}(\mathbf{z})$) eta 2∗**z** datuarekin **y** programari dagokiona ($\varphi_{\mathbf{y}}(2*\mathbf{z})$). Baina oraingo honetan bien konbergentzia ez zaigu interesatzen, bakarrarena nahikoa zaigulako. Horietatik batek bakarrik konbergitzen badu, zehazki zein den aurretik, *a priori*, jakitea ezinezkoa dela da gure arazoa, hau da, gure programak $\varphi_{\mathbf{x}}(\mathbf{z})$ prozesua $\varphi_{\mathbf{y}}(2*\mathbf{z})$ baino lehenago exekutatzen badu, χ funtzioa ez du ondo konputatuko $\varphi_{\mathbf{x}}(\mathbf{z})\uparrow$ baina $\varphi_{\mathbf{y}}(2*\mathbf{z})\downarrow$ betetzen den kasu guztietarako (programak ziklatu egingo du $\mathbf{z}^2$ itzuli beharko zuenean). Baina $\varphi_{\mathbf{y}}(2*\mathbf{z})$ prozesua lehendabizi exekutatu arren ez dugu ezer konpontzen, $\varphi_{\mathbf{x}}(\mathbf{z})\downarrow$ baina $\varphi_{\mathbf{y}}(2*\mathbf{z})\uparrow$ betetzen den beste kasu batzuk egongo baitira.

Irtenbide ideala bi prozesuak paraleloan exekutatzea izango litzateke, horrela bietako bat bukatzean bestea bertan behera utzi eta behar dugun emaitza sortzeko aukera izango genuke. Baina while-programetan ez dugu prozesaketa paralelorik. Paralelismoa simulatzea da egin dezakeguna, bi prozesuei konputazio-denbora txandaka emanez: lehendabizi $\varphi_x(\mathbf{z})$ prozesuaren pauso bat exekutatuko dugu, ondoren $\varphi_y(2*\mathbf{z})$-ren pauso bat, jarraian $\varphi_x(\mathbf{z})$-ren bigarren pausoa, beste bat $\varphi_y(2*\mathbf{z})$-rena, eta horrela pausoka jarraituz. Era honetan bi prozesuetatik batek bukaera duenentz detektatzeko gai izango gara, bestea bukatu nahiz ez. Egoera hori adibide hipotetiko baten bitartez 4.7 irudian azaltzen da.

PROZESUAK

Konputazio-pausoak	$\varphi_{21}(6)$	$\varphi_{248}(12)$
1	1. ↑	2. ↑
2	3. ↑	4. ↑
3	5. ↑	6. ↑
4	7. ↑	8. ↑
5	9. ↑	10. ↓
6		
…		

4.7 irudia: $\chi(21, 248, 6)$-ren kalkuluaren adibidea, $\varphi_{21}(6)$-k dibergitzen duela eta $\varphi_{248}(12)$-k 5 pausotan konbergitzen duela suposatuz. Bilatzen den baldintza topatu arte, taulan bi prozesuen pausoen exekuzio-ordena eta kasu bakoitzean lortutako emaitza erakusten dira. Emaitza hori, aipatu pausoetan prozesuaren amaiera (↓) ala prozesua, momentuz, ez bukatzea (↑) izan daiteke.

Simulazioa egiteko aurreko atalean definitu dugun **T** predikatu erabakigarriaren laguntzak berebiziko garrantzia izango du. Predikatu horrek konputazioak pauso-kopuru zehatz batean konbergitzen duenentz esaten digunez, "$\varphi_x(\mathbf{z})$ prozesuaren **p** pauso exekutatzeko" **T**(**x**, **z**, **p**) erabiliko dugu. χ funtzioari dagokion programa nolakoa izango litzatekeen ikus dezagun:

```
PAUSOAK := 1;
--Bi prozesuetan noraino konputatu dugun jakiteko PAUSOAK aldagaia daukagu
while not T(X1, X3, PAUSOAK) and not T(X2, 2∗X3, PAUSOAK) loop
      PAUSOAK := PAUSOAK+1;
end loop;
X0 := X3∗X3;
```

Berriz ere hiru egoera eman daitezke:

a) $\varphi_x(\mathbf{z})\downarrow$ baina $\varphi_y(2*\mathbf{z})\uparrow$ (edo alderantziz); PAUSOAK aldagaia konbergentea den prozesua osatzeko beharrezkoa den baliora iritsiko da, hortaz, begizta bukatu egingo da eta itxaroten den $\mathbf{z}^2$ balioa sortuko dugu.

b) $\varphi_x(\mathbf{z})\downarrow$ eta $\varphi_y(2*\mathbf{z})\downarrow$; PAUSOAK aldagaia, lehendabizi konbergitzen duen prozesua osatzeko beharrezkoa den baliora iritsiko da, hortaz, begizta bukatu egingo da eta itxaroten den $\mathbf{z}^2$ balioa sortuko dugu.

c) $\varphi_x(\mathbf{z})\uparrow$ eta $\varphi_y(2*\mathbf{z})\uparrow$; PAUSOAK aldagaia muga gabe handituz joango da eta begizta ez da bukatuko, eta horrela, orain ere portaera egokia dugu, kasu honetan $\chi(\mathbf{x},\mathbf{y},\mathbf{z})$ indefinitua delako.

> *Konputazioen simulazio paralelorako deskribatu dugun mekanikaren eta ingurune errealean, esaterako, sistema eragileren batean, arrazoizkoa izango litzatekeen mekanikaren artean desberdintasun nabarmena badago. 4.7 irudiko adibidean, $\varphi_{248}(12)$-k 4 pausotan ez duela konbergitzen egiaztatzen dugunean eta $\varphi_{21}(6)$-ren 5. pausoa kalkulatzeari ekiten diogunean, azken prozesu hori ez dugu utzi genuen puntutik jarraitzen, baizik eta aurreko iterazioan egindako lehenengo lau pausoak berriz kalkulatzen ditugu. Lan egiteko modu hori sistema erreal batean oso garestia izango litzatekeen arren, tartekatutako prozesu guztien egoera-bektorea gure programan esplizituki gorde behar izatea ekiditen du. Prozesu bakoitzari esleitutako denboraren inguruan gauza bera esan daiteke: gure kasuan txanda bakoitzean agindu berri bakarra egiten da, bizitza errealean milaka agindu exekutatzea arraroa ez den bitartean itxarote-denborak eta prozesatzailearen erregistroetako joan-etorriak optimizatzeko asmotan.*

Laburtzearren esan dezakegu: zenbait egoeratan, problema baten ebazpenak dibergenteak izan litezkeen prozesu batzuen multzoa analizatzea eska dezake, baina azterketa horrek ez gaitu derrigorrez konputazio-segida guztia osatzera behartzen, baizik eta, horien artean, dibergitzen ez duen eta baldintza zehatz bat betetzen duen bat aurkitzera. Kasu horietan ebazpena bilatzeko gure adimena zorroztu egin beharko dugu, infinituak diren konputazioak saihestuz, behar den erantzuna eman ahal izateko. Deskribatu berri dugun teknika, ***prozesu tartekatzea***ren teknika, laguntza handikoa izango zaigu. Gogoratu teknika honen ezaugarria dela prozesuei lotutako konputazioak tartekatzen edo mihiztatzen (‘*dovetail*’, inglesez) uztea, konputazio horiek infinituak izan litezkeen arren. Hori dela eta, **‘*dovetailing*’ *teknika*** ere deitu ohi zaio.

4.4.1 Prozesu-kopuru mugatuaren tartekatzea

Paraleloan exekutatu beharreko prozesuen kopurua bi baino askoz handiagoa izan daiteke, eta egikaritzapen-denborara arte ezin zehaztea ere gerta daiteke.

Adibidez, demagun $\xi: \mathbb{W} \longrightarrow \mathbb{B}$ funtzioaren konputagarritasuna frogatu nahi dugula eta funtzio horrek, $\mathbf{P_x}$ programa bakoitzarentzat, emaitza moduan **ab** hitza sortuko duen sarrera bilatzen duela. Gainera sarrera hori, **z**, balio-tarte baten barnean aurkitu behar da, **x**-ren menpekoa den tartearen barnean, hain zuzen ere:

$$\xi(\mathbf{x}) \cong \begin{cases} \text{true} & \exists \mathbf{z}\,(\mathbf{z} < \mathbf{x} \wedge \varphi_{\mathbf{x}}(\mathbf{z}) = \mathbf{ab}) \\ \perp & \text{bestela} \end{cases}$$

Kasu honetan, $\mathbf{P_x}$ programa emanda, $\varphi_{\mathbf{x}}(0)$, $\varphi_{\mathbf{x}}(1)$, …, $\varphi_{\mathbf{x}}(\mathbf{x}\text{-}1)$ prozesuetatik baten batek emaitza moduan **ab** sortzen duen egiaztatu behar dugu, horretarako prozesu horrek konbergitu egin beharko duela kontuan hartuta.

Berriz ere, prozesu horien exekuzio sekuentziala guztiz desegokia da eta beraien analisi paraleloa egitera joko dugu, prozesuei egikaritzapen-pausoak era ordenatuan esleituz. Analisi horren adibide hipotetikoa da 4.8 irudian azaltzen duguna.

PROZESUAK

Konputazio-pausoak	$\varphi_6(0)$	$\varphi_6(1)$	$\varphi_6(2)$	$\varphi_6(3)$	$\varphi_6(4)$	$\varphi_6(5)$
1	1^o ↑	2^o ↑	3^o ↑	4^o ↑	5^o ↑	6^o ↑
2	7^o ↑	8^o ↑	9^o ↑	10^o ↑	11^o ↑	12^o ↑
3	13^o ↑	14^o **=a**	15^o ↑	16^o ↑	17^o ↑	18^o ↑
4	19^o ↑	20^o **=a**	21^o ↑	22^o ↑	23^o **=ε**	24^o ↑
5	25^o ↑	26^o **=a**	27^o **=ab**			
6						**=ab**
…						

4.8 irudia: $\xi(6)$-ren kalkuluaren adibidea. Bilatutako baldintza bete arte prozesuen pausoen egikaritzapenak jarraitutako ordena adierazten da taulan. Gainera kasu bakoitzean eskuratutako emaitza ere erakusten da, hau izan daiteke, prozesua ez bukatzea (↑) ala bukatzea eta emaitza bezala **w** hitza sortzea (=**w**). Analisiak, $\varphi_6(2)$ prozesuak bai **ab** hitza 5 pausotan sortzen duela aurkitzen du, eta $\varphi_6(0)$ eta $\varphi_6(3)$ prozesuek dibergitzeak ez dio inolako eragozpenik sortzen horretarako. $\varphi_6(5)$ -k era berean emaitza bezala **ab** sortzen duela suposatu dugu, baina horretarako 6 pauso behar dituenez gure analisia ez da hori egiaztatzera iritsiko.

Aipatu irudian azaldutako algoritmoa inplementatuko duen programak begizta nagusi bat edukiko du, aurreko adibidean erabilitakoaren antzekoa den PAUSOAK aldagai batek kontrolatuko duena. Begizta horren barnean beste bat egongo da, esandako pauso-kopurura arte dagozkion prozesuak arakatuko dituena,

baliagarria den **E** predikatua erabiliz (kasu honetan konputazioaren emaitza interesatzen baitzaigu):

```
PAUSOAK := 1;
AURKITUA := false;
while not AURKITUA loop
      for Z in Ɛ .. aurre(X1) loop
            AURKITUA := AURKITUA or E(X1, Z, PAUSOAK, ab);
      end loop;
      PAUSOAK := PAUSOAK+1;
end loop;
X0 := true;
```

Aurreko kasuetan bezala, gerta daiteke begizta nagusiak PAUSOAK aldagaia etengabe handitzea, $\mathbf{P}_x$-ri **ab** emaitza sorraraziko dion **z** balioa aurkitu gabe. Baina kasu honetan gure programaren portaera dibergentea ere funtzioarenarekin, $\xi(\mathbf{x})\uparrow$, bat dator.

> *Simulazio paralelorako gure metodoak badu oso errealista ez den beste ezaugarri bat: prozesu bat egokia ez den emaitza batekin bukatzen denean, analisi-lana ez da prozesu horretarako amaitzen. 4.8 irudiko adibidean, bere hirugarren pausoa egikaritu ondoren badakigu $\varphi_6(1) = \boldsymbol{a}$ betetzen dela eta, ondorioz, 1 datua ez zaigula interesatzen. Hala ere, 4 eta 5 pausorekin probatzen jarraitzen dugu. Hau ekidin genezake, baina horrek arrakastarik gabe bukatutako prozesuen taula mantentzera behartuko gintuzke. Berriz ere, gure helburuetarako, eraginkorra ez izan arren motza eta erraza den programa nahiago dugula aurkitzen dugu. Eta honen zergatia da while-programetatik eta makroprogrametatik aipagarria den bakarra beren existentzia baizik ez dela. Ez baditugu exekutatu behar, gutxien-gutxienez diseinu garaian ahaleginak aurrez ditzagun.*

4.4.2 Prozesu-kopuru ez-mugatuaren tartekatzea

Hala eta guztiz ere, paralelismoaren simulazioaren ideia honekin oraindik ondoko beste birfinketa bat egin daiteke, horrela, zailagoak diren problemak ebazteko baliagarria izan dadin. Azken adibideko funtzioan, analizatu beharreko prozesuen kopurua, egikaritzapen-denboran zehazten zen arren, beti mugatuta zegoen ($\mathbf{P}_x$ sarrerarentzat **x** prozesu aztertu behar ziren) eta ondorioz horrek, egikaritzapen-denboren banaketarako algoritmo zuzena, txandatan oinarritutakoa, definitzen uzten zigun. Alabaina, beste kasu batzuetan kontuan hartu beharreko prozesuen kopurua infinitua izan daiteke, eta hau antzeko algoritmoa aplikatzeko zailtasun nabarmena izan daiteke.

Honen adibidea $\theta : \mathbb{W} \times \sum^* \longrightarrow \mathbb{B}$ funtzioa izan daiteke:

$$\theta(\mathbf{x},\mathbf{y}) \cong \begin{cases} \text{true} & \mathbf{y} \in \mathbf{R}_x \\ \perp & \text{bestela} \end{cases}$$

Demagun bere konputagarritasuna frogatu nahi dugula. $\mathbf{y} \in \mathbf{R}_x$ betetzen den egiaztatzeko balizko metodoa, balizko sarrera bakoitzeko, eta $\varphi_x(0)$, $\varphi_x(1)$, $\varphi_x(2)$… prozesuak aztertuz, $\mathbf{P}_x$ programak itzul ditzakeen emaitzak aztertzea izango litzateke. Baina, lehen ez bezala, prozesu-zerrenda horrek ez du amaierarik, eta honek gure teknikan ondorio desatsegina izango du: lehenik prozesu guztien konputazio-pauso bat kalkulatzea, gero bi, ondoren hiru ... horrela egiteak ez du zentzurik, besterik gabe, prozesu guztiei bere lehen pausoa esleitzearekin ez genukeelako inoiz bukatuko.

Arrazoizko irtenbidea izango litzateke pauso eta prozesu gutxirekin hastea, eta ondoren analisiak aurrera egiten duenarekin batera biak handitzen joatea. Hau kontu pixka batekin eginez gero, prozesu guztiek izango dute analizatuak izateko aukera, eta gainera denek beharrezkoa den bezain eskuzabala izango den pauso-esleipena jaso ahal izango dute. Adibidez, $\varphi_x(0)$, $\varphi_x(1)$, $\varphi_x(2)$, $\varphi_x(3)$ eta $\varphi_x(4)$ prozesuen 4 pauso kalkula ditzakegu. Itxaroten dugun emaitza lortzen ez badugu (bukatzen ez dutelako, edo bilatzen dugun irteera sortzen ez dutelako), $\varphi_x(0)$, $\varphi_x(1)$, $\varphi_x(2)$, $\varphi_x(3)$, $\varphi_x(4)$ eta $\varphi_x(5)$ prozesuen 5 pauso exekutatzera pasako gara, eta horrela jarraituko genuke. Argia dirudi era honetara eginez, prozesu batzuek dibergitzeak konbergenteak direnak amaierara eramateko eragozpenik ez duela sortzen. θ funtzioaren balio bat kalkulatzean ideia hau nola aplikatzen den 4.9 irudian erakusten da.

PROZESUAK

Konputazio-pausoak	$\varphi_{207}(0)$	$\varphi_{207}(1)$	$\varphi_{207}(2)$	$\varphi_{207}(3)$	$\varphi_{207}(4)$	$\varphi_{207}(5)$	…
1	1. ↑	2. ↑					
2	3. ↑	4. ↑	5. ↑				
3	6. ↑	7. =ε	8. ↑	9. ↑			
4	10. ↑	11. =ε	12. ↑	13. ↑	14. ↑		
5	15. ↑	16. =ε	17. **=ab**	18. ↑	19. ↑	20. **=ab**	
6	21. ↑	22. =ε	23. **=ab**	24. **=bb**			
7							
…							

4.9 irudia: $\theta(207,\mathbf{bb})$ kalkulatzearen adibidea. Bilatutako baldintza bete arte prozesuei pausoak esleitzeko jarraitutako ordena adierazten da taulan. Gainera kasu

bakoitzean eskuratutako emaitza ere erakusten da, hau izan daiteke prozesua aipatu pausotan ez bukatzea (↑) ala bukatzea eta emaitza bezala **w** hitza sortzea (=**w**). Analisiak $\mathbf{bb} \in \mathbf{R}_{207}$ dela aurkitzen du, $\varphi_{207}(3)$-k **bb** hitza 6 pausotan sortzen duelako. Azkenean 6 sarrera desberdin besterik ez da aztertu, baina $\varphi_{207}(3)$ kalkulatzeko, adibidez, 1000 pauso behar izan balira eta beste edozein konputaziotan emaitza bezala **bb** agertuko ez balitz, orduan $\varphi_{207}(1000)$ arte arakatuko zen.

θ funtzioa konputatuko lukeen programa honakoa litzateke:

```
PAUSOAK :=  1;
AURKITUA :=  false;
while  not AURKITUA loop
      for  DATUA  in  Ɛ .. PAUSOAK loop
            AURKITUA :=  AURKITUA or E(X1, DATUA, PAUSOAK, X2);
      end loop;
      PAUSOAK := PAUSOAK+1;
end loop;
X0:= true;
```

Tartekatze teknika, analizatu beharreko prozesuen zerrenda infinitua den kasuetara heda daitekeela ondoriozta dezakegu, beraz. Multzo infinitu bat sakonki aztertzeak suposatzen duen itxurazko kontraesan hau aska daiteke, soluzioa, aurkitzekotan, prozesu konbergente zehatzen batean aurkituko delako. Zehazki zein den aurretik ez jakitea da arazo bakarra, eta hortik dator gure bilaketa-eremua indefinituki eta era sistematikoan zabaltzeko beharra, arrakasta ziurtatuta edukitzeko, bilatzen dugun konputazio hori zerrendan oso urruti egon arren eta egiaztatzeko pauso asko eman behar izan arren. Jakina, tartekatze edo *dovetailing*-eko beste kasu guztietan bezala, bilaketa-mota honek ez du bukaerarik, soluzioaren baldintza inoiz betetzen ez bada, eta hori dela eta, huts egindako bilaketetan inolako erantzunik sortzeko exigentzia ez dugunean bakarrik aplikatu ahal izango dugu. Adibidez, $\mathbf{g}: \mathbb{W} \times \sum^* \rightarrow \mathbb{B}$ funtzioa konputatu nahiko bagenu:

$$\mathbf{g}(\mathbf{x},\mathbf{y}) \cong \begin{cases} \text{true} & \mathbf{y} \in \mathbf{R}_{\mathbf{x}} \\ \text{false} & \text{bestela} \end{cases}$$

orduan tartekatze teknika guztiz alferrikakoa izango litzateke[4].

4.4.3 Kodeketa funtzioen erabilpena

Zenbait kasutan prozesu tartekatzearen teknika zaildu egin daiteke, gure bilaketa-eremuak dimentsio asko izateagatik. Aurretik ikusi ditugun adibideetan (ξ eta θ funtzioen kasua) ez zen horrelakorik gertatzen: jatorrizko problemak, emandako programa baten gainean portaera zehatz bat sorraraziko lukeen datu bat

[4] Itxarotekoa zena, bestalde, funtzio konputaezina baita. Dena den, hurrengo kapitulura arte ez gara frogatzeko gai izango.

bilatzea eskatzen zigun eta tartekatzearen metodoak, gainera, programa konbergituz portaera hori erakusteko nahikoa den pauso-kopurua bilatzea eskatzen zigun. Bilaketarako bi dimentsio horiek (datua, pausoak) tartekatzearen ideia bi dimentsioko taula batean erakusten uzten ziguten. Baina honako $\eta : \mathbb{W} \longrightarrow \mathbb{B}$ funtzioarekin tratatzen badugu:

$$\eta(\mathbf{x}) \cong \begin{cases} \text{true} & \varphi_x \text{ ez da injektiboa} \\ \perp & \text{bestela} \end{cases}$$

egoera pixka bat konplikatuagoa da. $\mathbf{P_x}$ programa emanda, ez-injektiboa den funtzio bat konputatzen duela nola egiazta dezakegu?. Bi datu desberdin aurkitu beharko ditugu, zeintzuetarako programak emaitza bera, **r**, sortuko duen. Bi balio horien bilaketak oztopo larria edukiko duela, hots, bilatzen dugun hori bi balio zehatzekin, **y** eta **z**, egiaztatzen saiatzen bagara, $\varphi_x(\mathbf{y})$-k nahiz $\varphi_x(\mathbf{z})$-k dibergi dezaketela kontuan hartu behar dugu. Horregatik prozesu tartekatzearen teknika erabiliko dugu: konputazio-pauso batekin probatuz hasiko gara, gero birekin, eta horrela jarraian gainontzekoekin, eta **p** pauso-kopuru bakoitzarekin, beren balioak 0..**p** tartean dituzten (**y**,**z**) datu-bikote desberdin guztiak probatuko ditugu. Horrela ziur jakingo dugu, hala behar izanez gero, $\mathbf{P_x}$ programari sarrera moduan eman diezazkiokegun balizko datu-bikote desberdin guztiek egiaztatuak izateko aukera izango dutela eta, aldi berean, ($\varphi_x(\mathbf{y})$, $\varphi_x(\mathbf{z})$) prozesu-bikote hipotetiko bakoitzean beharrezkoa den konputazio-pauso kopurua exekutatua izango dela. Idatziko genukeen programa honakoa litzateke:

```
PAUSOAK := 1;
AURKITUA := false;
while not AURKITUA loop
      for Y in Ɛ .. PAUSOAK loop
            for Z in Ɛ .. PAUSOAK loop
                  if Y<>Z and T(X1, Y, PAUSOAK) and T(X1, Z, PAUSOAK) then
                        AURKITUA := AURKITUA ∨ Φ(X1,Y) = Φ(X1,Z);
                  end if;
            end loop;
      end loop;
      PAUSOAK := PAUSOAK+1;
end loop;
X0 := true;
```

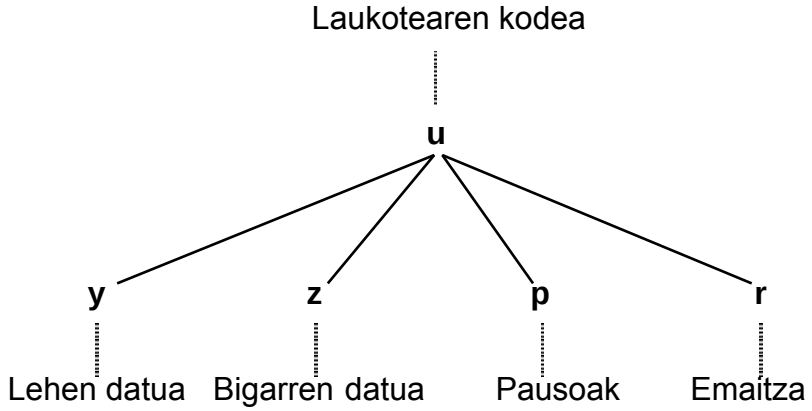
Mota honetako problematan oso baliagarria izan daiteke azalpen hau: tartekatze teknika, egia esan, eremu dimentsioanitz batean bilaketa sakona egiteko modu bat besterik ez da. Adibidez, η funtzioa konputatzeko zenbait gauza aurkitu behar ditugu: **y** datua, beste **z** datua, eta $\varphi_x(\mathbf{y})$ nahiz $\varphi_x(\mathbf{z})$ prozesuen konbergentzia ziurtatuko duen **p** pauso-kopurua.

> *Egia esan, $\varphi_x(y)$ eta $\varphi_x(z)$ konputazio-prozesuek, bukatzeko ez dutela zergatik pauso-kopuru bera behar kontuan har genezake, eta honekin bat eginez, p_1 eta p_2 bi balio bila genitzake, bakoitzarentzat berea. Hala ere, eraginkortasunaren irizpideak programen exekuzioaren simulazioa beharrezkoa dena baino harantzago ez eramatea aholka diezagukeen arren, gure helburuetarako, diseinatu beharreko programaren testua laburtzea garrantzitsuagoa da. Kasu honetan, **T** eta **E** predikatuak, behar-beharrezkoak direnak baino pauso-kopuru handiagoetarako ere betetzen direnaz baliatuz, **p** bakar baten bilaketa errazagoa zaigu.*

y≠z den bitartean, **y** datuak, nahiz **z**-k, nahiz **p**-k edozein balio har dezaketenez, benetan egiten duguna (**y**,**z**,**p**) hirukoarentzat balio posibleak era sistematikoan pasatzea da, egokiak direnak aurkitu arte, existitzen badira behintzat. Baina hitzen eta hitz-hirukoteen artean kodeketa biunibokoa dagoela badakigu (ikus C eranskineko ***b*** atala), eta beraz (**y**,**z**,**p**) erako balizko hirukote guztiak korritzea hirukote-kode posible guztiak (**u** = kod^3(**y**,**z**,**p**) erakoak) korritzearen baliokidea da, baina abantaila nabarmen batekin: azken hauek $\sum^*$-ren hitzak dira soilik eta era sekuentzialean ordena daitezke. Horrela, **u**-ren banako osagaietara iristeko deskodeketa funtzioetara joko dugu, **y** = $\text{deskod}_{3,1}$(**u**), **z** = $\text{deskod}_{3,2}$(**u**) eta **p** = $\text{deskod}_{3,3}$(**u**) betetzen delako. Azaldu berri dugun balioak korritu eta bilatzeko teknikari dagokion programa honakoa izan liteke:

```
X0 := true;
U := ε;
AURKITUA := false;
while  not AURKITUA loop
-- U-k egokia den (y,z,p) balio-hirukotea osatzen duenentz egiaztatzen dugu
      Y := deskod_3_1(U);
      Z := deskod_3_2(U);
      P := deskod_3_3(U);
      if  Y<>Z and T(X1, Y, P) and T(X1, Z, P)  then
            AURKITUA := AURKITUA ∨ Φ(X1, Y) = Φ(X1, Z);
      end if;
      U := hur(U);
end loop;
```

Oraindik programa trinkoago bat egin dezakegu, nahi dugun **r** = $\varphi_{\mathbf{x}}$(**y**) = $\varphi_{\mathbf{x}}$(**z**) emaitza komun hori kalkulatu ordez, posible diren guztien artean "bilatu" egiten badugu, 4.10 irudiak azaltzen duen moduan.



Honakoa bete behar dute:

$$\mathbf{y \neq z} \ \wedge E(\mathbf{x}, \mathbf{y}, \mathbf{p}, \mathbf{r}) \ \wedge \ E(\mathbf{x}, \mathbf{z}, \mathbf{p}, \mathbf{r})$$

4.10 irudia: φ_x funtzioa injektiboa ez dela erabakitzeko, honakoak bilatzen ditugu: desberdinak diren **y** eta **z** balioak, $\varphi_x(\mathbf{y})$ eta $\varphi_x(\mathbf{z})$ prozesuak konbergiaraziko dituen **p** pauso kopurua, eta bietarako komuna den **r** emaitza. Hori egiteko era erraz bat (**y**,**z**,**p**,**r**) balio-laukote posible guztiak korritzea da, horretarako beren kod[4](**y**,**z**,**p**,**r**) kodeak erabiliz.

Honako programak aipatu dugun bilaketa-teknika inplementatzen du:

```
X0 := true;   U := ℰ;
while  not (deskod_4_1(U) <> deskod_4_2(U) and
      E (X1, deskod_4_1(U), deskod_4_3(U), deskod_4_4(U)) and
      E (X1, deskod_4_2(U), deskod_4_3(U), deskod_4_4(U)))  loop
            U := hur (U);
end loop;
```

eta bere egitura aurreko bi bertsioena baino askoz sinpleagoa da, nahiz eta erabilitako bukaera-baldintza konplexu samarra izan. Metodo honen funtsezko abantaila bere eskalabilitatea da: balio desberdinak aztertzeko ordena esplizituki programako kontrol-eskeman jarri behar ez izatean, teknika hau zailtasun gehigarri gutxirekin aplikatzen da dimentsio anitz duen bilaketa-eremua behar duten kasutan.

4.4.4 Funtzioen alderanzketa

Prozesu tartekatzearen teknikaren bidez beste gai interesgarri bati ekin diezaiokegu, konputagailu batek burututako eragiketen itzulgarritasunari, hain zuzen ere. Programa batek **x** datua **y** emaitzan eraldatzen badu, **y**-tik abiatuta **x** berreskuratzeko aukera izango al dugu?

Zenbait kasutan galdera hori egiteak ez du zentzurik. **P** programak edozein **x** datu **abbca** hitzan eraldatzen badu, ezin dugu prozesua alderantzikatzea

planteatu, emaitzan oinarrituta, hau zein **x** datu zehatzetik sortua den jakitea ezinezkoa baita (**P** programaren ekintzak informazioa galarazten duela esaten da). Egoera hau **P** programak funtzio ez-injektibo bat konputatzen duen guztietan errepikatuko da, eta kasu horietan **P**-ren portaera alderantzikatzeak ez du zentzurik izango.

Hortaz, gaia beste era batera har dezagun. Demagun funtzio injektibo bat (hots, datu desberdinetarako emaitza berdinak inoiz sortzen ez dituena) konputatzen duen **P** programa dugula. Orduan, **P**-ren prozesua alderantzikatuko duen **Q** programa, hots, **P**-k sortutako emaitzan oinarrituta bere jatorrizko datua berreskuratuko duen **Q** programa noiz aurki dezakegu? Erantzuna da beti: kasu hauetan, programa batek egiten duen guztia, beste batek desegin dezake.

TEOREMA (ALDERANTZIZKO FUNTZIOAREN LEGEA): Bedi ψ: $\sum^* \longrightarrow \sum^*$ konputagarri eta injektiboa den funtzioa. Orduan ψ^{-1}: $\sum^* \longrightarrow \sum^*$ funtzioa ere konputagarria da.

⌦ ψ funtzioa konputagarria bada, $\psi \cong \varphi_a$ betetzen duen **a** kodea existitzen da. $\psi^{-1}(\mathbf{y})$ konputatzeko $\psi(0)$, $\psi(1)$, $\psi(2)$, $\psi(3)$, …, kalkulatuko ditugu, $\psi(\mathbf{x}) = \mathbf{y}$ beteko duen **x** balioaren bila. Jakina, kalkuluak paraleloan egingo ditugu azkenean infinituak izan daitezkeen $\psi(\mathbf{x})$ konputazioen arriskua ekiditeko. Datuen eta pauso-kopuruen (**x**,**p**) bikote posibleak korritzeko kodeketa funtzioak erabiltzen baditugu, programa honakoa litzateke:

```
BIKOTE := ε;
while not E (A, deskod_2_1 (BIKOTE), deskod_2_2 (BIKOTE), X1) loop
      BIKOTE := hur (BIKOTE);
end loop;
X0:= deskod_2_1 (BIKOTE);  ⌫
```

5. Diagonalizazioa

Funtzio batzuen konputagarritasuna frogatu nahian oso emaitza garrantzitsuak lortu ditugu, hala nola, while-programen idazketa sinplifikatzen laguntzen diguten makroen eta datu-moten garapena, funtzio unibertsalarena bezalako programa konplexuak garatzeko gaitasuna, *prozesu tartekatzea* bezalako teknika aurreratuak erabilgarri izatea, edo beren programa zehatza eraikitzea zaila den funtzioetarako *konputagarritasun eskuragaitza*ren kontzeptua. Beste era batera esanda, edozein diziplina informatikoren portaera klasikoa imitatu dugu: while-programen tresna definitu dugu eta berarekin gero eta problema konplexuagoak ebazten aritu gara. Bere aukerak erabiltzen zenbat eta trebetasun handiagoa lortu, tresnari gero eta baliagarritasun handiagoa aurkitu diogu, eta, irudimena pixka bat astinduz, jakitun ala inkontzienteki, gure programekin ebazterik izango ez den problema praktikorik ez dagoela pentsatzera ere iritsi gara.

Hala ere, badira urte asko frogatu zela inolako metodo informatikorekin ebatzi ezin diren problemak badaudela, metodoak oso sofistikatuak izanda ere. Problema horiek konputaezinak dira eta ez dago programazio-lengoaiarik hori aldatuko duenik. Are gehiago, arraroa dirudien arren, problematan normala, hots, ohikoena, konputaezinak izatea da.

Konputagarriak ez diren funtzioei ekiteko ordua heldu da. Egingo dugun lehen gauza horrelako funtzioak existitzen direla era orokorrean frogatzea izango da, eta jarraian adibide zehatzak aztertuko ditugu. Azkenik, txosten honen funtsezko gaiaren xehetasunak emango ditugu: funtzioa konputagarria ez dela nola froga daiteke, ahalik eta sistematikoena den prozedura erabiliz. Kapitulu honetan ikusiko ditugun teknikak eta frogapenak, funtsean, eraikitzaileak ez diren arren, bukatzen dugunerako funtzioen oso klase zabalaren konputaezintasuna zehazteko gaitasuna lortua izango dugu.

5.1 Kardinalaren argumentua

Argumentu bakarra duten funtzio konputagarri guztien zerrenda dagoela badakigu, $\{\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots\}$ zerrenda, hain zuzen ere. Honek funtzio konputagarrien kopuruan ondorioak ditu, hau zenbaki transfinitua baita, existitzen diren funtzio guztien klase osoa hartzeko nahikoa ez dena.

PROPOSIZIOA 1: Infinitu funtzio konputaezin dago.

⌦ Gogora dezagun ikur-kateak hartu eta itzultzen dituzten funtzio guztien klaseari $\mathcal{F}$ deitu diogula, eta konputagarriak direnen klaseari, berriz, $\mathcal{C}$ (ikus 3.2

atala). Oraingo honetan arreta, argumentu bakarreko funtzioen $\mathcal{F}_1 \subset \mathcal{F}$ klasean eta hauen artean konputagarriak direnen $C_1 \subset C$ klasean jarriko dugu. C_1 klasea $\mathcal{F}_1$ baino txikiagoa dela frogatuko dugu, horretarako bere kardinala txikiagoa dela egiaztatuz.

Lehenik C_1 klase zenbagarria dela ikus dezagun. $\mathbb{W}$ motaren inplementazioari esker $C_1 = \{\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots\}$ betetzen dela ikusita dugu. Hortaz, badago osoa eta supraiektiboa den funtzio bat:

$$\mathbf{f}: \mathbb{N} \to C_1$$

edozein **i** zenbaki φ_i funtzio konputagarriarekin erlazionatzen duena. **f** funtzio hau ez da bijekzioa, funtzio konputagarriak zerrendan errepikatuta baitaude (adibidez, $\mathbf{f}(0) = \mathbf{f}(2)$ betetzen da, $\mathbf{P_0}$ (X0:= ε;) eta $\mathbf{P_2}$ (X1:= ε;) programek ε funtzio konstantea konputatzen dutelako). Horregatik ditugun funtzio konputagarriak ez dira zenbaki arruntak baino gehiago, hau da, $|C_1| \leq |\mathbb{N}|$ (berdintza ere ikus liteke, baina gure arrazonamenduarentzat garrantzirik gabekoa da).

Orain $\mathcal{F}_1$ zenbagarria ez den klasea dela egiazta dezagun. Horretarako, [0,1] tarte errealean osoa eta supraiektiboa den funtzio bat eraikiko dugu:

$$\mathbf{g}: \mathcal{F}_1 \to [0,1]$$

Argumentu bakarreko ψ funtzio bakoitzari $\mathbf{g}(\psi)$ zenbaki erreala dagokio, 2 oinarrian eta $0,\mathbf{d_0}\mathbf{d_1}\mathbf{d_2}\mathbf{d_3}\mathbf{d_4}\ldots$ erara idatzitako zenbakia, non dezimal bakoitza honela kalkulatzen den:

$$\mathbf{d_i} = \begin{cases} 0 & \psi(\mathbf{w_i})\uparrow \\ 1 & \text{bestela} \end{cases}$$

Adibidez, ψ funtzioa osoa bada, orduan $\mathbf{g}(\psi)=0,11111111\ldots$ betetzen da, hots, 1 balioa. Era berean, $\mathrm{dom}(\chi) = \{0\}$ eta $\mathrm{dom}(\xi) = \{\mathbf{x}: \mathbf{x} \bmod 2 = 0\}$ betetzen badira, orduan $\mathbf{g}(\chi)=0,1$ eta $\mathbf{g}(\xi)=0,1010101\ldots$. Lehen kasuan $1/2$ zenbakiaren adierazpide bitarra dugu eta bigarrengoan $2/3$ zenbakiarena, honako adierazpenak egiaztatzen duenez:

$$\frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{128} + \ldots = \sum_{\mathbf{i}=0}^{\infty} \frac{1}{2 * 4^{\mathbf{i}}} = \frac{2}{3}$$

Argi dago [0,1] tarteko edozein **r** zenbaki erreal, dezimal bitarren segida moduan $(0,\mathbf{d_0}\mathbf{d_1}\mathbf{d_2}\mathbf{d_3}\mathbf{d_4}\ldots)$ jar daitekeela, eta segida hori emanda, $\mathbf{g}(\psi) = 0,\mathbf{d_0}\mathbf{d_1}\mathbf{d_2}\mathbf{d_3}\mathbf{d_4}\ldots = \mathbf{r}$ betetzen duen ψ funtzioa existitzen dela. Beraz, **g** funtzioa supraiektiboa da eta ondorioz $|[0,1]| \leq |\mathcal{F}_1|$ betetzen da (berriro ere berdintza ikus genezake, baina gure arrazonamenduarentzat garrantzirik gabekoa

da). Baina [0,1] tartean $\mathbb{R}$ multzoan bezainbat zenbaki erreal daudela badakigu, eta ondorioz $|\mathbb{R}| \leq |\mathcal{F}_1|$ betetzen da.

Hala eta guztiz ere, Cantor-ek frogatu zuenetik, badakigu $|\mathbb{N}|$ kardinal transfinitua $|\mathbb{R}|$ baino hertsiki txikiagoa dela, eta ondorioz:

$$|\mathcal{C}_1| \leq |\mathbb{N}| < |\mathbb{R}| \leq |\mathcal{F}_1|$$

Beste era batera esanda, while-programen kardinala zenbagarria denez, funtzio konputagarrien kopurua ere hala da. Alabaina, existitzen diren funtzioen multzoa ez da zenbagarria, eta hortaz, denak konputatzea ezinezkoa da: nahiko programarik ez dago. ⌫

Izan ere, $|\mathbb{R}| \geq 2^{|\mathbb{N}|}$ denez, existitzen diren funtzioetatik gehien-gehienak ez dira konputagarriak, eta beraien kopurua konputagarriena baino askoz handiagoa da. Gainera, hau imajina dezakegun edozein programazio-lengoaiarentzat baliozkoa da: programak alfabeto finitu (edo infinitu zenbagarri) baten ikurren bidez adierazten diren bitartean, eskura izango dugun programa guztien multzoa beti zenbagarria izango da, eta hortaz, balizko funtzio guztien klase osoa hartzeko nahikoa ez da izango.

Aurreko arrazoiketak, oso erantzunezina iruditu arren, konputaezintasunaren izaeraren inguruan ez digu ezer eransten: funtzio konputaezin asko daudela esaten digu, baina ez non dauden ezta zein itxura duten ere. Hala ere, ikuspuntu praktikotik, zenbat dauden jakitea ez zaigu interesatzen, ezta adibide batzuk ezagutzera mugatzea ere, baizik eta, funtzio bat emanik, hau konputaezina denentz zehazteko metodoren bat edukitzea. Funtzioa konputaezina bada, jakin egin beharko genuke, berau programatzen saiatuz denbora galtzea ekidingo baikenuke.

> *30eko hamarkada izan zen lehen problema konputaezinak aurkitu ziren garaia, hau da, konputagailuak oraindik etorkizunerako promesa ere ez zirenean. Hauek eraikitzeko teknologia eskura ez zegoen arren, konputagailu digitalek hamabost urte geroago erakutsiko zituzten propietateetako asko ezagunak ziren. Horrela, jakina zen, bere ahalmena handiena izanda ere, konputagailurik ez zela egongo, exekutatzen ari ziren programen portaera aurreikusteko gai izango zenik, edo bi programa emanda hauek baliokideak ziren (hots, hasierako baldintza berdinekin lan eginez gero emaitza berdinak sortzen zituzten) erabakitzeko aukera izango zuenik.*

Problema baten konputagarritasuna frogatzen saiatzen garenean berau ebatziko duen algoritmo zehatz bat aurkitzea nahikoa da. Aldiz, bere konputaezintasuna frogatzeko atazari ekitean, algoritmo hori ez dela existitzen, edo beste era batera esanda, existitzen diren algoritmoetatik bat bera ere problema hori

ebazteko gai ez dela ziurtatuko duen argumentu logikoa behar dugu. Izaera unibertsaleko horrelako argumentua ezartzea ez da batere lan erraza, eta oro har, absurdura eramandako frogapen batekin erlazionatuta dago. Helburu hori lortzeko teknika bat baino gehiago dago, eta garrantzitsuenak diagonalizazioarena, laburtzearena eta puntu finkoarena dira. Lehenengoa da kapitulu honetan landuko dugun bakarra.

Berriro ere, funtzio konputagarri guztien $\mathcal{C}_1 = \{\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots\}$ zerrendatzea existitzea izango da konputaezintasunerako frogapenek edukiko duten oinarria, baina oraingo honetan era eraikitzaileago batean. Funtzio bat konputagarria ez dela frogatu ahal izango dugu hau aipatu zerrendan ez dagoela egiaztatuz. Gainera funtzio konputaezin batzuk definitu ahal izango ditugu, zerrendan dauden guztietatik desberdintzen diren eran eraikiz.

5.2 Geratze-problema

Teknika algoritmikoen bitartez ebatzi ezin diren problemen artean, **geratze-problema** da lehengoa eta ezagunena, eta horregatik *Konputagarritasun Teoria*ko edozein liburutan derrigorrezko kapitulua [Har 87, MAK 88, SW 88]. Gainera erlazionatutako beste problema batzuen konputaezintasuna aurreratzeko euskarri intuitibo bat ere ematen digu.

Aurreko ataletan while-programen ezaugarrien gainean lan egiten duten funtzioen adibideak ikusi genituen. Horiek propietate estatikoak izan zitezkeen, while-programaren sintaxiari lotutakoak (esaterako *azken_aldagaia*), baina while-programaren semantikarekin edo exekuzioarekin erlazionatutako propietate dinamikoak ere izan zitezkeen (funtzio unibertsalaren kasua adibidez). Geratze-problema adierazten duen predikatua azken funtzio horietakoa da, **x** eta **y** sarrera-balioetatik abiatuz, $\mathbf{P_x}$ programak **y** datuaren gainean exekutatzean konbergitzen duenentz jakin nahi baitu.

TEOREMA (GERATZE-PROBLEMAREN ERABAKIEZINTASUNA): Ondoren definitzen dugun *halt*: $\sum^* \times \sum^* \rightarrow \mathbb{B}$ funtzioa ez da konputagarria

$$\text{halt}(\mathbf{x}, \mathbf{y}) = \begin{cases} \text{true} & \varphi_{\mathbf{x}}(\mathbf{y}) \downarrow \\ \text{false} & \varphi_{\mathbf{x}}(\mathbf{y}) \uparrow \end{cases}$$

Teorema honen ondorioz while-programen geratzea erabakitzeko *metodo orokorrik* (eta beraz, inolako programazio formalismorik) ez dago.

⌦ Frogapena, absurdura eramanez egingo dugu. Demagun *halt* funtzioa konputagarria dela. Orduan bera kontuan hartuz $\delta : \sum^* \longrightarrow \sum^*$ funtzioa definituko dugu

$$\delta(\mathbf{x}) \cong \begin{cases} \varepsilon & \neg\text{halt}(\mathbf{x},\mathbf{x}) \\ \perp & \text{halt}(\mathbf{x},\mathbf{x}) \end{cases}$$

eta hau ere konputagarria aterako zaigu, honako programak frogatzen duenez:

if halt(X1, X1) **then** X0 := ⇊ ; **else** X0 := ε; **end if**;

δ konputagarria izateagatik, $\delta \cong \varphi_{\mathbf{e}}$ betetzen duen **e** indizea[5] existitzen da. $\delta(\mathbf{e})$-ren balioa kalkulatzea plantea dezakegu, δ-ren izaera kontuan hartuz konbergentea nahiz dibergentea izan daitekeena. Ikus ditzagun bi aukera horiek:

$$\underline{\text{1. kasua:}}\ \delta(\mathbf{e})\downarrow \overset{\delta\cong\varphi_e}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{e})\downarrow \overset{\text{def. halt}}{\Rightarrow} \text{halt}(\mathbf{e},\mathbf{e}) \overset{\text{def . }\delta}{\Rightarrow} \delta(\mathbf{e})\uparrow \ \lightning$$

kontraesanean dagoena, eta beraz ezinezkoa. Bestalde

$$\underline{\text{2. kasua:}}\ \delta(\mathbf{e})\uparrow \overset{\delta\cong\varphi_e}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{e})\uparrow \overset{\text{def. halt}}{\Rightarrow} \neg\text{halt}(\mathbf{e},\mathbf{e}) \overset{\text{def . }\delta}{\Rightarrow} \delta(\mathbf{e}) = \varepsilon \Rightarrow \delta(\mathbf{e})\downarrow \ \lightning$$

kontraesanean dagoena ere. Labur esanda, $\delta(\mathbf{e})\downarrow \Leftrightarrow \delta(\mathbf{e})\uparrow$ baliokidetasunera iritsi gara, eta honek esan nahi du δ funtzioak, **e** sarreraren gainean aplikatzean, ezin duela ez konbergitu, ez dibergitu ere egin. Kontraesan honek hipotesia nahitaez faltsua izan behar dela, eta hortaz *halt* funtzioa ezin dela konputagarria izan, ondorioztatzera garamatza. ⌫

Aurretik azaldu dugun moduan, programen egikaritzapenek zein kasutan konbergituko duten eta beste zein egoeratan dibergituko duten aurretik erabakitzeko metodo orokorrik (hots, kasu guztietarako balioko duen metodorik) ez dagoela zehazten du, Informatika Teoriko osoaren garrantzitsuenetakoa den teorema honek.

> *Programa bat bukatuko denentz askotan aurreikusteko gai izango garen arren, hau ez da aurreko ondorioarekin kontraesanean egongo. Adibidez, begiztarik ez daukan programa bat, edozein sarrera emanda ere, gelditu egingo dela zehazteko ez da oso azkarra izan beharrik. Bestalde, ondorengo programaren azaleko analisiak:*
>
> **while** $\text{car}_{\mathbf{a}}$?(X1) **loop** X1:= $\text{cons}_{\mathbf{a}}$(X1); **end loop**;
>
> ***a*** *ikurraz hasten ez diren hitzetarako bakarrik konbergitzen duela aurkitzeko aukera emango digu. Baina hori jakiteko jarraitutako metodoa ez da edozein programatarako orokortzerik izango, eta hori da,*

[5] Balizko indize bat, konputagarritasuna frogatzeko balio izan digun makroprograma hedatzean lortuko genukeen while-programaren kodetzea da.

hain zuzen, dugun arazoa. Izan ere, badaude programa erraz samar batzuk konbergitzen dutenentz ziur ez dakigunak. Horien adibidea honako programa da:

```
while X1<>1 loop
    if X1 mod 2 = 0 then  X1:= X1 / 2;
    else  X1:= 3*X1 + 1;
    end if;
end loop;
```

Bere eragiketa-sekuentzia eta emaitza argi samarrak daude (konbergitzen duenean beti 1 itzultzen du), baina itxuraz portaera alderraia eta aurretik jakinezina du. Sarrera batzuentzat (esaterako, 2-ren berredura direnentzat) oso azkar konbergitzen duen bitartean, beste batzuentzat zenbait bira eman behar du (43 iterazio 39. zenbakiarentzat, adibidez). Zenbakizko datu askotarako bukatzen duela egiaztatu den arren, guzti-guztietarako egiten duenik ez da frogatzerik izan, nahiko matematikari aditu saiatu izanagatik ere. Geratze-problema benetan erabakiezina dela guztiz sinetsita ez dauden horientzat antidotorik onena programa horren portaera aurreikusten saia daitezela da, zein kasutarako konbergitzen duen eta zeintzuetarako ez, arakatuz.

5.3 Paradoxak eta diagonalak

Geratze-problemaren erabakiezintasuna frogatzean egindako arrazonamenduaren inguruan hausnarketa egin dezagun orain. Frogapenaren eskema nahiko zuzenekoa da. Lehenik *halt* funtzioa konputagarria dela suposatzen dugu. Gero δ funtzioa definitzen dugu, zeinen konputagarritasuna *halt*-enatik ondorioztatzen den. Azkenik, δ funtzioan portaera kontraesankorra aurkitzen dugu, funtzioa bere **e** indizearen gainean aplikatzean, eta gure hasierako suposaketa faltsua zela ondorioztatzen dugu.

Argi dirudi frogapenaren kako nagusia, zuzenean kontraesanera garamatzan δ funtzio misteriotsua dela. Baina funtzio horrek kontraesankorretik zer dauka? Hori ikusteko hausnarketa egitea komeni da. While-programek portaera konbergentea nahiz dibergentea eduki dezaketela badakigu, eta gainera, portaera hori sarreran emandako datuaren arabera alda daitekeela. Baina datu hori, bere aldetik, while-programa bat bezala interpretatzen denean, while-programek beste programa batzuen gainean konbergitzen edo dibergitzen dutela esateak zentzua badu. Adibidez, honako **P** programak:

```
X0 := ⇊;
```

sarrera bezala ematen zaion edozein programaren gainean dibergitzen du, eta aldiz **Q** programak:

```
X0 := konposaketa_eratu (X1, X1);
```

sarreran emandako edozein programaren gainean konbergitzen du. Portaera pixka bat konplexuagoa **R** programak erakusten duena da. Honek datu moduan ematen zaion while-programa arakatzen du, begizta bat aurkitu nahian. Aurkituz gero horrek osatutako azpiprograma itzultzen du, eta bestela ziklatu egiten du:

```
PILA :=  <];
PILA := pilaratu (X1, PILA);
AURKITUA := false;
-- AURKITUA aldagaiak datuaren egituran while_loop aurkitu den adierazten du
while not  P_hutsa? (PILA) and not AURKITUA  loop
        PROG := tontorra (PILA);
        PILA := despilatu (PILA);
        if konposaketa? (PROG) then
                PILA := pilaratu  (barne_agindu_2 (PROG), PILA);
                PILA := pilaratu (barne_agindu (PROG), PILA);
        elsif baldintzazkoa? (PROG) then
                PILA := pilaratu (barne_agindu (PROG), PILA);
        elsif iterazioa? (PROG)  then
                AURKITUA := true;
                X0 := PROG;
        end if;
end loop;
if  not AURKITUA  then   X0 := ⇊;
end if;
```

Emango dugun azken adibidea **S** programak aurkezten duena da. Honek, datu moduan emandako while-programak 12 sarrerarekin daukan portaera aztertzen du. Itzultzen den emaitza 5 denean bakarrik geratzen da, beste edozein kasutan dibergitu egiten du:

```
R := Φ(X1, 12);
if  R=5  then
        X0 := 0;
else X0 := ⇊;
end if;
```

Programa hauek sarreran ematen zaien datuaren arabera gelditzen direnez eta datu hori bere aldetik while-programa bat denez, honako jakinmina eduki dezakegu, zer gertatzen da ematen den datua *programa bera* denean? Edo, beste era batera esanda, programa hauek beren buruaren gainean konbergitzen al dute? **P** eta **Q** programen kasuan erantzuna erraza da: **P**-k ez du inoiz konbergitzen (noski, bere kodearen gainean ere ez) eta **Q**-k funtzio oso bat inplementatzen duenez, bere buruaren gainean ere konbergitzen du.

Puntu honetan zehaztapen bat egitea komeni da. **P** eta **Q** programek jasotzen dituzten datuak $\mathbb{W}$ datu-motari dagozkion while-programak dira.

Makroprogramak ez dira inplementatuak izan eta, beraz, ezin dira hitzen bitartez adierazi eta ezin dute datu bezala funtzionatu. Horregatik, zorrotzean hartuta, **P** programak ezin du ez konbergitu ezta dibergitu ere. Hortaz, "**P**-k bere buruaren gainean dibergitzen du" esaten dugunean, berez esan nahi dugu "**P**-ren hedapenaren ondorioz sortzen den while-programak bere buruaren gainean dibergitzen du".

R programaren kasuan ere bere buruaren gainean konbergitzen duela daukagu, **R**-k bere hedapenean ere azalduko den while_loop motako agindua duelako. **S**-ren kasua pixka bat bihurriagoa da. **S**-k dibergitu edo 0 balioa eman dezakeenez soilik, 5 emaitza ez duela inoiz sortuko argi dago, ez 12 sarreraren gainean ezta beste edozeinen gainean ere. Hortaz **S**-k bere buruaren gainean dibergitzen du.

Hortik atera dezakegun ondorioa edo ikasketa da programak bi motatan sailka ditzakegula: bere buruaren gainean konbergitzen dutenena eta ez dutenena. Baina, aurreko ataleko frogapenean erabilitako δ funtziora itzulita, arretaz aztertuz gero, bere funtzionamendua sailkapen bitxi horretan oinarritzen duela dirudi, bere buruaren gainean konbergitzen ez duten programetan aplikatzen denean bakarrik konbergitzen baitu:

$$\delta(\mathbf{x}) \cong \begin{cases} \varepsilon & \varphi_x(\mathbf{x})\uparrow \\ \perp & \varphi_x(\mathbf{x})\downarrow \end{cases}$$

Hau da, δ funtzioak **Q** edo **S** bezalako programen gainean konbergituko du eta **P** edo **R** bezalako beste batzuen gainean dibergituko. Baina jarraitzen dugun arrazonamenduak δ funtzioaren konputagarritasuna frogatzera garamatzanez, jauzi kualitatibo garrantzitsua ematen ari gara: δ funtzio bitxi samarra moduan ez ezik, $\mathbf{P_e}$ programaren itxurarekin ere ikus dezakegu, $\mathbf{P_e}$, lehen esan bezala, "bere buruaren gainean konbergitzen ez duten programetan aplikatzen denean bakarrik konbergitzen duen" while-programa izanik. Honaino iritsita, honako galdera planteatzera bultzatzen gaituen jakinmina guztiz bidezkoa dirudi: zer gertatzen da $\mathbf{P_e}$ programa bere buruaren gainean, hots, bere **e** indizearen gainean aplikatzen dugunean?

Alde batetik, $\mathbf{P_e}$ programa, **P** edo **R** bezala, bere buruaren gainean konbergitzen duten horietakoa izatea gerta liteke. Baina orduan bera sarrera bezala hartuta, $\mathbf{P_e}$ dibergiarazten duten sarreretako bat da. Beraz, $\mathbf{P_e}$ programa **e** sarrerarekin geratu egiten bada, orduan **e**-ren gainean ziklatzera behartuta dago.

Baina beste alternatiba ere ez da balizkoa. $\mathbf{P_e}$ programa, **Q** edo **S** bezala, bere buruaren gainean dibergitzen duten horietakoa balitz, orduan, bera sarrera moduan

hartuta, $\mathbf{P_e}$ konbergentea egingo luketen sarreretako bat izango litzateke. Hortaz, $\mathbf{P_e}$ programak **e** sarrerarekin ziklatzen badu, orduan **e**-rekin gelditzera behartuta dago.

Beraz $\mathbf{P_e}$ programak bere buruaren gainean ezin du ez konbergitu ezta dibergitu ere, edozer egiten duela ere kontrakoa egitera behartuta baitago. Irteerarik gabeko paradoxa baten aurrean gaude. Baina nola sartu gara bertan? Bere izaera kontraesankorra dela-eta existitu ezin den $\mathbf{P_e}$ programa zoroa sortu zaigulako. Nondik sortu da halako mamua? δ funtzioaren konputagarritasunaren zuzeneko ondorioa da. Eta konputagarritasun hori nola ondorioztatu dugu? *halt* funtzioa konputagarria zela suposatzeagatik. Paradoxatik irteteko bide bakarra bertara eraman gaituen suposaketa bertan behera uztea da, hau da, *halt*-en konputagarritasunaren hipotesia baztertzea, hori baita oinarririk ez duen asertzio bakarra gure arrazonamenduan.

> *Paradoxak sortzeko bide hau ez da Konputagarritasun Teoriaren berezko zerbait. [Gar 83] erreferentzian honen oso antzekoak diren beste zenbait aurki ditzakegu, Bertrand Russell-ek formulatu zuen bizarginarena, esaterako. Bere bizarra egiten ez duten pertsona guztiei, eta horiei bakarrik, bizarra kentzen dien bizarginaren existentzia planteatzeak sortzen dituen ondorioak aztertzen ditu paradoxa honek. Bizargina kontraesanean erortzen da bere buruarekin zer egin erabaki behar duenean. Bere bizarra egitea erabakitzen badu "eta horiei bakarrik" baldintza ez luke beteko, eta bere bizarra ez kentzea erabakiz gero, berriz, "bere bizarra egiten ez duten pertsona guztiei" ez luke beteko. Paradoxa honek joan den mendean garatu zen Multzo Teorian berebiziko garrantzia du.*
>
> *Paradoxa guztien eraiketa, multzo bat emanik, bertakoa den elementu berezi bat definitzean datza. Elementu hori, gainontzekoak, ezaugarriren baten arabera, bi azpimultzotan bereizteko gai da eta hori kontuan harturik bere portaera erabakitzeko ere. Aipatu portaera kasu bakoitzean betetzen den ezaugarriaren kontrakoa izan beharko da. Bere burua bi azpimultzotatik zeinetan kokatu behar duen erabakitzeko unean sortuko da paradoxa. Aukeratutako azpimultzoa edozein dela ere, erakutsi behar duen portaerarekin kontraesanean egongo da.*

Definitu dugun δ funtzioa nahiko berezia da. Bere eraiketa φ_0, φ_1, φ_2, φ_3, … funtzio konputagarri guztien desberdina izateagatik bereizten da. Izan ere, edozein φ_x funtzio konputagarri emanda, bi aukera ditugu:

a) $\varphi_x(\mathbf{x})\downarrow$ izatea gerta liteke. Orduan, δ eraiki da $\delta(\mathbf{x})\uparrow$ izateko

b) $\varphi_x(\mathbf{x})\uparrow$ izatea gerta liteke. Orduan, δ eraiki da $\delta(\mathbf{x})\downarrow$ izateko

Beste era batera esanda, δ ezinbestez φ_x funtzio konputagarri bakoitzaren desberdina dela, **x** puntuan zehazki (ikus 5.1 irudia). Beraz, δ funtzioa ezin da $\mathcal{C}_1 = \{\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots\}$ zerrendan egon eta ondorioz konputaezina izan behar da.

δ funtzioa eraikitzeko erabilitako mekanismoa ***diagonalizazioa*** deitzen da eta lehenengo aldiz Georg Cantor-ek erabili zuen (beste batzuen artean, $|\mathbb{N}| < |\mathbb{R}|$ frogatzeko, 5.1 atalean erabili dugun emaitza, hain zuzen ere). Metodo hau aplikagarria da objektuen zerrenda infinitu bat izanik, horietako bakoitza bere aldetik azpiobjektuen zerrenda infinitu baten bidez definitzen denean (gure kasuan objektuak funtzio konputagarriak dira eta azpiobjektuak, funtzio bakoitzak datu posibleen gainean dituen portaerak). Baldintza horietan zerrendako guztien desberdina den objektu berri bat eraikitzea posible da.

δ	**x**	φ_0	φ_1	φ_2	...	φ_x
$\uparrow$	**0**	$\downarrow$				
$\downarrow$	**1**		$\uparrow$			
$\uparrow$	**2**			$\downarrow$		
...	...				...	
$\uparrow$	**x**					$\downarrow$

5.1 irudia: δ funtzioa konputagarri guzti-guztien desberdina da, δ-ren eta φ_x-ren portaerak zehazki **x**-ren gainean kontrakoak direlako. Adibide-taula honetan[6] egoera hori ikus daiteke.

Edozelan ere, eraiki dugun δ funtziotik garrantzitsuena ez da konputaezina izatea. Funtsezkoena bere izaera kontraesankorra dela da, eta bere kontraesanak *halt*-en konputagarritasunaren hipotesia berarekin batera arrastan daramala. Eta zergatik izaera kontraesankor hori? δ funtzioa, konputaezina izan dadin, kontu handirekin eraiki dela ikusi dugu. Baina, bestaldetik, funtzio horrentzat programa bat eraiki dugu, bertan *halt*-ek oso paper nabarmena izanik. Beraz, δ funtzioak izaera bikoitza du:

[6] Atal honetan erabiliko ditugun adibide-tauletan, emandako datuak sinesgarriak izan daitezen ez dugu arreta berezirik jarriko, aldiz, aztertu beharreko aukera desberdinak erakusteko balio izatea da bilatuko duguna. Adibidez, badakigu $\varphi_1(1)\uparrow$ ez dela egia (P1 ≡ X0:=cons$_a$(X0); delako). Hala ere, $\varphi_x(x)\uparrow$ betetzen den kasu bat aztertzea interesatzen zaigu, eta hori betetzen duen lehenengo programa erreala topatu arte taula hedatzea ez da arrazoizkoa (bi ikurreko alfabeto batekin lan eginez gero programa hori honakoa litzateke: P36 ≡ **while** nonem?(X1) **loop** X0:=ε; **end loop**;). Irtenbidea programen portaera erreala "ahaztea" eta hauei ausaz edo nahi dugun eran kasu desberdinak esleitzea da, horrela adibide moduan balio dezaten.

- Alde batetik konputaezina da (funtzio konputagarri guztien desberdina).
- Bestaldetik konputagarria da (bere konputagarritasuna *halt*-en ustezko konputagarritasunetik ondorioztatzen da).

Bi faktore horiek biltzeak sortzen du kontraesana, guk nahi genuen tokira garamatzan kontraesana, hain zuzen ere. Prozesu honetan pasa ditugun mugarri desberdinak laburbilduko ditugu:

1. *halt* funtzioa konputagarria dela suposatu dugu.

2. Aurrekoaren arabera, behar dugun guztietan *halt*(**x**,**y**) egia ote den galdetu ahal izango dugu. Edozein funtzio konputagarriri buruz *halt*-ek ematen duen informazioan oinarrituta, δ funtzio berria eraiki dugu.

3. Eraiketa horren mamia, δ funtzioa φ_x funtzio konputagarri bakoitzetik puntu zehatz batean desberdintzeko moduan definitzean datza (gure kasuan, desberdintasunerako puntu zehatz hori **x** bera da, baina hala izatea ez da derrigorrezkoa).

4. δ konputagarria dela frogatzen dugu, eta gainera bere konputagarritasuna *halt*-enaren menpekoa dela.

5. δ konputagarria denez, **e** indizea esleitzen diogu.

6. δ eta φ_e funtzioak desberdindu behar diren puntu zehatza bilatzen dugu.

7. δ funtzioa puntu horren gainean aplikatzen dugu eta δ bere buruaren desberdina izan behar dela ondorioztatzen dugu. Honek kontraesana osatzen du.

5.4 Diagonalizazio teknika

Geratze-problemaren erabakiezintasuna frogatzeko jarraitu dugun diagonalizazio metodoa oso erraz orokor daiteke beste konputaezintasun frogapen batzuetan aplikatzeko. Bere konputaezintasuna frogatu nahi den **f** funtzioa emanik, metodoa ***funtzio diagonala*** deitzen den δ beste funtzio bat eraikitzean datza. Funtzio horrek izaera kontraesankorra du: alde batetik konputagarria izango da, **f** ere hala dela suposatuz, baina bestetik, funtzio konputagarri guztien desberdina izango da, kontu handiz horrela eraikiko dugulako. Baina, metodoaren alderdi teknikoenak aplikatzeari ekin baino lehen, problema eta honen ebazpenean jarraitu beharreko faseak kontu handiz aztertzea beharrezkoa da. Hoberena ahal den erarik metodikoenean egitea da, pauso zailenak identifikatzeko eta irteerarik gabeko bideak hartzea ekiditeko.

Jarraian teknika honen lau faseak zehaztuko ditugu eta aldi berean xehetasunak argitzen joango gara, honako funtzioaren konputaezintasuna frogatzeko metodoa nola erabili erakutsiko digun adibidearen bitartez:

$$\mathbf{f}(\mathbf{x},\mathbf{y}) = \begin{cases} \text{true} & \varphi_{\mathbf{x}}(\mathbf{y}) = \text{'}\mathbf{aba}\text{'} \\ \text{false} & \text{bestela} \end{cases}$$

Funtzio hau, edozein $\mathbf{P_x}$ programak edozein **y** daturekin aplikatzean emaitza bezala **aba** hitza sortzen ote duenentz zehazten duena da.

> **1. FASEA – JUSTIFIKAZIOA**. **f** konputagarria ez denaren susmoa oinarritzeko arrazoiak ba ote dauden pentsatu behar dugu, era intuitiboan.

Ezertan hasi baino lehen, **f**-ren izaerari buruzko pistaren bat edukitzea garrantzitsua da. Kontuan hartu konputagarritasun frogapenek (esaterako, while-programa baten diseinuaren bidez egindakoek) eta konputaezintasunarenek beraien artean inolako loturarik ez dutela. Hori dela eta, hartu beharreko bidean asmatzea komeni da, bestela, egindako lan guztia alferrikakoa izango baita. Azkenean **f** konputagarria izatea gertatuko balitz, diagonalizazio teknika aplikatzeko egindako saio guztiek huts egingo dute, eta gainera **f** konputatuko duen programa nondik nora joan daitekeenari buruz inolako pistarik[7] ez digute emango. Eta hortaz, hasieran bezala egongo gara, baina nekatuagoak.

Aurreko funtzio konputaezin batzuekin lortutako esperientzia izango da, **f**-ren konputaezintasunaren arazoak nondik etor daitezkeen, eta bereziki funtzio honek geratze-problemarekin izan dezakeen erlazioa, susmatzera lagun gaitzakeena.

ADIBIDEA: Esplizituki bi kasu erakusten dituen arren, berez **f**-k hirugarren bat ere ezkutatzen du:

- Baldin $\varphi_{\mathbf{x}}(\mathbf{y})=\mathbf{aba}$, $\mathbf{P_x}$ programa **y** datuaren gainean egikarituz egiazta genezake eta programa bukatzean $\mathbf{f}(\mathbf{x},\mathbf{y})$=true dela jakingo genuke.
- Baldin $\varphi_{\mathbf{x}}(\mathbf{y})\neq\mathbf{aba}$ (hots, $\mathbf{P_x}$-k **y**-ren gainean **aba** ez den emaitza sortzen badu), programa egikarituz egiazta genezake ere, eta bera bukatzean $\mathbf{f}(\mathbf{x},\mathbf{y})$=false dela jakingo genuke.
- Arazoa $\varphi_{\mathbf{x}}(\mathbf{y})\uparrow$ denean sortzen da. Kasu honetan $\mathbf{P_x}$ programa **y**-ren gainean egikaritzeak ez digu ezer adierazten, *halt* funtzioa erabakiezina

[7] Egoera horrek ez dauka zergatik Logikan edo Matematikan ohikoena izan beharrik. Askotan, hala ez den propietate bat egia dela frogatzen saiatzean, gure saio oker horiek pistaren bat ematen digute, propietatea ez betetzearen arrazoiaren ingurukoa eta baita faltsua dela nola egiazta dezakegunaren ingurukoa ere.

denez ezin baitugu aurresan programa ez dela bukatzen nahiko denbora eman ez zaiolako ala indefinituki ziklatzen duelako. Kasu honetan **f**(**x**,**y**)=false da φ_x(**y**)=**aba** ez baita egia, baina ez dirudi hori zehazteko metodorik dagoenik.

Ondorioz, ez dirudi funtzioa konputagarria denik, baina kontuz! aurkeztu dugun argudiaketak ez du probarik osatzen (hala balitz, metodoaren gainontzekoa sobera egongo litzateke). Gurea bezalako arrazonamendua iruzurra izan liteke: esaten dugu **f**(**x**,**y**) ez dirudiela konputagarria, "φ_x(**y**)$\uparrow$ zein kasutan betetzen den erabaki beharko genukeelako eta hori ezinezkoa delako". Hala ere, ezin ebatzizko problema horren soluzioa atera beharrik izan gabe, **f** konputatzeko beste biderik edo era alternatiborik ez dagoela berma al dezakegu? Ziurtasuna lortzeko era bakarra frogapena formalki osatzea da.

2. FASEA – PLANIFIKAZIOA. **f** funtzioak ematen digun informazioa xehetasun osoz aztertu behar dugu. Gero, informazio hori oinarrian harturik δ funtzio diagonalaren eraiketa planifikatuko dugu: φ_x funtzio konputagarri guztien desberdina izaten saiatuko gara, eta bere konputagarritasuna **f**-renetik ondorioztatuko da.

Diagonalizazioaren argudiaketaren mamia **f** konputagarria dela suposatzea da eta horrek ezinezkoa den objektu baten eraiketara garamatzala frogatzea. Eraikiko den objektu hori δ funtzioa izango da, aldi berean konputagarria eta konputaezina izango dena. Konputagarri guztien desberdina izango den funtzio diagonala eraikitzea oso erraza da. Zaila **f**-rekin estuki erlazionaturiko δ funtzioa eraikitzea da, hots, δ funtzioaren konputagarritasuna **f**-renetik deduzitzeko aukera izateko moduan eraikitzea, horrela δ-ren izaera kontraesankorrak **f** berarekin batera arrastaka eramango baitu.

Hori dela eta, δ eraikitzeko erabili behar dugun tresna edo azpierrutina nagusia **f** izan behar dela planteatzea ezinbestekoa da. Funtzio diagonalaren eraiketari ekin baino lehen tresna horrek eskaintzen dituen aukerak sakonean ulertu behar ditugu. δ-ren definizioan, konputaezina izan daitekeenaren susmoa dugun beste edozein elementu sartzea ekidin behar dugu, bestela, δ-ren konputagarritasuna ezin dela mantendu egiaztatzen dugunean, ezingo baitiogu **f**-ri errua bota.

φ_x funtzio konputagarri desberdinei buruz **f**-k ematen digun informazioa analizatzeko eta ondoren, δ funtzioa konputagarri guztietatik desberdintzea lortzearren hain zuzen, informazio hori erabil ahal izateko 5.1. irudian erakusten dugun eskemaren antzekoak erabiliko ditugu. Funtzio askoren informazioa edukiko dugunez, hau ***adibide-taula*** batean antolatzea erabilgarria izango da.

Taula infinitu honen zutabeen goiburukoak φ_0, φ_1, φ_2, φ_3, φ_4, … funtzio konputagarriek beteko dituzte. Lerroen goiburukoek, funtzio horien sarreran[8] eman daitezkeen balioak adierazten dituzte 0, 1, 2, 3, ... Beraz, taulako gelaxka bakoitza φ_i funtzioak **j** datuaren gainean itzulitako emaitzari dagokio eta bertan **f** funtzioaren laguntzari esker $\varphi_i(\mathbf{j})$-ri buruz jakin dezakeguna adierazten saiatuko gara. **f** funtzioak zer ematen digun ideia garbia edukitzen dugunean, **f** hori δ funtzio diagonalaren eraiketan erabiltzeko prest egongo gara.

δ funtzioa taulan azaltzen diren guztien desberdina izan dadin horietako bakoitzetik, hots, zutabe bakoitzetik, gutxienez puntu batean desberdindu behar da. Puntu hori era desberdinetan aukera liteke, baina sinplifikatzearren, ahal den guztietan ***diagonalekoa*** hartuko dugu. Hortaz, δ eta φ_i funtzioak **i** puntuan desberdintzen saiatuko gara. Horretarako, **f**-ri esker $\varphi_i(\mathbf{i})$-ri buruz zer dakigun taulan begiratuko dugu eta $\delta(\mathbf{i})$ definituko dugu, $\varphi_i(\mathbf{i})$-ren portaera horrekin bateraezina izateko moduan.

ADIBIDEA: Gure adibidean praktikara eramanez, **f** funtzioak zein informazio ematen digu? φ_i funtzio bakoitzeko eta **j** datu bakoitzeko, $\varphi_i(\mathbf{j})$-ren emaitza **aba** den ala ez esaten du. Azken kasu horretan, baina, ez digu esaten arrazoia $\varphi_i(\mathbf{j})$ beste balio bat izatea den ala indefinituta egotea. 5.2. irudiak kasu honetarako adibide-taula erakusten du.

	φ_0	φ_1	φ_2	…	φ_i
0	**=aba**	**≠aba | ⊥**	**=aba**		**=aba**
1	**=aba**	**=aba**	**=aba**		**≠aba | ⊥**
2	**=aba**	**≠aba | ⊥**	**≠aba | ⊥**		**=aba**
…				**…**	
j	**=aba**	**≠aba | ⊥**	**≠aba | ⊥**		**≠aba | ⊥**

5.2 irudia: Funtzio konputagarri desberdinei buruz **f** funtzioak ematen digun informazioa adierazteko adibide-taula. $\mathbf{f}(\mathbf{i},\mathbf{j})$ den kasuetan, $\varphi_i(\mathbf{j})$-ren balioa **aba** dela aurkitzen dugu. Aldiz, $\neg\mathbf{f}(\mathbf{i},\mathbf{j})$ denean $\varphi_i(\mathbf{j})$-ren balioa **aba**-ren desberdina dela ala indefinitua dela dakigu.

Adibide honetan, geratze-problemaren kasuan bezala, funtzio bakoitzaren puntu guztietan gertatzen denari buruzko informazioa daukagu. Ondorengoko

[8] Lekua aurreztearren eta erabilpena erraztearren, sarrerako balizko hitzak, hauek errepresentatzen dituzten zenbaki arrunten bitartez indexatzea askoz erosoagoa da, eta hitzarmen hori erabiliko dugu.

adibideetan ikusiko ditugun beste kasu batzuetan informazioa puntu batzuetara bakarrik mugatuko da, baina taulak puntu horiek ikusten lagunduko digu.

Beraz, puntu guztietan ematen zaigun informazioa da emaitza bezala **aba** katea azaltzen den ala ez. δ funtzioak **i** puntu bakoitzean φ_i funtzioaren kontrakoa egin dezan diagonaleko balioak aukera ditzakegu. Horrela, δ(**i**) balioa **aba** izango da φ_i(**i**) hala ez denean, eta **aba**-ren desberdina den edozein balio (**bb**, esaterako) izango da, φ_i(**i**)-ren emaitza **aba** dela, hain zuzen, dakigunean, **f** funtzioari esker betiere.

Definizioa hobeto ulertzeko eta prozesuaren laburpen moduan, aurreko taulari, beste zutabe batean, δ funtzio berriari dagozkion balioak eransten dizkiogu, 5.3 irudian erakusten den moduan.

Guztiz nabaria da δ funtzioa, nahi genuen bezala, taulako funtzio bakoitzaren desberdina dela, diagonalari dagokion puntuan gutxienez:

- δ eta φ_0 funtzioak **0** puntuan desberdinak dira, $\varphi_0(0)$=**aba** eta $\delta(0)$=**bb** baitira
- δ eta φ_1 funtzioak **1** puntuan desberdinak dira, $\varphi_1(1)$=**aba** eta $\delta(1)$=**bb** baitira
- δ eta φ_2 funtzioak **2** puntuan desberdinak dira, $\neg(\varphi_2(2)$=**aba**) eta $\delta(2)$=**aba** baitira
- …
- Oro har, δ eta φ_i funtzioak **i** puntuan desberdinak dira, φ_i(**i**)=**aba** eta δ(**i**)=**bb** direlako edo bestela $\neg(\varphi_i$(**i**)=**aba**) eta δ(**i**)=**aba** betetzen delako

δ	**x**	φ_0	φ_1	φ_2	…	φ_i
bb	**0**	**=aba**	**≠aba | ⊥**	**=aba**		**=aba**
bb	**1**	**=aba**	**=aba**	**=aba**		**≠aba | ⊥**
aba	**2**	**=aba**	**≠aba | ⊥**	**≠aba | ⊥**		**=aba**
…	**…**				**…**	
aba	**x**	**=aba**	**≠aba | ⊥**	**≠aba | ⊥**		**≠aba | ⊥**

5.3 irudia: δ funtzio diagonalaren eraiketa-plana. **i** puntu bakoitzaren gainean funtzioak duen portaera, balio berarekin φ_i-k duenaren kontrakoa da. δ definitzeko kontuan hartu diren puntuak, kasu honetan diagonalekoak, belztu egin dira.

Laburbilduz, konputagarri guztien desberdina izango den δ funtzio berria definitzeko, **f** funtzioaren berariazko ezaugarrietaz baliatzea lortu dugu. δ-ren diseinuan, susmagarria izan litekeen edozein eragiketa (**f** bera izan ezik noski) ez

erabiltzeko kontu handia jarri dugu. Orain funtzio berria inplementatu beharko dugu.

> **3. FASEA – ERAIKETA.** Frogapenaren beraren elementuak ezartzearen unea da. **f**-ren konputagarritasuna suposatu ondoren δ funtzioa formalki definituko dugu, eta bere konputagarritasuna frogatuko. δ-ren konputagarritasuna **f**-renean oinarritzen dela ziurtatu beharko dugu betiere. **e** indizeren batentzat $\delta \cong \varphi_e$ betetzen dela ondorioztatuko dugu.

Kasu gehienetan δ-ren definizioa zuzenean taulan oinarrituta atera daiteke, eta bera konputatuko duen programak ez du erronka handirik sortzen. Ariketa korapilatsuago batzuetan, definizioak nahiz programaren diseinuak arreta berezia eska dezakete, adibidez, δ funtzioa errekurtsiboki definitzen denean.

ADIBIDEA: Planaren arabera, gure kasuan δ funtzioa honela definitzen dugu:

$$\delta(\mathbf{x}) \cong \begin{cases} \mathbf{bb} & \mathbf{f(x,x)} \\ \mathbf{aba} & \neg\mathbf{f(x,x)} \end{cases}$$

f konputagarria dela suposatuz, δ-ren konputagarritasuna programa honen bidez froga dezakegu:

if f(X1, X1) **then** X0 := '**bb**'; **else** X0 := '**aba**'; **end if**;

δ funtzio konputagarria denez, $\delta \cong \varphi_e$ betetzen da. Beraz, funtzio konputagarrien zerrendako elementua izanik, δ funtzioa zehazki taulako **e**-garren zutabearekin bat etorri beharko litzateke.

> **4. FASEA – KONTRAESANA.** δ-ren portaeran kontraesana aurkitu behar dugu. Funtzio konputagarri guztien desberdina denez, gutxienez puntu batean, δ funtzioa, φ_e-tik, hots, bere burutik desberdinarazten duen balio zehatzaren gainean aplikatuko dugu (oro har, diagonaleko **e** beraren gainean edo **e**-rekin erlazionatutakoren baten gainean).

δ funtzioa egokiro eraiki bada, kontraesanera iristea ez da zaila izan behar. Konputagarria dela frogatu dugu eta, hortaz, $\delta \cong \varphi_e$ betetzen dela, baina δ funtzioa eraiki dugu espreski, berau eta φ_e puntu konkretu batean (demagun **e** dela) desberdinak izatea lortzeko. Beste era batera esanda, δ funtzioa **e** puntuan bere buruaren desberdina izan behar da, eta honek ez du zentzurik. $\delta(\mathbf{e})$-rentzat balizko aukerak aztertuko ditugu eta kasu posible guztietan kontraesanera iristen garela ikusiko dugu.

Arretaz definitu ez bada, bere kasu guztiak aztertzea konplikatu samarra izatea gerta daiteke (adibidez, kasu batzuk teilakatzen direlako eta baldintza logiko

konplexuak sortzen dituztelako). Egoera horretan kontraesana bilatzeko **f** beraren kasuak erabiltzea aukera ona izaten da.

ADIBIDEA: Gure δ funtziora itzuliz, φ_x bakoitzarekin **x** puntuan desberdintzeko diseinatua izan dela daukagu. Hortaz, **e** puntuan zer gertatzen den aztertzen dugu eta δ-ren definizioaren arabera $\delta(\mathbf{e})$-rentzat bi aukera ditugula ikusten dugu: bere balioa **bb** da edo bestela **aba** (beste aukerarik ez dago).

1. kasua: $\delta(\mathbf{e})=\mathbf{bb} \overset{\delta\,\text{def.}}{\Rightarrow} \mathbf{f}(\mathbf{e},\mathbf{e}) \overset{\mathbf{f}\,\text{def.}}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{e})=\mathbf{aba} \overset{\delta\cong\varphi_e}{\Rightarrow} \delta(\mathbf{e})=\mathbf{aba}$ ↯

2. kasua: $\delta(\mathbf{e})=\mathbf{aba} \overset{\delta\,\text{def.}}{\Rightarrow} \neg\mathbf{f}(\mathbf{e},\mathbf{e}) \overset{\mathbf{f}\,\text{def.}}{\Rightarrow} \neg(\varphi_{\mathbf{e}}(\mathbf{e})=\mathbf{aba}) \overset{\delta\cong\varphi_e}{\Rightarrow} \neg(\delta(\mathbf{e})=\mathbf{aba})$ ↯

Bi kasuek kontraesanera garamatzate, eta hortaz $\delta(\mathbf{e})$-ren balioa ezin da **aba** izan baina **aba** ez izatea ere ezinezkoa da. Kontraesana, frogatu gabe egin dugun hipotesi bakarretik etorri behar da, hots, **f** konputagarria zela suposatzetik.

Kasu honetan $\delta(\mathbf{e})$-ren aukerak aztertzea erraza izan da. Beste era batera eginda, **f** funtziotik beratik induzitutako kasuak (hots, *true* eta *false* balioei dagozkienak) kontuan har genitzakeen:

1. kasua: $\mathbf{f}(\mathbf{e},\mathbf{e}) \overset{\mathbf{f}\,\text{def.}}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{e})=\mathbf{aba} \overset{\delta\cong\varphi_e}{\Rightarrow} \delta(\mathbf{e})=\mathbf{aba} \overset{\delta\,\text{def.}}{\Rightarrow} \neg\mathbf{f}(\mathbf{e},\mathbf{e})$ ↯

2. kasua: $\neg\mathbf{f}(\mathbf{e},\mathbf{e}) \overset{\text{def.}\,\delta}{\Rightarrow} \neg(\varphi_{\mathbf{e}}(\mathbf{e})=\mathbf{aba}) \overset{\delta\cong\varphi_e}{\Rightarrow} \neg(\delta(\mathbf{e})=\mathbf{aba}) \overset{\delta\,\text{def.}}{\Rightarrow} \delta(\mathbf{e})=\mathbf{bb} \overset{\text{def.}\,\delta}{\Rightarrow}$

$\Rightarrow \mathbf{f}(\mathbf{e},\mathbf{e})$ ↯

eta hauek ere kontraesanera eramango gintuzketen.

5.5 Diagonalizazio teknikaren aldaerak

Funtzioen konputaezintasuna frogatzeko den diagonalizazioaren metodoa, aurreko atalean azaldu dugun antzeko eran beste kasu askotan aplika daiteke. Hala ere, beste batzuetan metodoaren hain zuzeneko interpretazioa aplikatzea ez da horren erraza, batez ere, **f** funtzioak ematen digun informazioa δ eraikitzeko aprobetxatzean zailtasunak izan ditzakegulako. Jarraian, aipatu zentzuan zailtasunak sortzen zaizkien zenbait adibide ikusiko ditugu, eta beraiekin metodoa aplikatzeko aukera izan dezagun hau nola molda dezakegun egiaztatuko dugu. [IIS 00]-n ebatzitako adibide gehiagoren bilduma aurkezten da.

5.5.1 Diagonalaren desplazamendua

f funtzioak diagonaleko $\varphi_i(\mathbf{i})$ balioei buruzko informaziorik ez ematea gerta daiteke. $\varphi_i(\mathbf{i})$ bakoitzean gertatzen denari buruzko daturik ez badugu, ezingo dugu δ diseinatu **i** puntuan φ_i funtzioaren desberdina izan dadin. Hala ere, metodoaren izpiritua manten dezakegu, δ eta funtzio konputagarri bakoitza desberdinarazteko, diagonalekoa ez izan arren, beste balizko punturen bat topatzen badugu. Ideia hori erakusteko honako funtzio hau konputagarria ez dela frogatuko dugu:

$$\mathbf{f}(\mathbf{x}) = \begin{cases} 0 & \varphi_x(\mathbf{x}+10) = \mathbf{x}^2 \\ \mathbf{x}+1 & \text{bestela} \end{cases}$$

JUSTIFIKAZIOA: Bere balizko konputaezintasunaren oinarria $\varphi_x(\mathbf{x}+10)\uparrow$ kasuan datzala pentsa dezakegu intuizioz, hau da, geratze-problemaren erabakiezintasuna dela kausa, $\varphi_x(\mathbf{x}+10)\uparrow$ denean ezingo dugula eskatutako **x**+1 erantzuna eman. Ez dirudi beste bi kasuak ($\varphi_x(\mathbf{x}+10)=\mathbf{x}^2$ eta $\varphi_x(\mathbf{x}+10)\neq\mathbf{x}^2$) berez problematikoak direnik.

PLANIFIKAZIOA: **f**-k funtzio bakoitzaren puntu zehatz batean gertatzen denari buruzko informazioa ematen digu. Zehazkiago, φ_i bakoitzarentzat **i**+10 puntuan itzultzen duen balioa $\mathbf{i}^2$ den ala ez esaten digu. Bereizketa hori ez du, aurreko adibideetan bezala, balio logikoen bitartez egiten, desberdin daitezkeen zenbakizko balioak erabiliz baizik: lehenengo kasuan egongo gara $\mathbf{f}(\mathbf{i})=0$ denean eta bigarrenean $\mathbf{f}(\mathbf{i})>0$ denean. 5.4 irudiko adibide-taulan $\mathbf{f}(\mathbf{i})$-ren balio hipotetikoak sartzen ditugu, funtzio horrek ematen dizkigun datuak irudikatzeko.

Gogoan izan ez dugula taulako puntu guztien inguruko informaziorik, zutabe bakoitzeko, puntu bakarraren berri dugu soilik. Eraiki behar dugun δ funtzioa taulan ageri diren funtzio guztien desberdina izan behar da, baina bereizketa hori sortzeko derrigorrez **f** funtzioa euskarri bezala hartu behar dugunez, **f**-k zein sarreretarako ematen duen informazioren bat kontuan hartu eta puntu horietan oinarrituko gara. Taulan ikusten den bezala *desplazatutako diagonala* lortzen dugu:

		f(0)>0	**f**(1)=0	**f**(2)=0	**f**(3)>0	**f**(4)=0	**f**(5)=0
δ	**x**	φ_0	φ_1	φ_2	φ_3	φ_4	φ_5
	0						
	1						
	…						
0	**10**	$\perp \mid \neq 0$					
$\perp$	**11**		1				
$\perp$	**12**			4			
9	**13**				$\perp \mid \neq 9$		
$\perp$	**14**					16	
$\perp$	**15**						25

5.4 irudia: Adibide-taula, non diagonaleko balioen inguruko informaziorik ez dugula ikusten den. Hala ere, belztutako balioek δ funtzioaren eraiketa, desplazatutako diagonal batean lortutako informaziotik abiaturiko eraiketa, ahalbidetzen dute.

Baldintza hori kontuan hartuz, δ funtzioa φ_0-rekin 10. puntuan desberdinarazten saiatuko gara, φ_1-ekin 11 puntuan, φ_2-rekin 12 puntuan eta horrela gainontzekoekin. Adibidez, badakigu $\varphi_0(10)$-ek ez duela 0 balioa (dibergitu egiten duelako edo bestela beste balio desberdin bat hartzen duelako), hortaz, $\delta(10)=0$ definitzea oso ondo datorkigu. $\varphi_1(11)=1$ dela badakigu ere, horrela $\delta(11)$ dibergentea izatea nahikoa izango zaigu (1-en desberdina den edozein balio jartzea erabaki genezakeen). Oro har, **i** indizea duen funtzioak **i**+10 puntuan $\mathbf{i}^2$ itzultzen badu, δ funtzio berriak puntu horretan dibergitu egingo du, eta bestela $\delta(\mathbf{i}+10)$-k $\mathbf{i}^2$ balioko du, hain zuzen ere. Noski, $\varphi_{\mathbf{i}}(\mathbf{i}+10)$-n zer gertatzen den jakiteko $\mathbf{f}(\mathbf{i})=0$ den ala $\mathbf{f}(\mathbf{i})>0$ den ikustea nahikoa izango da. Gure plana 5.4 irudiko adibide-taulan isladatuta geratzen da.

Gure arrazonamenduaren emaitzarako $\delta(0)$ ... $\delta(9)$ balioek ez dute garrantzirik, funtzio diagonala lortzeko gainontzekoekin moldatzen garelako. Definizioa errazteko, funtzioa puntu horietan dibergiarazi dezakegu.

Beraz, $\mathbf{x}\geq 10$ den puntu bakoitza $\varphi_{\mathbf{x-10}}$ funtzio konputagarriarekin desberdintzeko erabiliko duen δ funtzioa definitzera goaz. $\mathbf{x}<10$ diren puntuak nahi den moduan definituko ditugu:

- $\delta(0)$ ... $\delta(9)$ balioek ez dute inolako erabilgarritasunik

- $\delta(10)$-en bitartez δ eta φ_0 bereiz daitezke, $\delta(10)=0^2$ delako eta $\varphi_0(10)$ berriz ez
- $\delta(11)$-ren bitartez δ eta φ_1 bereiz daitezke, $\varphi_1(11)=1^2$ delako eta $\delta(11)$ berriz ez
- $\delta(12)$-ren bitartez δ eta φ_2 bereiz daitezke, $\varphi_2(12)=2^2$ delako eta $\delta(12)$ berriz ez
- …
- Oro har, $\delta(\mathbf{x})$-ren bitartez δ eta $\varphi_{\mathbf{x}-10}$ bereiz daitezke, $\varphi_{\mathbf{x}-10}(\mathbf{x})=(\mathbf{x}-10)^2$ denean orduan $\delta(\mathbf{x})$-k balio hori hartzen ez duelako, eta alderantziz

ERAIKETA: Hortaz **f** funtzioa konputagarria dela suposatuko dugu. Planaren arabera, δ funtzioa formalki definituko dugu:

$$\delta(\mathbf{x}) \cong \begin{cases} (\mathbf{x}-10)^2 & \mathbf{x} \geq 10 \wedge \mathbf{f}(\mathbf{x}-10) > 0 \\ \perp & \text{bestela} \end{cases}$$

Eta bere konputagarritasuna honako programaren bidez argi dago

if X1≥10 **and** f(X1-10)>0 **then** X0 := (X1-10)^2; **else** X0 := ⇊ ; **end if**;

Beraz, δ konputagarria izanik badago **e** indize bat, non $\delta \cong \varphi_{\mathbf{e}}$ betetzen den.

KONTRAESANA: Ikus dezagun, δ-ren eraiketan arrazoitu dugun moduan, aipatu indize hori existitzea ezinezkoa dela, δ funtzioa **i** indizea duen funtzio konputagarri bakoitzarekin **i**+10 puntuan desberdintzen delako. δ funtzioa **e**+10 puntuan aplikatzen badugu kontraesana lortzen dugula egiaztatuko dugu. $\delta(\mathbf{e}+10)$ balioarentzat bi kasu posible ditugu:

1. kasua: $\delta(\mathbf{e}+10)=\mathbf{e}^2 \overset{\delta \text{ def.}}{\Rightarrow} \mathbf{e}+10 \geq 10 \wedge \mathbf{f}(\mathbf{e})>0 \overset{\mathbf{f} \text{ def.}}{\Rightarrow} \mathbf{f}(\mathbf{e})=\mathbf{e}+1 \overset{\mathbf{f} \text{ def.}}{\Rightarrow}$
$\neg(\varphi_{\mathbf{e}}(\mathbf{e}+10)=\mathbf{e}^2) \overset{\delta \cong \varphi_e}{\Rightarrow} \neg(\delta(\mathbf{e}+10))=\mathbf{e}^2$ ↯

2. kasua: $\delta(\mathbf{e}+10)\uparrow \overset{\delta \text{ def.}}{\Rightarrow} \mathbf{e}+10<10 \vee \mathbf{f}(\mathbf{e})=0 \Rightarrow \mathbf{f}(\mathbf{e})=0 \overset{\mathbf{f} \text{ def.}}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{e}+10)=\mathbf{e}^2 \overset{\delta \cong \varphi_e}{\Rightarrow}$
$\Rightarrow \delta(\mathbf{e}+10)=\mathbf{e}^2$ ↯

Egia esan, horietatik bat ere ez da posible, kontraesanera eramaten gaituztelako. Eta hau egin dugun hipotesi bakarretik sortzen da, **f** konputagarria dela suposatzetik, hain zuzen ere.

5.5.2 Diagonalaren desitxuratzea

Ideia berarekin jarraituz, hots, diagonaleko $\varphi_i(\mathbf{i})$ balioen inguruko informazioa ematen ez duen **f** funtzio baten konputaezintasuna frogatzearen ideiarekin jarraituz beste aldaera batzuk ere aurki ditzakegu. Aldaera hauetan diagonalaren ordezkoak erabiltzera behartuta egongo gara eta ordezko horien definizioak zailtasun-maila desberdinetakoak izango dira. Adibide bat, honako funtzioa konputagarria ez dela frogatzea izan daiteke:

$$\mathbf{f}(\mathbf{x}) = \begin{cases} 8 & \varphi_{\mathbf{x}}(2 * \mathbf{x}) \bmod 3 = 0 \\ 5 & \text{bestela} \end{cases}$$

JUSTIFIKAZIOA: Ez dela konputagarria intuizioz susma dezakegu. **x** indizea duen funtzioak $2*\mathbf{x}$ puntuan konbergitzen duenean bere emaitza 3-ren multiploa denentz zehazteko ez dago inolako arazorik. Alabaina, $\varphi_{\mathbf{x}}(2*\mathbf{x})\uparrow$ denean ezin dugu erantzun egokirik eman, geratze-problemaren erabakiezintasunak egoera hori aurreikustea eragozten digulako.

PLANIFIKAZIOA: **f**-k funtzio bakoitzaren puntu zehatz batean gertatzen denari buruzko informazioa ematen digu. $\mathbf{f}(\mathbf{x})$-ren emaitzaren arabera, $\varphi_{\mathbf{x}}$ funtzioak $2*\mathbf{x}$ puntuan 3-ren multiplo bat sortzen duenentz jakingo dugu. 5.5 irudiko adibide-taulan puntu horiek belztuta erakusten dira, eta kasu honetan zutabe bakoitzeko balio bakar baten informazioa dugula ikusten dugu, eta gainera balio hori diagonalekoarekin bat ez datorrela, ezta diagonal desplazatua hartuta ere. Alabaina, diagonal desitxuratu horrek ere δ funtzio diagonala eraikitzeko baliozko alternatiba osatzen du.

δ funtzioa φ_i funtzio guztietatik desberdintzeko, informazioa ematen digun puntua erabiliko dugu. δ funtzioa φ_0 funtzioarekin 0 puntuan desberdinduko dela bermatuko dugu, φ_1-ekin 2 puntuan, φ_2-rekin 4 puntuan, eta horrela gainontzekoekin. Gainera φ_0 funtzioak 0 puntuan 3-ren multiploa den balio bat ematen duela dakigunez, funtzio berriari 100 balioa harraraziko diogu (edo 3-ren multiploa ez den beste edozein balio, edo bestela dibergitu ere egin lezake). φ_1 funtzioa 2 puntuan 3-ren multiploa ez dela dakigunez (dibergentea delako edo bestela $3*\mathbf{n}$ erakoa ez den balio bat hartzen duelako) δ funtzio berriak 3 balioa har dezan egin dezakegu (edo 3-ren multiploa den beste edozein ere). Prozesua orokor dezakegu eta δ funtzioa φ_i bakoitzarekin desberdintzeko erabiliko diren puntuak lortuko genituzke. Baina **f**-k ematen digun informazioaren arabera, horrek δ funtzioa sarrera bikoitietan nola definitu jakitera garamatza, besterik ez.

		f(0)=8	f(1)=5	f(2)=8	f(3)=8	**f**(4)=5
δ	**x**	φ_0	φ_1	φ_2	φ_3	φ_4
100	**0**	mod 3 = 0				
$\perp$	**1**					
3	**2**		$\perp$ | mod 3 ≠ 0			
$\perp$	**3**					
100	**4**			mod 3 = 0		
$\perp$	**5**					
100	**6**				mod 3 = 0	
$\perp$	**7**					
3	**8**					$\perp$ | mod 3 ≠ 0

5.5 irudia: Adibide-taula, non diagonaleko puntuen inguruko informaziorik ez dugula ikusten den. Hala ere, belztutako balioek δ funtzioaren eraiketa, desitxuratutako diagonal batean lortutako informaziotik abiaturiko eraiketa, ahalbidetzen dute.

δ-ren definizioa egokia izan dadin puntu bakoitietan nola definitzen den ere zehaztu behar dugu, nahiz eta arrazoiketaren emaitzarako garrantzitsua ez izan. Ez 100 ezta 3 ere ez erabiltzea komenigarriagoa dela dirudi, definizioan teilakatzeak ekiditeko, gerora kontraesana ezartzea zaildu dezaketen teilakatzeak. Funtzioari puntu horietan dibergituarazi diezaiokegu, adibidez.

- $\delta(1)$, $\delta(3)$, $\delta(5)$, $\delta(7)$, $\delta(9)$, ... balioek ez dute inolako erabilgarritasunik
- $\delta(0)$-rekin δ eta φ_0 bereiz daitezke, $\delta(0)=100$ eta $\varphi_0(0)$ 3-ren multiploa baitira
- $\delta(2)$-rekin δ eta φ_1 bereiz daitezke, $\delta(2)=3$ delako eta $\varphi_1(2)$ ez
- $\delta(4)$-rekin δ eta φ_2 bereiz daitezke, $\delta(4)=100$ eta $\varphi_2(4)$ 3-ren multiploa baitira
- …
- Oro har, $\delta(2*\mathbf{x})$-ren bitartez δ eta φ_x bereiz daitezke, $\varphi_x(2*\mathbf{x})$ balioa 3-ren multiploa denean orduan $\delta(\mathbf{x})$ balioa 100 delako eta $\varphi_x(2*\mathbf{x})$-ren emaitza 3-ren multiploa ez denean orduan $\delta(\mathbf{x})$ balioa 3 delako

ERAIKETA: δ funtzioa formalki definitzen dugu, lehen ikusi dugun bezala sarrera bikoitiak eta bakoitiak bereizten ditu eta **f**-ren informazioa sarrera bikoitietarako bakarrik erabiltzen dugu:

$$\delta(\mathbf{x}) \cong \begin{cases} 100 & \mathbf{x} \bmod 2 = 0 \wedge f(\mathbf{x}/2) = 8 \\ 3 & \mathbf{x} \bmod 2 = 0 \wedge f(\mathbf{x}/2) = 5 \\ \perp & \mathbf{x} \bmod 2 \neq 0 \end{cases}$$

Behin funtzioa definituta eta **f** konputagarria dela suposatuz, δ funtzioa ere konputagarria dela daukagu, honako programak frogatzen duen bezala

```
X0 := 3;
if  X1 mod 2 = 0  then
        if  f(X1 div 2) = 8  then X0 := 100; end if;
else   X0 := ⇊ ;
end if;
```

Beraz, badago **e** indizea zeinarekin $\delta \cong \varphi_{\mathbf{e}}$ betetzen den.

KONTRAESANA: δ-ren eraiketa arrazoitu dugun moduan, indize hori ezinezkoa dela ikus dezagun. Ezinezkoa, δ funtzioa $2*\mathbf{e}$ puntuan aplikatuz gero kontraesana lortzen dugulako, hain zuzen ere.

1. kasua: $\delta(2*\mathbf{e})=100 \overset{\delta \text{ def.}}{\Rightarrow} (2*\mathbf{e}) \bmod 2=0 \wedge \mathbf{f}(\mathbf{e})=8 \overset{\mathbf{f} \text{ def.}}{\Rightarrow} \varphi_{\mathbf{e}}(2*\mathbf{e}) \bmod 3 = 0 \overset{\delta \cong \varphi_e}{\Rightarrow}$
$\delta(2*\mathbf{e}) \bmod 3 = 0$ ↯

2. kasua: $\delta(2*\mathbf{e})=3 \overset{\delta \text{ def.}}{\Rightarrow} (2*\mathbf{e}) \bmod 2=0 \wedge \mathbf{f}(\mathbf{e})=5 \overset{\mathbf{f} \text{ def.}}{\Rightarrow} \neg(\varphi_{\mathbf{e}}(2*\mathbf{e}) \bmod 3 = 0) \overset{\delta \cong \varphi_e}{\Rightarrow}$
$\neg(\delta(2*\mathbf{e}) \bmod 3 = 0)$ ↯

3. kasua: $\delta(2*\mathbf{e})\uparrow \overset{\delta \text{ def.}}{\Rightarrow} (2*\mathbf{e}) \bmod 2 \neq 0$ ↯

Beraz $\delta(2*\mathbf{e})$ ezin da ez 3 ez 100 izan (bi kasuetan kontraesana daukagu), baina dibergitu ere ezin du egin, funtzioaren definizioagatik hori bakoitietarako bakarrik gertatzen delako. Beraz $2*\mathbf{e}$ puntuan dugun kontraesana egin dugun suposaketa bakarretik sortzen da, **f** konputagarria dela suposatzetik.

5.5.3 Diagonalizazio asimetrikoa

Azkenik kasu bereziago bat ikusiko dugu. Batzuetan **f** funtzioak, zeinen konputaezintasuna frogatu nahi dugun, ez digu informazio osoa ematen. Funtzio konputagarri batzuen inguruan datu asko ematen digu, gero funtzio diagonala eraikitzeko ahalegin berezirik gabe aprobetxa dezakegun informazioa. Beste batzuen inguruan, berriz, ia ez digu ezer esaten eta δ berria funtzio hauetatik bereizteko lana asko zailtzen du.

Fenomeno hau gertatzen denean diagonalizazioa burutzeko bi estrategia desberdin hartu beharko ditugu aldi berean, azaldu berri ditugun bi egoerei era

eraginkorrean ekiteko. Horrelako kasu baten adibidea honako funtzioa konputagarria ez dela frogatzen saiatzean ematen da:

$$\mathbf{f}(\mathbf{x}) = \begin{cases} \text{true} & \mathbf{x} \in \text{TOT} \Leftrightarrow \mathbf{W}_x = \Sigma^* \Leftrightarrow \varphi_x \text{ osoa da} \\ \text{false} & \text{bestela} \end{cases}$$

JUSTIFIKAZIOA: Intuitiboki konputagarria ez dela susma dezakegu, $\mathbf{x} \in$TOT den aztertzea $\forall \mathbf{y}\ \varphi_x(\mathbf{y})\downarrow$ dela probatzearen parekoa delako, eta honek geratze-problema sarrera-kopuru infinitu batentzat ebaztea eskatzen duela dirudi.

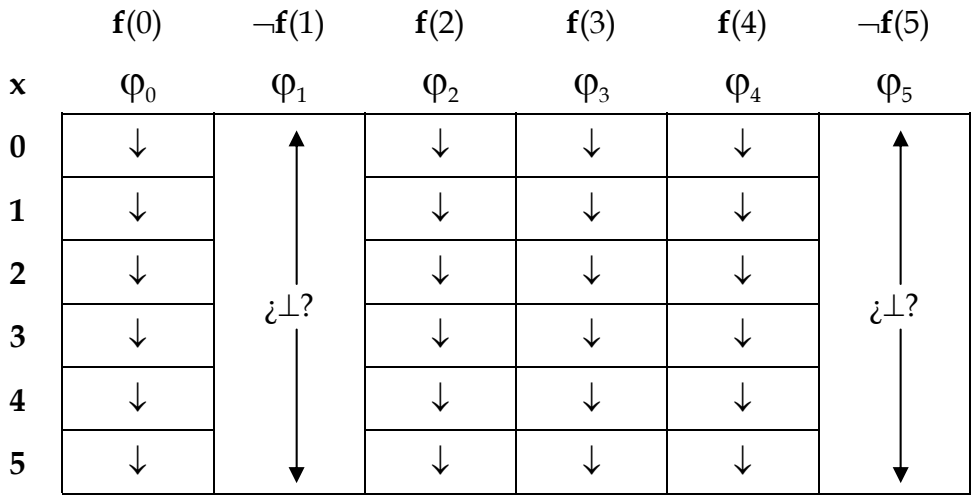
PLANIFIKAZIOA: **f** funtzioak taulako funtzio konputagarri bakoitzarekin gertatzen denari buruzko informazioa ematen digu. Zehazkiago, φ_x bakoitzarentzat puntu guztietarako konbergitzen duen ala baten batean dibergitzen ote duen aipatzen digu.

- **x** balioa TOT multzoan badago ($\mathbf{f}(\mathbf{x})$ betetzen delako jakingo dugu), φ_x funtzioa **y** hitz guztietarako definituta dagoela jakingo dugu. φ_x-ri buruzko informazio hori oso baliozkoa da, ziklatuko ote duen beldurrik gabe, edozein $\varphi_x(\mathbf{y})$ baliori buruz galde baitezakegu, eta jasotako erantzuna $\delta(\mathbf{y})$ balioa desberdina izan dadin lortzeko erabili. Zehazkiago, $\varphi_x(\mathbf{x})$ diagonalean desberdindu nahi badugu $\delta(\mathbf{x})\uparrow$ egin dezakegu, edo $\varphi_x(\mathbf{x})$=0 denentz galde dezakegu eta erantzunaren arabera 1 edo 0 erabili, edo bestela $\varphi_x(\mathbf{x})$-ren balio desberdinak kalkulatu, esaterako $\varphi_x(\mathbf{x})$+100, $\varphi_x(\mathbf{x})$+1 edo $2*\varphi_x(\mathbf{x})$+7.
- **x** balioa TOT multzoan ez badago ($\mathbf{f}(\mathbf{x})$ betetzen ez delako jakingo dugu), φ_x funtzioak **y** hitzen batentzat gutxienez dibergitzen duela jakingo dugu. φ_x-ri buruzko informazio hori oso eskasa da, indefinizio hori zehazki zein hitzen gainean gertatuko den ez baitzaigu esaten. Beraz, diagonaleko **x** hitzaren gainean φ_x-k duen portaerari buruzko informazio zehatzik ez daukagu, eta are larriagoa dena, ezta beste edozein **y** hitzen gaineko portaerari buruz ere. Edozein balioren gainean ere ezin dugu gehiago galdetu, φ_x ez-osoa izanik jakin-min hori zoritxarrekoa suerta baitaiteke. Horrela, δ funtzioa sarrera zehatz batean φ_x-tik desberdintzea lortzeko inolako oinarririk ez daukagula da atera dezakegun ondorioa.

Mota desberdineko funtzioei buruz jasotzen dugun informazioaren kalitatea desorekatuta dagoenez nabarmenki, kasu honi *diagonalizazio asimetrikoarena* deitzen diogu. Egoera 5.6 irudian adierazitako taulan aurkezten dena bezalakoa da.

Ikusten dugun bezala, zailtasuna $\mathbf{x} \notin$TOT kasuan dago, δ eta φ_x funtzioak inolako puntutan lokalki desberdintzea ezin baitugu lortu. Hala ere, φ_x-ri buruz daukagun informazioa ez da guztiz alferrikakoa: funtzio ez-osoa dela dakigu

gutxienez. Horrela ikusita, δ eta φ_x funtzioak desberdinak izatea lortu ahal izango genuke, δ funtzioa osoa izateko moduan definituko bagenu. Diferentzia lokal hori zehazki zein puntutan gertatzen den jakin ez arren, helburu hori betez gero δ (osoa) eta φ_x (ez-osoa) ezin direla funtzio bera izan ziurtatuta edukiko dugu, eta hori da benetan interesatzen zaiguna. Kasu honetan egiten duguna, δ funtzioa φ_x-tik *era orokorrean desberdintzeko* moduan definitzea da.



5.6 irudia: Diagonalizazio asimetrikoaren kasuen ezaugarri nagusia, ustez konputaezina den **f** funtzioak kasu batzuetan oso informazio eskasa ematen duela da. Hemen **f** funtzioak φ_x osoa dela esaten digunean arazorik ez dago, baina $\neg$**f**(**x**) betetzen bada, φ_x funtzioa sarreraren batentzat indefinituta dagoela besterik ez dakigu, baina zeinetan zehazki ez dakigu.

Baina, δ funtzioa osoa bihurtzea erabakitzen badugu, $\mathbf{x}\in$TOT kasuan genituen aukeren multzo zabala murriztu egingo zaigu, dibergitzeko aukera baztertuta geratuko baita. δ osoa izango denez, deskribatzeko asmoa dugun funtzio diagonalari **d** deitzea egokiagoa izango da.

- $\mathbf{x}\in$TOT bada, **d**(**x**) balioa definituko dugu $\varphi_{\mathbf{x}}(\mathbf{x})$+3-rekin, eta honekin **d** funtzioa **x** puntuan lokalki $\varphi_{\mathbf{x}}$-ren desberdina izango da. Hau egin dezakegu $\varphi_{\mathbf{x}}(\mathbf{x})\downarrow$ dela dakigulako eta horrela **d** osoa izateko dugun asmoa mantentzen dugu.
- $\mathbf{x}\notin$TOT bada, φ_x funtziotik lokalki desberdintzeko ez dugu **x** puntua behar. Hori bai, **d**(**x**)-ri balioren bat eman beharko diogu, 10 esaterako, **d** osoa izatea eta horrela φ_x-tik era orokorrean desberdintzea lortzeko.

Eskema 5.7 irudiko adibide-taulan erakutsitakoa litzateke, bertan **d** funtzio berria, funtzio konputagarri osoetatik diagonaleko puntuan eta konputagarri ez-osoetatik era orokorrean desberdintzen dela ikus dezakegu. Orokorrean desberdintzean zehazki zein puntutan den ezin dugu esan, baina ziur gaude **d**-ren indizeak TOT multzoan daudela eta uneko φ_x-renak berriz ez.

		f(0)	¬**f**(1)	**f**(2)	**f**(3)	**f**(4)	¬**f**(5)
d	**x**	φ_0	φ_1	φ_2	φ_3	φ_4	φ_5
7	**0**	4	¿⊥?	↓	↓	↓	¿⊥?
10	**1**	↓		↓	↓	↓	
3	**2**	↓		0	↓	↓	
5	**3**	↓		↓	2	↓	
9	**4**	↓		↓	↓	6	
10	**5**	↓		↓	↓	↓	

5.7 irudia: d funtzio diagonal asimetriko baten eraiketa. Alde batetik, konputagarri eta osoa izatea lortzen dugu, eta bestetik, funtzio konputagarri eta oso guztietatik diagonaleko balioetan desberdintzea. Azken hau lortzeko aipatu balio horiek ateratzen ditugu, funtzioa osoa dela dakigula aprobetxatuz.

Deskribatu berri dugun bereizketa-mekanismoa nabarmentzea komeni daiteke agian, **d** funtzioa taulan dagoen funtzio bakoitzetik zergatik desberdintzen den ikusteko:

- **d** eta φ_0 funtzio osoa desberdinak dira 0 puntuan, $\varphi_0(0)=2$ eta $\mathbf{d}(0)=5$ baitira.
- **d** eta φ_1 funtzio ez-osoa desberdinak dira, non ez dakigun arren. Baina ziur gaude badagoela **y** puntu bat non $\varphi_1(\mathbf{y})\uparrow$ betetzen den, **d** funtzioak puntu horretan konbergituko duen bitartean.
- ...
- **d** eta edozein $\varphi_\mathbf{x}$ funtzio oso **x** puntuan desberdinak dira, $\varphi_\mathbf{x}(\mathbf{x})$ definituta dagoelako eta $\mathbf{d}(\mathbf{x})=\varphi_\mathbf{x}(\mathbf{x})+3$ delako.
- **d** eta edozein $\varphi_\mathbf{x}$ funtzio ez-oso desberdinak dira, non ez dakigun arren. Baina badago **y** puntu bat non $\varphi_\mathbf{x}(\mathbf{y})\uparrow$ betetzen den, $\mathbf{d}(\mathbf{y})$ konbergentea izango den bitartean. $\varphi_\mathbf{x}(\mathbf{x})=10=\mathbf{d}(\mathbf{x})$ kasua eman daitekeela azpimarratzea garrantzitsua da eta gainera horrek gure arrazoiketan inolako eraginik ez daukala, **d** eta $\varphi_\mathbf{x}$ funtzioak desberdinak izatea ez baitago **y** puntuan ematen duten balioaren menpe.

ERAIKETA: Demagun **f** konputagarria dela. **d** funtzioa formalki definitzen dugu, planeatu dugun moduan:

$$\mathbf{d}(\mathbf{x}) = \begin{cases} 10 & \neg\mathbf{f}(\mathbf{x}) \\ \varphi_\mathbf{x}(\mathbf{x})+3 & \mathbf{f}(\mathbf{x}) \end{cases}$$

f konputagarria denez (zehazkiago, erabakigarria, predikatua baita), **d** funtzioa ere konputagarria dela daukagu, eta honako programak frogatzen du:

if not f(X1) **then** X0 := 10; **else** X0 := Φ(X1,X1)+3; **end if**;

Beraz badago **e** indize bat, zeinarekin $\mathbf{d}=\varphi_{\mathbf{e}}$ betetzen den. **d** funtzioa osoa dela nabarmentzea garrantzitsua da: **else** zatian funtzio unibertsalari deitzen diogunean, bere egikaritzapena simulatuko dugun programa funtzio oso bati dagokiola **f**-ren bitartez egiaztatu gabe ez baitugu egiten. Beraz *gainera* ***f(e)*** *ere egiaztatzen da*.

KONTRAESANA: **d**-ren eraiketa arrazoitu dugun moduan, aipatu indize hori ezinezkoa dela ikus dezagun. $\mathbf{d}=\varphi_{\mathbf{e}}$ osoa denez, **d** funtzioa $\varphi_{\mathbf{e}}$-tik **e** puntuan desberdinduko da, eta ondorioz **d**(**e**) kalkulatzean kontraesana lortuko dugu. Hasteko bi aukera daude, baina bigarrenarekin ez gara oso urrutira iritsiko $\varphi_{\mathbf{e}}$ osoa delako:

1. kasua: $\mathbf{d}(\mathbf{e})\cong\varphi_{\mathbf{e}}(\mathbf{e})+3 \overset{\mathbf{d}\cong\varphi_e}{\Rightarrow} \mathbf{d}(\mathbf{e})\cong\mathbf{d}(\mathbf{e})+3 \Rightarrow \mathbf{d}(\mathbf{e})\uparrow \Rightarrow$ **d** ez da osoa $\lightning$

2. kasua: $\mathbf{d}(\mathbf{e})=10 \overset{\mathbf{d}\text{ def.}}{\Rightarrow} \neg\,\mathbf{f}(\mathbf{e}) \overset{\mathbf{f}\text{ def.}}{\Rightarrow} \varphi_{\mathbf{e}}$ ez da osoa $\overset{\mathbf{d}\cong\varphi_e}{\Rightarrow}$ **d** ez da osoa $\lightning$

Kontraesan hau orain arte aurkitu ditugunetatik zertxobait desberdintzen dela ikus dezagun. Arrazoia diagonalizazioaren asimetrian datza. Itxarotekoa zen bezala, **d**-ren osotasuna frogapenaren funtsezko elementua bihurtu da, eta kontraesanak, **f** konputagarria zela esaten zuen hasierako suposaketa faltsua dela frogatzen digu.

> *Aztertu berri dugun f funtzioa konputaezina izateak programazio-lanetan oso muga zorrotza ezartzen du. Informatika praktikoaren aplikazio gehienetan, esleitu zaien ataza bukatuko duten programak eta errutinak eraikitzea bilatzen dela nabaria da. Sortutako softwarearen arazketarako lehen irizpidea, baldintzaren baten pean indefinituki zikla lezaketen (hots, ez-osoa den funtzio bat konputatzen duten) programa guztiak kentzea edo ordezkatzea izan beharko litzateke. Zoritxarrez, lan hori era algoritmikoan burutzeko modurik ez dagoela egiaztatu berri dugu. Honek, erroreak dituen hainbeste software zergatik ote dagoen hausnartzera eraman gaitzake. Errore horiek, nabarmenenak izanda ere (esaterako, programan nahi gabe begizta infinituak sartzea), sistematikoki detektatzeko metodorik ez dagoela da arazoa. Geratze-problemaren kasuan bezala, honek ez du esan nahi programa bat betirako segurua dela edo egikaritzapen batzuetarako zikla dezakeela egiaztatzea ezinezkoa denik beti: badaude oso programa sinpleak, analisi minimoa nahikoa dutenak.*

Badago aipatzeko modukoa den azken gai bat. Egin dugun azken frogapena azalekoa dela pentsa liteke, *halt*-en konputaezintasunak, programa osoak eta ez-

osoak bereizten dituen **f** funtzioarena ondorioztatzen duela dirudielako. Arrazoiketa honakoa litzateke: programa bat datu zehatz baten gainean geratzen denentz erabaki ezin badugu, datu guztien gainean geratuko ote den nola erabakiko dugu? Baina irudipen hori faltsua da, hasteko, *halt* eta **f**-ren konputaezintasunak independenteak direlako. Programa baten osotasuna egiaztatzeko metodoren bat egongo balitz ere, *halt*-ek konputaezina izaten jarraituko luke. Arrazoia honakoa da: **f(x)** beteko balitz, $\mathbf{P}_x$ programa edozein daturekin geratzen dela jakingo genuke, eta beraz "bere" geratze-problema "berezia" ebatzita egongo litzateke. Hala ere, $\neg\mathbf{f}(\mathbf{x})$ izatean φ_x ez dela osoa jakingo genuke, baina metodoak ez liguke zertan funtzioak non konbergitzen duen eta non ez jakiten utzi behar, eta ez genuke *halt*-en erabakiezintasuna ebatziko.

6. Erabakigarritasuna eta sasierabakigarritasuna

Aurreko atalean konputagarriak ez diren problemak badaudela frogatu dugu eta gainera problema baten konputaezintasuna ezartzeko erabil daitekeen diagonalizazio teknika ere aurkeztu dugu. Adibide batzuen bidez konputaezintasunaren gakoa non kokatzen den susmatu ahal izan dugu. Kasu askotan algoritmorik ez dago, ezinezkoa delako erantzuna ematea, hau programak ziklatzen duenentz egiaztatzearen menpekoa denean (eta hori ebazteko beste modurik ez dagoenean).

Programa batek zikla dezakeela ikusteak portaera hori (hots, emaitza indefinitua) errepresentatzeko $\perp$ balio asmatua hitzartzera, eta gainera funtzio partzialekin era sistematikoan lan egitera behartu gaitu. Alde batetik, esan bezala, programek zikla dezakete eta ebidentzia honek, konputagarritasunaren fenomenoa zehazki deskribatzeko egite hori kontuan hartu behar dugula adierazten digu. Bestetik, amaitzen ez den programaren ideia algoritmo kontzeptuaren aurka dago, azken hau metodo finitu bezala ulertuta. Funtzio hutsa konputatzen duena eta ondorioz, inoiz inolako emaitzarik sortzeko gai ez dena bezalako programak algoritmo moduan kontuan hartu behar izatea zentzugabekeria dirudi.

> *Geratzen ez diren programen problema gogaikarria ez lukeen informatika asmatzea askoz arrazoizkoagoa izango zela dirudi. Irtenbide bat, aukera hori ematen ez duten lengoaiak definitzea izango zen, baina hurbilketa horrekin lortzen den bakarra, konputagarriak diren funtzio interesgarri asko konputatzera iristen ez diren sistema okerrak sortzea besterik ez da (hori da, hain zuzen, 21 orrialdeko oharrean aipaturiko for programen kasuan gertatzen dena). Beste balizko bide bat, while-programena bezalako lengoaia osoa hartu eta funtzio ez-osoak konputatzen dituzten programak kentzea izango litzateke. Aurreko atalean hori ere ezinezkoa dela ikusi dugu, programa-multzo hori erabakigarria ez delako, hots, kanpoan zein elementu dauden eta barnean zeintzuk erabaki ezin delako: programa "onak" eta "txarrak" bereiztu ezin baditugu, azken horiek gainetik kentzeko gaizki edukiko dugu. Portaera dibergentearekin bizitzera kondenatuta gaude, era batera edo bestera.*

Hala ere, zenbaitetan bukatzen ez diren algoritmo batzuk konputaziorako funtsezkoak dira, 4.2 atalean deskribaturiko **U** programa unibertsalak frogatzen duen moduan. Eta zein da horren arrazoia? Ez-osoak diren programa horiek direla hagitz zailak diren problema batzuk ebazteko (partzialki bada ere) daukagun onena. Hortaz problema konputagarriak berriro kontuan hartuko ditugu,

ondorengo irizpidearen araberako sailkapena egiteko: guztiz ebatzi ditzakegun problemak eta hurbilketa ez-oso bat besterik ez duten problemak.

6.1 Erabakitze-problemak eta multzoak

Lehenik eta behin, 'problema' termino orokorra zerbait zehatzagora murriztuko dugu. Dokumentu honen hasieratik, ebatzi beharreko problemak funtzio bezala aurkeztu ditugu, sarrera-datu batzuetatik abiaturik emaitzak (irteera) lortuko dituen programa (existitzen bada) aurkitu nahi dela ulertuz. Hala ere, 5. kapituluan zehar garatzen joan garen adibideetan, funtzioaren konputaezintasunerako bere emaitza soil-soilik inoiz ez dela garrantzizkoa gertatu egiaztatu ahal izan dugu. Funtzioa konputagarria izateko oztoporen bat egon denean, hau beti *true*/*false* motako baldintza edo erabakia hartzeko zailtasunean oinarritua izan da, eta dilema hori ebatzi aurreko nahiz ondorengoko ekintzak funtzioaren izaera konputaezinari eragiten ez zioten apaingarriak besterik ez dira izan. Zerbait konputatzearen zailtasunaren (edo ezintasunaren) ikuspegitik, funtzio interesgarrienak predikatuak direla susma dezakegu.

Esperientzia horretan oinarrituz gure eszenatokia pixka bat sinplifikatuko dugu, arreta emaitza boolearra duten funtzioetan, hots, predikatuetan bakarrik jartzeko. Predikatuen bitartez adieraz daitezkeen problemak ***erabakitze-problemak*** deitzen dira. Eta, problema interesgarri guztiak mota honetakoak ez diren arren, Konputagarritasun Teoriarako garrantzitsuak diren problema guztiei erabakitze-problema bat dagokiela esan dezakegu.

Argumentu horren funtsa argitzeko, har dezagun adibide bezala problema zaila ebatziko duen programa bat, esaterako, C konpilatzaile bat: bere ataza programa bat C lengoaiatik makina-lengoaia zehatz batera itzultzea da. Noski bere funtzionamendua eta emaitza, *true* edo *false* balioak bakarrik itzultzen dituzten programek erakusten dituztenetatik oso urruti geratzen dira. Baina konpilatzaileak, besteak beste, iturburu programaren baliotasunari buruz erabaki behar du. Berez, hori da bere funtsezko ekintza, analisi horren ondoren bakarrik eraiki baitezake helburu lengoaian beste programa baliokide bat. Hortaz, konpilatzaileak ebazten duen problemaren konputagarritasuna, bere aldetik, itxuraz errazagoa den beste problema baten konputagarritasunean datza. Problema errazago hori honako galderari erantzun boolearra ematen diona da: sarrerako karaktere-segidak, C lengoaiaren espezifikazioaren arabera, programa egokia osatzen al du?. Programa berri hau konputagarria izango ez balitz, C konpilatzailea ez litzateke existituko.

Bereziki erabakitze-problemen ardura hartzeaz gainera, ikuspegi funtzionalaren pixka bat desberdina den ikuspuntutik egingo dugu: gure

diskurtsoaren gaia bezala predikatuak hartu ordez, aldiz, *multzoei* buruz hitz egingo dugu. Oso erosoa gertatuko zaigun arren, bereizketa ez da hain garrantzitsua, predikatu bakoitzari multzo bat dagokiolako eta alderantziz. Adibidez, "sarrerako karaktere-segidak, C lengoaiaren espezifikazioaren arabera, programa egokia osatzen al du?" galderari *true* edo *false* balioarekin erantzuten dion predikatuak, C lengoaiako programak osatzen dituzten, hots, predikatua betetzen duten karaktere-kateen multzoa definitzen du. Era berean, zehatz-mehatz 1.347 zatitzaile desberdin dituzten zenbaki arruntek osatutako multzoari honako predikatua dagokio: "sarrerako zenbakiak zehatz-mehatz 1.347 zatitzaile al ditu?".

Erabiliko ditugun multzoak, inplementaturiko datu-motetakoak izango diren objektuez osatuta egongo dira, multzo horietatik interesatzen zaiguna beren konputagarritasun-propietateak direlako, eta beraz beren elementuak while-programen bitartez maneiatzeko aukera ere. Ondorengokoak multzo horien adibideak dira:

$\mathbf{A} = \{ \mathbf{x} \in \sum^*: |\mathbf{x}|_a > 3 \}$

$\mathbf{B} = \{ \mathbf{x} \in \mathbb{N}: \exists \mathbf{y} \leq \mathbf{x}\ \mathbf{y}^3 = \mathbf{x} \}$

$\mathbf{C} = \{ (\mathbf{x},\mathbf{y}) \in \sum^* \times \mathbb{P}:$ **x** hitza tontorra(**y**)-ren aurrizkia da $\}$

baina, orain arte ikusitakoarekin bat etorriz, gure gogokoenak programekin zerikusia duten multzoak izango dira, horiek baitira propietate interesgarrienak dituzten objektuak:

$\mathbf{D} = \{ \mathbf{x} \in \mathbb{W}:$ $\mathbf{P}_\mathbf{x}$ programak begiztarik ez dauka $\}$

$\mathbf{E} = \{ (\mathbf{x},\mathbf{y}) \in \mathbb{W} \times \sum^*: \varphi_\mathbf{x}(\mathbf{y}) \downarrow \}$

$\mathbf{F} = \{ (\mathbf{x},\mathbf{y},\mathbf{z}) \in \mathbb{W} \times \sum^* \times \mathbb{N}:$ $\mathbf{P}_\mathbf{x}(\mathbf{y})$-k zehazki **z** pausotan konbergitzen du $\}$

Multzoak eta predikatuak erlazionatzeko modu sistematikoari buruz, berriz, honako definizioa sartzen dugu:

DEFINIZIOA: Bedi $\mathbf{A} \subseteq \sum^*$ hitz multzoa. Honako predikatuari **A**-ren ***ezaugarri funtzioa*** deitzen diogu:

$$\mathbf{C}_\mathbf{A}(\mathbf{x}) = \begin{cases} \text{true} & \mathbf{x} \in \mathbf{A} \\ \text{false} & \mathbf{x} \notin \mathbf{A} \end{cases}$$

Beraz, **A** multzo baten ezaugarri funtzioa **A**-ko elementuak multzokoak ez direnetatik *bereizteko* balio duen predikatua da. Azken finean, gure erabakitze-problemak, datu bat multzo zehatz bateko kidea izatea edo ez izatearen inguruko deliberoarekin erlazionatuta egongo dira.

Nozio edo kontzeptu hori k hitzez osatutako tuplen multzoetara zabal daiteke era naturalean. Hala da, $\mathbf{B}\subseteq\sum^{*k}$ bada, orduan bere ezaugarri funtzioa honakoa izango da:

$$\mathbf{C_B}(\mathbf{x}_1,\mathbf{x}_2,...\mathbf{x}_k) = \begin{cases} \text{true} & (\mathbf{x}_1,\mathbf{x}_2,...,\mathbf{x}_k)\in\mathbf{B} \\ \text{false} & (\mathbf{x}_1,\mathbf{x}_2,...,\mathbf{x}_k)\notin\mathbf{B} \end{cases}$$

Era berean definitzen dira hitzak ez diren objektuen edo objektu-tuplen multzoen ezaugarri funtzioak, hau da, era hauetako multzoena: $\mathbf{A}\subseteq\mathbb{T}$ edo $\mathbf{B}\subseteq\mathbb{T}_1\times\mathbb{T}_2\times...\times\mathbb{T}_k$, non $\mathbb{T}$, $\mathbb{T}_1$, $\mathbb{T}_2$,..., $\mathbb{T}_k$ inplementaturiko datu-motak diren.

Goragoko adibideen inguruan, $\mathbf{C_F}$: $\mathbb{W}\times\sum^*\times\mathbb{N}\rightarrow\mathbb{B}$ ezaugarri funtzioa, adibidez, zehazki honakoa izango dela daukagu:

$$\mathbf{C_F}(\mathbf{x},\mathbf{y},\mathbf{z}) = \begin{cases} \text{true} & \mathbf{P_x}(\mathbf{y}) \text{ - k zehazki } \mathbf{z} \text{ pausotan konbergitzen du} \\ \text{false} & \text{bestela} \end{cases}$$

6.2 Multzo erabakigarriak eta beren propietateak

Multzoetara hurbiltzeko erabaki dugun ikuspuntua kontuan edukiz eta, aitortu dugun bezala, interesatzen zaiguna beren konputagarritasun-propietateak direla aintzat hartuz, plantea dezakegun lehen zalantza da multzo batekin, eta programa baten bitartez, zein eratako lana egin dezakegun.

DEFINIZIOA: Bedi $\mathbf{A}\subseteq\sum^*$ multzoa. **A *erabakigarria*** dela esaten dugu, baldin eta bere ezaugarri funtzioa, $\mathbf{C_A}$, konputagarria bada.

Nozio hori, era naturalean, dimentsio anitzeko multzoetara edo beren elementuak beste datu-mota batzuetakoak dituzten multzoetara orokortzen da.

Multzo erabakigarriaren ideia eta lehen eman dugun *predikatu erabakigarriarena* koherenteak dira, multzo bat erabakigarria baita, baldin eta soilik baldin berari dagokion predikatua (hots, bere ezaugarri funtzioa) ere hala bada. Beraz, erabakigarria terminoa bi erako elementuekin berdin-berdin erabiltzeko aukera izango dugu.

Multzo bat erabakigarria dela frogatu nahi izanez gero, beraz, bere ezaugarri funtzioa konputatuko duen programa aurkitu beharko dugu. **B** adibidearen kasuan honako programak balio izan dezake:

```
X0 := false;
for IND in 0..X1 loop
      X0 := X0 ∨ (X1=IND∗IND∗IND);
end loop;
```

sarrerako datua kubo perfektua denean *true* eta bestela *false* itzultzen duena.

Beraz, intuitiboki, multzo bat erabakigarria da multzoko elementuak eta kanpokoak direnak *bereizteko* aukera ematen duen algoritmoa existitzen denean. Normalean lantzen ditugun multzo gehienak (denak ez badira) erabakigarriak dira (adbz, zenbaki lehenen multzoa, posizio ez-nuluak baino posizio nulu gehiago dituzten bektoreena, UNIX BSDn baliozko agindua errepresentatzen duten karaktere-kateen multzoa, etab.)

Aldiz, multzoa erabakiezina izango da bere kideak diren elementuak eta ez direnak bereizteko aukera ematen duen algoritmoa aurkitzea ezinezkoa denean. Multzo bat erabakigarria ez dela frogatzeak ezaugarri funtzio baten konputaezintasuna frogatzea ondorioztatzen du, eta hortaz, oro har, diagonalizaziora jo beharko dugu. Erabakiezina dela badakigun multzo bat goragoko **E** adibidea da, bere $\mathbf{C_E}$ ezaugarri funtzioa lehen landu dugun *halt* funtzioa bera baita. **E**-rekin erlazionaturik **K** multzoa dugu, bere buruaren gainean konbergitzen duten programen multzoa, hain zuzen:

$$\mathbf{K} = \{ \mathbf{x} \in \mathbb{W}: \varphi_{\mathbf{x}}(\mathbf{x})\downarrow \}$$

$\mathbf{C_K}$ funtzioa konputagarria ez dela formalki frogatu ez dugun arren, hori ez da batere zaila gertatzen. *Halt*-en konputaezintasunarentzat 5.2 atalean aurkezturiko frogapena zertxobait moldatzea besterik ez dugu. Beste adibide bat **TOT** multzoa dugu:

$$\mathbf{TOT} = \{ \mathbf{x} \in \mathbb{W}: \forall \mathbf{y}\ \varphi_{\mathbf{x}}(\mathbf{y})\downarrow \}$$

$\mathbf{C_{TOT}}$ funtzioaren erabakiezintasuna 5.5.3 atalean zehaztu baikenuen (orduan oraindik izen horrekin deitzen ez bagenuen ere)

NOTAZIOA: Erabakigarriak diren multzo guztien klaseari Σ_0 deitzen diogu. Hortaz, **A** erabakigarria dela esatea edo $\mathbf{A} \in \Sigma_0$ dela esatea berdina da.

> *Konputagarritasun Teoriaren inguruko literatura klasikoaren zati handi batean multzo erabakigarriei **errekurtsibo** deitzen zaiela aurkitzen dugu. Honen arrazoia historikoa da, konputagarritasun-propietateak aztertzeko erabilitako lehenengo ereduekin erlazionatuta. Hala ere termino hori nahasgarria dela pentsatzen dugu eta horregatik bere baliokidea den erabakigarria nahiago dugu.*

6.2.1 Multzo erabakigarrien itxitura-propietateak

Oro har, multzo erabakigarrien propietateak, ulertu eta frogatzeko errazak dira.

PROPOSIZIOA 1: Multzo *finituak* eta *kofinituak* erabakigarriak dira.

⌦ Bedi $\mathbf{A}\subseteq\sum^*$ multzo finitua eta bitez $\{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ bere **n** elementuak. $\mathbf{C_A}$ ezaugarri funtzioa konputatzen duen programa honakoa da:

```
X0 := false;
if  X1=Y1  or  …  or  X1= YN  then   X0 := true;   end if;
```

Bedi orain $\mathbf{B}\subseteq\sum^*$ multzo kofinitua eta bitez $\{\mathbf{z}_1, \ldots, \mathbf{z}_m\}$ $\bar{\mathbf{B}}$ multzo finituaren **m** elementuak. $\mathbf{C_B}$ ezaugarri funtzioa konputatzen duen programa honakoa da:

```
X0 := true;
if  X1=Z1  or  …  or  X1=ZM  then   X0 := false;   end if;
```
⌫

Emaitza honek, alde batetik, oso logika erraza du: multzo baten elementuek **n** kideko kopuru finitua osatzen badute, beti egongo da programa bat (hau hedatzean **n** if kasu dituen zerrenda batez osatuta egongo da, if bat sarrerako datua multzoko balizko elementu bakoitzarekin alderatzeko). Baina bestaldetik ondorio konplexuak ditu, ulertzeko edo beren mugak ikusteko zailtasun handiak ditugun multzo finitu batzuk badaudelako. Adibidez, Malabar kostako tokiren batean euria egin zuen 1325 urteko otsaileko egun guztien multzoa, edo beren kodea milioi bat baino txikiagoa den programa oso guztien multzoa. Bi hauek multzo finituak dira eta beraz erabakigarriak, baina ez ginateke beren ezaugarri funtzioentzat programak eraikitzeko gai izango. Bi alderdiak bateratzeko, 3.1 atalean ikusi genuen *konputagarritasun eskuragaitzaren* kontzeptura jotzen dugu berriro. Multzo hauek erabakigarriak dira eta beren ezaugarri funtzioak konputatzen dituzten programak existitzen dira. Baina, beren espezifikazioek laguntzen ez digutelako, multzo horien izaera ez dugu programa zehatzak aurkitzeko lain ezagutzen, hori da gertatzen zaiguna.

PROPOSIZIOA 2: Multzo erabakigarrien bildura eta ebakidura erabakigarriak dira.

⌦ Bitez $\mathbf{A}, \mathbf{B}\subseteq\sum^*$ multzo erabakigarriak. Beraz, beren $\mathbf{C_A}$ eta $\mathbf{C_B}$ ezaugarri funtzioak konputagarriak dira. $\mathbf{A}\cap\mathbf{B}\in\sum_0$ dela ikusteko, bere $\mathbf{C_{A\cap B}}$ ezaugarri funtzioa konputatzen duen honako programa erabiliko dugu:

$X0 := C_A(X1) \wedge C_B(X1);$

$\mathbf{A}\cup\mathbf{B}\in\sum_0$ dela ikusteko, bere $\mathbf{C_{A\cup B}}$ ezaugarri funtzioa konputatzen duen honako programa erabiliko dugu:

X0 := $C_A(X1) \vee C_B(X1)$; ⌫

PROPOSIZIOA 3: Multzo erabakigarri baten osagarria erabakigarria da.

⌦ Bedi $\mathbf{A} \subseteq \sum^*$ multzo erabakigarria. Beraz, bere $\mathbf{C_A}$ ezaugarri funtzioa konputagarria da. $\mathbf{C_{\overline{A}}}$ ezaugarri funtzioa konputatzen duen programa honakoa da:

X0 := $\neg C_A$ (X1); ⌫

Ikusten dugun moduan, multzo erabakigarriak Multzo Teoriako eragiketa klasikoen bitartez manipulatzeak emaitza onak ematen ditu. Horren arrazoia da, eragiketa horiek multzoei dagozkien predikatuei aplikatzen zaizkien eragiketa logikoekin bat datozela eta, beren aldetik, eragiketa hauek programa osoen konbinaketa errazetan (kasukako bereizketen bitartez) isladatzen direla.

Propietate hauek, era naturalean, k hitzetako tuplak dituzten multzoetara ($\mathbf{B} \subseteq \sum^{*k}$) edo hitzak ez diren objektuen multzoetara ($\mathbf{B} \subseteq \mathbb{T}_1 \times \mathbb{T}_2 \times ... \times \mathbb{T}_k$, non $\mathbb{T}_1$, $\mathbb{T}_2$,..., $\mathbb{T}_k$ inplementaturiko datu-motak diren) orokortzen dira. Frogapenak lehenagokoen antzekoak dira baina datuak dimentsio anitzekoak direla, eta beraz, **k** sarrera-aldagai erabiltzen dituztela kontuan hartuz.

6.3 Multzo sasierabakigarriak eta beren propietateak

Beren ezaugarri funtzioak konputagarriak ez direlako **K** eta **TOT** multzoak erabakigarriak ez direla aipatu dugu. Hala ere bi multzoen artean desberdintasunen bat badagoela susma dezakegu. Ezinezkoa egiten saiatzen bagara, hau da $\mathbf{C_K}$ ezaugarri funtzioa konputatzen, zailtasun guztia $\varphi_{\mathbf{x}}(\mathbf{x})\uparrow$ denean *false* itzultzean datzala aurkitzen dugu, baina $\varphi_{\mathbf{x}}(\mathbf{x})\downarrow$ denean ez genuke *true* itzultzeko inolako arazorik izango. Zentzu horretan "ia konputagarria" dela esan dezakegu, *false* balioaren ordez indefinitua jarrita nahikoa izango bailitzateke funtzio konputagarria lortzeko.

Aldiz, **TOT** multzoarentzat antzeko irtenbidea lortzea ez dirudi erraza: ez $\forall \mathbf{y}\ \varphi_{\mathbf{x}}(\mathbf{y})\downarrow$ ($\mathbf{x} \in \mathbf{TOT}$) kasuan ezta $\exists \mathbf{y}\ \varphi_{\mathbf{x}}(\mathbf{y})\uparrow$ ($\mathbf{x} \notin \mathbf{TOT}$) denean ere erantzun egokia lortzeko aukerarik ez daukagu. $\mathbf{C_{TOT}}$ funtzioa konputagarria izateko $\mathbf{C_K}$ baino urrutiago dagoela esan dezakegu nolabait.

Une honetan dugun multzoen zatiketa (erabakigarriak eta ez erabakigarriak) birfintzea da oraingoan bete nahi dugun helburua, horretarako erabakiezinen arteko batzuk, konputagarritasunetik gertuago daudenak, bereiziko ditugularik. Multzo horiek, erabakigarriak izan ez arren, soluzio partziala onartzen dute: badute programa bat, multzokoak diren sarreretarako erantzun positiboa emateko gai dena baina multzokoak ez direnetarako ziklatu egiten duena.

Ikusi dugun funtzio konputaezinen lehenengo adibidea berreskura dezagun: geratze-problemaren *halt* funtzioa. Beti bukatzeko eskakizuna erlaxatuz honako funtzio konputagarria aurkitzen dugula ikusten dugu:

$$\xi(\mathbf{x},\mathbf{y}) \cong \begin{cases} \text{true} & \varphi_{\mathbf{x}}(\mathbf{y})\downarrow \\ \perp & \text{bestela} \end{cases}$$

Funtzio honen konputagarritasuna frogatzea erraza da funtzio unibertsala erabiltzea nahikoa baita.

```
R := Φ(X1, X2);
X0 := true;
```

Atal honen sarreran aurreratu dugun moduan, multzo erabakiezin batzuetarako funtzio konputagarri bat aurki dezakegu, ezaugarri funtzioa pixka bat erlaxatuz, geratze-problemarekin oraintxe egin dugunaren antzera. Ezaugarri funtzioan *false* irteera indefinituarekin ordezkatuz erdiezagaurri funtzioa lortzen dugu.

DEFINIZIOA: **A** multzoa emanik, **A**-ren ***erdiezaugarri funtzioa*** honako funtzio partzialari deitzen diogu:

$$\chi_{\mathbf{A}}(\mathbf{x}) \cong \begin{cases} \text{true} & \mathbf{x} \in \mathbf{A} \\ \perp & \text{bestela} \end{cases}$$

DEFINIZIOA: $\mathbf{A} \subseteq \sum^*$ multzoa ***sasierabakigarria*** da, baldin eta bere erdiezaugarri funtzioa, $\chi_{\mathbf{A}}$, konputagarria bada.

Bi nozio horiek, era naturalean, dimentsio anitzeko multzoetara edo beren elementuak beste datu-mota batzuetakoak dituzten multzoetara orokortzen dira.

Multzo bat sasierabakigarria dela frogatzea nahi izanez gero, beraz, bere erdiezaugarri funtzioa konputatuko duen programa aurkitu beharko dugu. Adibidez, $\mathbf{K} = \{ \mathbf{x} \in \mathbb{W} : \varphi_{\mathbf{x}}(\mathbf{x})\downarrow \}$ multzoa (erabakigarria ez dela ikusita duguna) sasierabakigarria suertatzen da. Bere $\chi_{\mathbf{K}}$ erdiezaugarri funtzioa programa honek kalkulatzen du:

```
R := Φ(X1, X1);
X0 := true;
```

Multzo honek garrantzi teknikoa handia du, oso maneiagarria suertatzen baita berau erabakiezina baina sasierabakigarria den multzoaren arketipo bezala hartzeko (hau da, erabakigarriak ez izan arren "ia" direnen arketipo bezala). Adibidez zenbait multzoren erabakiezintasuna egiaztatzeko absurdura eramanez egiten diren frogapen askotan erabiltzen da.

Multzo sasierabakigarriaren beste adibide bat honakoa litzateke

$$\overline{\mathbf{VAC}} = \{ \mathbf{x} \in \mathbb{W}: \exists \mathbf{y}\ \varphi_{\mathbf{x}}(\mathbf{y})\downarrow \}$$

kasu batean gutxienez emaitza itzultzeko gai diren programen multzoa. Bere erdiezaugarri funtzioa prozesu tartekatzearen teknika erabiltzen duen programa baten bidez kalkulatzen da:

```
BIKOTE := 0;
while not T(X1,deskod_2_1(BIKOTE),deskod_2_2(BIKOTE)) loop
      BIKOTE := hur(BIKOTE);
end loop;
X0 := true;
```

Beraz, intuitiboki, multzoa erabakigarria da, multzoko elementuak *detektatzeko* aukera ematen duen algoritmoa existitzen denean, nahiz eta algoritmo horrek multzokoak ez direnentzat erantzun bat aurkitu beharrik ez daukan.

Aldiz, multzoa ez da sasierabakigarria izango bere elementuentzat detekzio-algoritmo bat aurkitzea ere ezinezkoa denean. Multzo bat sasierabakigarria ez dela frogatzeak erdiezaugarri funtzio baten konputaezintasunaren frogapena ondorioztatzen du, baina 5. kapituluan ikusi ditugun diagonalizazioaren aldaerak, oro har, funtzio ez-osoen konputaezintasuna egiaztatzeko ez direla ondo moldatzen egiazta daiteke. Arazo honi aurre egiteko, 6.3.4 atalean diagonalizazio teknika aberastuko dugu.

NOTAZIOA: Sasierabakigarriak diren multzo guztien klaseari Σ_1 deitzen diogu. Hortaz, **A** sasierabakigarria dela esatea edo $\mathbf{A} \in \Sigma_1$ dela esatea berdina da.

> *Multzo erabakigarriekin gertatzen den bezala (errekurtsiboak ere deitzen baitzaie), multzo sasierabakigarrientzat **errekurtsiboki zenbagarriak** izena oso hedatuta dagoela aurkitzen dugu. Antzeko arrazoiengatik ez dugu nomenklatura hori erabiliko.*

6.3.1 Sasierabakigarritasunaren eta erabakigarritasunaren arteko erlazioa

Multzo sasierabakigarriak zailagoak diren aparteko problemak direnaren inpresioa eman izana gerta daitekeen arren, hori ez da horrela, multzo klaseen arteko erlazioa partekotasuna baita.

PROPOSIZIOA 4: Erabakigarriak diren multzoak sasierabakigarriak dira. Hau da, $\Sigma_0 \subset \Sigma_1$

⌦ Bedi $\mathbf{A} \subseteq \sum^*$ multzo erabakigarria. Beraz, bere $\mathbf{C_A}$ ezaugarri funtzioa konputagarria da, eta honetan oinarrituko gara $\chi_{\mathbf{A}}$ erdiezaugarri funtzioa konputatuko duen programa idazteko:

if $C_A(X1)$ **then** X0 := true; **else** X0 := $\perp\!\!\!\perp$; **end if**;

Honekin $\Sigma_0 \subseteq \Sigma_1$ betetzen dela frogatu dugu. Partekotasun erlazioa hertsia dela ikusteko, $\mathbf{K} \in \Sigma_1$ izanik $\mathbf{K} \notin \Sigma_0$ dela gogoratzea besterik ez dugu. ⌫

Hortaz, Σ_1 klasea zentzu ez murriztailean ulertu behar dugu. **B** sasierabakigarria bada, **B**-ren elementuak detektatzeko **P** programa badagoela dakigu. Programa honek akatsak ditu $\overline{\mathbf{B}}$-ren elementuak tratatzean, baina hortik ezin dugu **P** programa **B**-rekin lan egiteko izan dezakegun onena denik ondorioztatu. Agian, **P**-z gain, beste **Q** programa bat egon daiteke, eraginkorragoa dena, hots, **B**-ko sarreretatik bereiziz, $\overline{\mathbf{B}}$-koak behar bezala tratatzen dituena. Kasu horretan **B** multzoa sasierabakigarriaz gain, erabakigarria ere izango da. Σ_1 (multzo erabakigarriak nahiz erabakiezinak dauzkana) eta $\Sigma_1 - \Sigma_0$ (multzo erabakiezinak bakarrik dauzkana) klaseak ez nahastea funtsezkoa da.

Dena den, multzo klase bien artean badago erlazio askoz estuago bat, eta hau klase horien osagarrien portaerak ematen digu.

PROPOSIZIOA 5 (OSAGARRIAREN TEOREMA): $\mathbf{A} \subseteq \sum^*$ multzoa erabakigarria da, baldin eta soilik baldin **A** nahiz bere osagarria, $\overline{\mathbf{A}}$, sasierabakigarriak badira.

⌦ Lehenik alde nabariena, hots, $\mathbf{A} \in \Sigma_0 \Rightarrow \mathbf{A} \in \Sigma_1 \wedge \overline{\mathbf{A}} \in \Sigma_1$ betetzen dela egiaztatuko dugu, enuntziatu eta frogatu berri ditugun propietateetatik ondorioztatzen baita.

$$\left.\begin{array}{l} \mathbf{A} \in \Sigma_0 \overset{\text{prop. 4}}{\Rightarrow} \mathbf{A} \in \Sigma_1 \\ \mathbf{A} \in \Sigma_0 \overset{\text{prop. 3}}{\Rightarrow} \overline{\mathbf{A}} \in \Sigma_0 \overset{\text{prop. 4}}{\Rightarrow} \overline{\mathbf{A}} \in \Sigma_1 \end{array}\right| \Rightarrow \mathbf{A} \in \Sigma_1 \wedge \overline{\mathbf{A}} \in \Sigma_1$$

Orain $\mathbf{A} \in \Sigma_1 \wedge \overline{\mathbf{A}} \in \Sigma_1 \Rightarrow \mathbf{A} \in \Sigma_0$ dela ikus dezagun. Hasteko, **A** eta $\overline{\mathbf{A}}$ multzoak sasierabakigarriak badira, beren erdiezaugarri funtzioak, $\chi_{\mathbf{A}}$ eta $\chi_{\overline{\mathbf{A}}}$, konputagarriak izango dira. $\chi_{\mathbf{A}}$ funtzioa konputagarria izateak, elementu bat **A**-koa noiz den detektatzeko algoritmo bat daukagula esan nahi du. Baina $\chi_{\overline{\mathbf{A}}}$ funtzioa ere konputagarria izateak, gainera elementu bat $\overline{\mathbf{A}}$-koa noiz den detektatzeko beste algoritmo bat daukagula esan nahi du. Bi algoritmoak paraleloan konbinatuz, edozein elementu bi multzoetatik zeinetan dagoen zehazteko hutsezinezko metodoa edukiko dugula ondorioztatzea logikoa dirudi.

Orduan, bitez $\mathbf{e_1}$ eta $\mathbf{e_2}$ hitzak, $\chi_{\mathbf{A}}$ eta $\chi_{\overline{\mathbf{A}}}$ funtzioen indizeak, hurrenez hurren. $\mathbf{C_A}$ funtzioa konputagarria dela (eta hortaz **A**, erabakigarria dela) frogatuko dugu, honako programaren bitartez:

```
PAUSOAK := 1;
while not T(E1, X1, PAUSOAK) and not T(E2, X1, PAUSOAK) loop
      PAUSOAK := suc(PAUSOAK);
end loop;
X0 := T(E1, X1, PAUSOAK);
```

Begizta hori eduki arren, gogoan hartu programa honek egiazki funtzio oso bat konputatzen duela, edozein **x** daturentzat, $\chi_{\mathbf{A}}(\mathbf{x})$ eta $\chi_{\overline{\mathbf{A}}}(\mathbf{x})$ konputazioetatik, ezinbestez, bakarra izango baita konbergentea. ⌫

Emaitza hau multzo batzuk sasierabakigarriak ez direla frogatzeko aplika dezakegu, konputaezintasun frogapenen bat egin beharrik gabe. Adibide moduan $\overline{\mathbf{K}}$ multzoa har dezagun eta Σ_1 klasean ezin dela egon ikus dezagun frogapena absurdura eramanez. $\overline{\mathbf{K}}$ multzoa sasierabakigarria balitz, **K** ere hala dela jadanik badakigunez, 5. proposizioa aplika genezake eta hortik **K** erabakigarria dela ondorioztatu, egia ez dela badakiguna. Beraz $\overline{\mathbf{K}}$ sasierabakigarria ez den multzo baten gure lehenengo adibidea da.

6.3.2 Multzo sasierabakigarrien itxitura-propietateak

Multzo erabakigarrien propietateetako batzuk sasierabakigarrientzat ere egiaztatzen dira, baina hauen frogapena ezin dugu pauso berdin-berdinak errepikatuz bete, programa ez-osoak konbinatzea osoekin egitea baino askoz zailagoa delako. Gainera murriztapen sintaktikoak ditugu: ezaugarri funtzioak osoak izanik makrobaldintzetan erabil daitezke eta aldiz, erdiezaugarri funtzioekin, oro har, osoak ez izateagatik, hori debekatuta dugu.

PROPOSIZIOA 6: Multzo sasierabakigarrien bildura eta ebakidura sasierabakigarriak dira ere.

⌦ Bitez **A** eta **B** multzo sasierabakigarriak. Beraz, beren erdiezaugarri funtzioak, $\chi_{\mathbf{A}}$ eta $\chi_{\mathbf{B}}$, konputagarriak dira. $\mathbf{A}\cap\mathbf{B}\in\Sigma_1$ betetzen dela ikusteko $\chi_{\mathbf{A}\cap\mathbf{B}}$ funtzioa konputatzen duen honako programa erabiliko dugu:

```
R := χA (X1);
R := χB (X1);
X0 := true;
```

Lehenengo bi esleipenek iragazki-lana betetzen dutela eta bi horiek arrakastaz gainditzen dituzten bakarrak ebakidurako elementuak direla ikusten dugu.

$\mathbf{A}\cup\mathbf{B}\in\Sigma_1$ betetzen dela ikusteko ezin dugu hain zuzeneko metodorik erabili. Edozein **x** sarrera emanda, lehendabizi zein erdiezaugarri funtzio aplikatzen diogun erabakitzeko arazoa dugu. $\chi_{\mathbf{A}}(\mathbf{x})$ kalkulatuz hasten bagara arrisku handia dugu, **x** balioa $\mathbf{B}-\mathbf{A}$ multzoan egon litekeelako eta orduan ziklatu egingo genuke,

B multzoan, eta hortaz bilduran, dagoela egiaztatzeko aukerarik izan gabe. Baina $\chi_B(x)$-rekin hasiko bagina berdin-berdina gertatuko litzateke, **x** sarrera **A**–**B** multzoan dagoenentz aurretiaz ezin baitugu jakin. Irtenbidea bi funtzioak, seriean aplikatu beharrean, paraleloan egikaritzea izango litzateke, horrela **x** balioa **A**-n edo **B**-n dagoen detektatzeko aukera biei emateko.

χ_A eta χ_B funtzioak konputagarriak direnez, bitez $\mathbf{e_1}$ eta $\mathbf{e_2}$ beren bi indize. $\chi_{A\cup B}$ erdiezaugarri funtzioa konputatzen duen programa da:

```
PAUSOAK := 1;
while not (T(E1, X1, PAUSOAK) or T(E2, X1, PAUSOAK)) loop
      PAUSOAK := suc(PAUSOAK);
end loop;
X0 := true;  ⌫
```

PROPOSIZIOA 7: Multzo sasierabakigarri baten osagarriak ez dauka zergatik sasierabakigarria izan beharrik.

⌦ Kontradibide bezala **K** multzoa jar dezakegu. Hau sasierabakigarria dela baina, 5. proposizioaren ondorioz, bere osagarria ezin dela hala izan ikusi dugu ⌫

$\overline{\mathbf{K}} \notin \Sigma_1$ dela egiaztatzeko **K**-ri aplikatu diogun frogapena, berez Σ_1–Σ_0 klasean dagoen beste edozein **A** multzora heda daiteke: **A**-ren elementuak detektatzeko prozedura badugu, ezin dugu $\overline{\mathbf{A}}$-ren elementuak detektatzeko eduki ere, orduan **A**-rentzat erabakitze-prozedura eraikitzeko biak konbinatzea edukiko baikenuke. Hortaz, $\overline{\mathbf{A}} \notin \Sigma_1$.

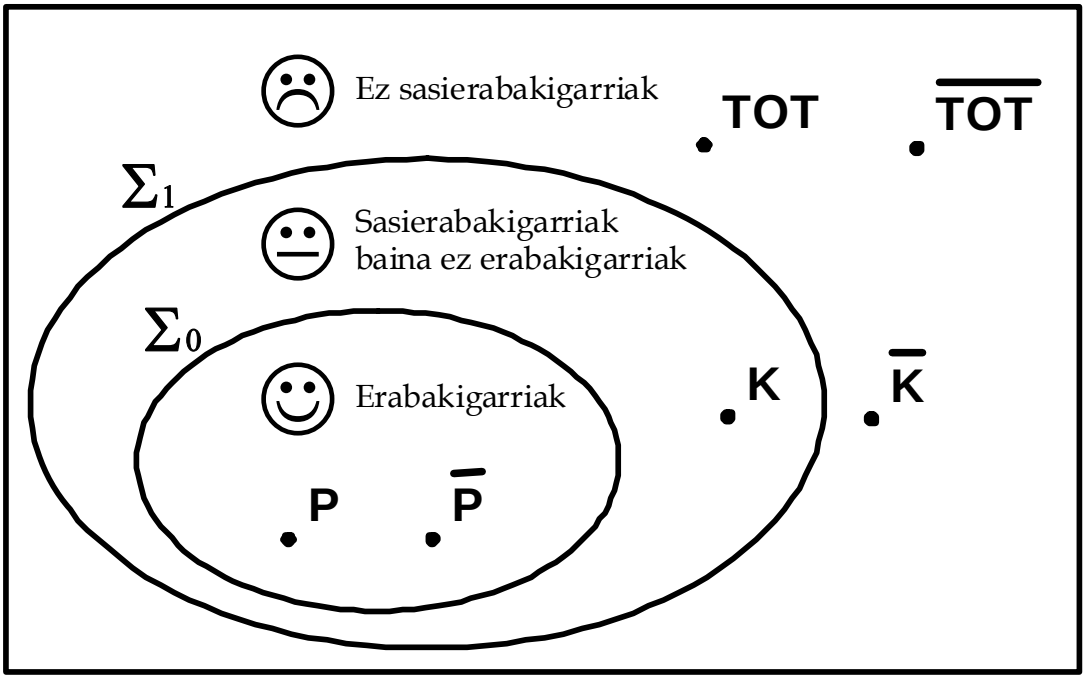
A multzoa 6.1 irudian erakusten diren hiru klaseetako batean sailkatzen laguntzeko bere osagarria aztertzera jotzea, oro har, oso taktika ona da, balizko konbinazioak hiru besterik ez baitira:

1. $\mathbf{A} \in \Sigma_0$ (eta orduan $\overline{\mathbf{A}} \in \Sigma_0$ ere). **A**-ren elementuetarako detekzio-prozedura bat eta $\overline{\mathbf{A}}$-koentzat beste bat posible dira (bi multzoak sasierabakigarriak dira eta beraz erabakigarriak).
2. $\mathbf{A} \in \Sigma_1$–Σ_0 (eta orduan $\overline{\mathbf{A}} \in \overline{\Sigma_1}$). **A**-ren elementuetarako detekzio-prozedura bat posible da baina ez $\overline{\mathbf{A}}$-koentzat (lehenengo multzoa sasierabakigarria da eta bigarrena ez). Alderantzizko kasua ere gerta daiteke.
3. $\mathbf{A} \in \overline{\Sigma_1}$ (eta orduan $\overline{\mathbf{A}} \in \overline{\Sigma_1}$ ere)[9]. Bi multzo horietarako detekzio-prozedurarik ez dago.

Egia esan, gaur egungo gure tresnekin, 3. klaseko multzorik ba ote dagoen ezin dugu frogatu oraindik, baina **TOT** horietako bat izango dela bai aurrera

[9] Berez **A**-ren osagarria sasierabakigarria izan liteke, baina orduan 2. kasuan egongo ginateke, alderantzizko bertsioan.

dezakegu, 6.1 irudian erakusten den moduan, bera eta bere osagarria ez baitira sasierabakigarriak.



6.1 irudia: Multzoen sailkapena, beren erabakigarritasuna edo/eta sasierabakigarritasunaren arabera. Multzo batekin eta bere osagarriarekin eman daitezkeen hiru egoeren hiru adibide erakusten dira: biak Σ_0 klasean egotea (zenbaki bikoitien multzoa den **P**-ren kasua), bat Σ_1-Σ_0 klasean eta bestea Σ_1 klasetik kanpo (bere buruaren gainean konbergitzen duten programen multzoa den **K**-ren kasua) edo biak Σ_1 klasetik kanpo (programa osoen multzoa den **TOT**-en kasua).

Aurreko ataletan enuntziatutakoen kasuan bezala, propietate hauek, era naturalean, k hitzetako tuplak dituzten multzoetara ($\mathbf{B}\subseteq\sum^{*k}$) edo hitzak ez diren objektuen multzoetara ($\mathbf{B}\subseteq\mathbb{T}_1\times\mathbb{T}_2\times...\times\mathbb{T}_k$, non $\mathbb{T}_1$, $\mathbb{T}_2$,..., $\mathbb{T}_k$ inplementaturiko datu-motak diren) orokortzen dira. Frogapenak antzekoak dira eta ez ditugu ematen ez dutelako ezer adierazgarririk gehitzen.

6.3.3 Multzo sasierabakigarrien karakterizazioa

Multzo erabakigarriak ez bezala, multzo sasierabakigarriak karakterizatzeko beste modu batzuk badaude, bere erdiezaugarri funtziora jo gabe. Multzo horiekin lanean dihardugunean beste aukera hauek guztiz baliagarriak dira, eta jarraian teorema moduan enuntziatu eta frogatzeari ekingo diogu.

TEOREMA (Σ_1 KLASEAREN KARAKTERIZAZIOA): Bedi $\mathbf{A}\subseteq\sum^*$ edozein multzo. **A**-ren honako propietateak baliokideak dira:

a) $\mathbf{A}\in\Sigma_1$.

b) **A** multzoa funtzio konputagarri baten eremu edo domeinua da. Hau da, badago ψ: $\sum^* \longrightarrow \sum^*$ funtzio konputagarria, zeinek dom(ψ)=**A** betetzen duen.

c) **A** bi argumentuko predikatu erabakigarri baten kuantifikazio existentziala da (edo bere proiekzioa). Hau da, badago $\mathbf{P}: \sum^* \times \sum^* \to \mathbb{B}$ predikatu erabakigarria, zeinek $\mathbf{A} = \{ \mathbf{x} \in \sum^* : \exists \mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y}) \}$ betetzen duen.

d) **A**, multzo hutsa da ala funtzio konputagarri eta oso baten heina. Hau da, $\mathbf{A} \neq \varnothing$ bada orduan badago $\mathbf{f}: \sum^* \to \sum^*$ funtzio konputagarri eta osoa zeinek ran(**f**)=**A** betetzen duen.

e) **A** edozein funtzio konputagarriren heina da. Hau da, badago ξ: $\sum^* \longrightarrow \sum^*$ funtzio konputagarria zeinek ran(ξ)=**A** betetzen duen.

⌦ Baldintza hauek beraien artean oso desberdinak dira, oro har. Binakako baliokidetasuna frogatzea baino, frogapen zirkular modukoren bat burutzea askoz probetxugarriagoa izango da. Adibidez, **a)** $\Rightarrow$ **b)** $\Rightarrow$ **c)** $\Rightarrow$ **d)** $\Rightarrow$ **e)** $\Rightarrow$ **a)** frogatzen badugu, denen arteko baliokidetasuna frogatuta edukiko dugu, gure ahaleginak minimizatuz.

a) $\Rightarrow$ **b)**

A sasierabakigarria izanik, domeinua **A**-ren berdina duen edozein funtzio konputagarri aurkitu behar dugu.

Baina hori nabaria da, $\mathbf{A} \in \Sigma_1$ bada orduan bere erdiezaugarri funtzioa konputagarria izango baita:

$$\chi_{\mathbf{A}}(\mathbf{x}) \cong \begin{cases} \text{true} & \mathbf{x} \in \mathbf{A} \\ \perp & \text{bestela} \end{cases}$$

eta funtzio honen domeinua **A** da, hain zuzen. Beraz, $\psi \cong \chi_{\mathbf{A}}$ aukeratzen badugu, bilatzen ari ginen eta dom(ψ)=**A** bete behar zuen funtzio konputagarria topatu dugu.

b) $\Rightarrow$ **c)**

ψ konputagarria izanik, **A**=dom(ψ) betetzen dela da gure abiapuntua, eta **P** predikatu bitar eta erabakigarria aurkitu behar dugu, $\mathbf{A} = \{ \mathbf{x} \in \sum^* : \exists \mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y}) \}$ beteko duena.

$\mathbf{x} \in \mathbf{A}$ adierazpenaren esanahia garatuko dugu, horrela kuantifikazio existentzialaren baldintzen arabera adieraztea lortzekotan. Dakigun bakarra hau da:

$$\mathbf{x} \in \mathbf{A} \Leftrightarrow \mathbf{x} \in \text{dom}(\psi) \Leftrightarrow \psi(\mathbf{x})\downarrow$$

baina ψ konputagarria denez, **e** indizea daukala onar dezakegu eta horrela kuantifikazio existentziala sartu, **T** predikatuari esker (eta hau erabakigarria da gainera).

$$\mathbf{x}\in\mathbf{A} \Leftrightarrow \varphi_{\mathbf{e}}(\mathbf{x})\downarrow \Leftrightarrow \exists\mathbf{y}\ \mathbf{T}(\mathbf{e},\mathbf{x},\mathbf{y})$$

eta **e** konstantea dela kontuan hartuz, $\mathbf{P}(\mathbf{x},\mathbf{y}) \equiv \mathbf{T}(\mathbf{e},\mathbf{x},\mathbf{y})$ defini dezakegu, zalantzarik gabe erabakigarria dena, **T**-ren partikularizazioa izateagatik. Honela **P** predikatu erabakigarria daukan $\mathbf{A} = \{ \mathbf{x}\in\sum^*: \exists\mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y}) \}$ adierazpena lortzen dugu.

c) $\Rightarrow$ d)

P predikatu erabakigarriarekin, $\mathbf{A} = \{ \mathbf{x}\in\sum^*: \exists\mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y}) \}$ adierazpena da gure abiapuntua, eta $\mathbf{A}\neq\varnothing$ suposaketa gehigarriaren pean, $\mathbf{f}: \sum^* \to \sum^*$ funtzio konputagarri eta osoa aurkitu behar dugu, $\mathrm{ran}(\mathbf{f})=\mathbf{A}$ beteko duena.

A hutsa ez bada gutxienez elementu bat eduki behar du, $\mathbf{c}\in\mathbf{A}$. Orduan honako funtzioa definitzen dugu:

$$\mathbf{g}(\mathbf{x},\mathbf{y}) = \begin{cases} \mathbf{x} & \mathbf{P}(\mathbf{x},\mathbf{y}) \\ \mathbf{c} & \neg\mathbf{P}(\mathbf{x},\mathbf{y}) \end{cases}$$

argi eta garbi osoa eta konputagarria dena, honako programaren bitartez:

if P(X1, X2) **then** X0:=X1; **else** X0 := C; **end if**;

eta bere heina da

$$\mathrm{ran}(\mathbf{g}) = \{ \mathbf{x}\in\sum^*: \exists\mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y}) \} \cup \{\mathbf{c}\} = \mathbf{A} \cup \{\mathbf{c}\} = \mathbf{A}$$

Orain $\mathbf{f}(\mathbf{x}) = \mathbf{g}(\mathrm{deskod}_{2,1}(\mathbf{x}), \mathrm{deskod}_{2,2}(\mathbf{x}))$ funtzioa definitzearekin nahikoa da. Funtzio konputagarri eta osoen konposaketa izateagatik bera era konputagarria eta osoa izango da, eta bere heina aurrekoaren berdina izango da. Beraz, bilatzen ari ginen funtzioa **f** da.

d) $\Rightarrow$ e)

Kasu honetan bi abiapuntu ditugu, $\mathbf{A}=\varnothing$ da ala $\mathbf{A}=\mathrm{ran}(\mathbf{f})$, non **f** konputagarria eta osoa den, eta $\xi: \sum^* \not\to \sum^*$ funtzio konputagarria aurkitu behar dugu, $\mathrm{ran}(\xi)=\mathbf{A}$ beteko duena.

Lehenengo kasuan bagaude, $\mathbf{A}=\varnothing$ izanik bilatzen dugun funtzio konputagarria $\xi \cong \perp\!\!\!\perp$ izango litzateke, argi eta garbi $\mathrm{ran}(\xi) = \varnothing = \mathbf{A}$ delako.

Bigarren kasuan bagaude, $\mathbf{A}=\mathrm{ran}(\mathbf{f})$ izanik zuzenean $\xi \cong \mathbf{f}$ har dezakegu, konputagarria dena.

e) $\Rightarrow$ a)

$\xi: \sum^* \longrightarrow \sum^*$ funtzio konputagarriarekin, $\mathbf{A}$=ran(ξ) adierazpena da gure abiapuntua, eta $\mathbf{A} \in \Sigma_1$ betetzen dela erakutsi behar dugu. $\chi_{\mathbf{A}}$ funtzioa konputagarria dela frogatzeko honako baliokidetasunak behatzen ditugu:

$$\chi_A(\mathbf{x}) \cong \begin{cases} \text{true} & \mathbf{x} \in \mathbf{A} \Leftrightarrow \mathbf{x} \in \text{ran}(\xi) \Leftrightarrow \exists \mathbf{z}\ \xi(\mathbf{z}) = \mathbf{x} \\ \perp & \text{bestela} \end{cases}$$

Funtzio hau konputatzeko $\xi(0)$, $\xi(1)$, $\xi(2)$, $\xi(3)$, ... balio zerrenda kalkula dezakegu, paraleloan, **x**-rekin bat etorriko den bat topatu arte. Agertzen ez bada, gure bilaketa ez da bukatuko, baina horrek ez du garrantzirik izango, **x**=ran(ξ) dela esan nahiko baitu. Orduan bedi **b**, ξ funtzio konputagarriaren indize bat. $\chi_{\mathbf{A}}$ funtzioa konputatuko duen programa honakoa izango da:

```
BIKOTE := ε;
while not E(B, deskod_2_1(BIKOTE), deskod_2_2(BIKOTE), X1)  loop
      BIKOTE := hur(BIKOTE);
end loop;
X0 := true;
```

eta ondorioz $\mathbf{A} \in \Sigma_1$ betetzen da. Honekin frogapen zirkularra osatu dugu eta bost baldintzen baliokidetasuna erakutsi dugu, beraz horietako edozein adierazpen multzo sasierabakigarrien karakterizazioa da. ⌫

Oraintxe frogatu dugun teoremaren **b)-e)** karakterizazioak multzo sasierabakigarrien definizio alternatiboak besterik ez dira, eta guztiek multzo horien gaineko ezaugarri interesgarriak adierazten dizkigute.

Adibidez, **c)** aukerak, edozein **A** multzo sasierabakigarri **P** predikatu erabakigarri baten kuantifikazio existentziala bezala jar daitekeela esaten duenean, **A**-ren izaerari buruzko oso ideia intuitiboa ematen ari zaigu. $\mathbf{x} \in \mathbf{A}$ barnekotasun baldintza beti $\exists \mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y})$ motako formula baten arabera adieraz daitekeela esaten digu, eta ondorioz horrek iradokitzen digu **x** balioa **A**-n dagoenentz jakiteko **y**-ren balizko balioen gainean *bilaketa* egin behar dugula, $\mathbf{P}(\mathbf{x},\mathbf{y})$ beteko duen bat topatu arte. Bilaketaren nozio honek ondoko ideia ongi islatzen du: multzo sasierabakigarri baten elementuak detektagarriak dira, bilaketak arrakasta badu, baina kanpokoak agian ez dira hala izango, bilaketa-metodoak ez duelako beraientzat bukatzen. Beste elementu horiek detektatzeko agian beste metodo alternatiboren bat egon daiteke (multzoa erabakigarria bada gainera), edo agian ez.

b) karakterizazioak multzo sasierabakigarriak eta funtzio konputagarrien domeinuak gauza bera direla ziurtatzen digu. $\mathbf{W_e}$ = dom(φ_e) notazioa onartu dugunez, multzo sasierabakigarri guztien klasea (Σ_1) zerrenda infinitu baten bitartez jar dezakegula aurkitzen dugu, hots, $\Sigma_1 = \{ \mathbf{W_0}, \mathbf{W_1}, \mathbf{W_2}, \mathbf{W_3}, \mathbf{W_4}, \ldots \}$. Zerrenda honetan, noski, multzo zehatz bakoitza aldi askotan errepikatuta agertzen

da: adibidez, $\sum^*$ multzo sasierabakigarria funtzio osoren bati lotutako posizio guztietan azaltzen da (eta badakigu horietatik infinitu daudela eta bakoitza infinitu aldiz errepresentatzen dela, berau konputatzeko infinitu programa desberdin daudelako). Zerrendatze horri esker multzo sasierabakigarriei *indizeak* lot diezazkiekegu, funtzio konputagarriei egiten genien bezala, honakoa betetzen delako: $\mathbf{A} \in \Sigma_1 \Leftrightarrow \exists \mathbf{e}\ \mathbf{A} = \mathbf{W_e}$.

e) karakterizazioarekin ere oso antzera gertatzen da, baina kasu honetan Σ_1 klasearen zerrendatze alternatiboa lortzeko $\mathbf{R_e} = \text{ran}(\varphi_e)$ notazio estandarrean oinarritzen gara, hots, $\Sigma_1 = \{ \mathbf{R_0}, \mathbf{R_1}, \mathbf{R_2}, \mathbf{R_3}, \mathbf{R_4}, \ldots \}$.

Hala ere, sasierabakigarritasunaren formulazioetakoren batek nabarmentzea merezi baldin badu, hori **d)** izango da, multzo hutsa den kasu nabarian izan ezik, multzo sasierabakigarri bat **f** funtzio oso eta konputagarri baten heina bezala karakterizatzen duena. **A** multzo sasierabakigarria hartuta, berau **f**-ren heina bezala adierazteko aukera izateak multzoarekin lan egiteko oso-oso tresna baliotsua ematen digu, **f** funtzioak **A** multzoaren elementu guztiak zerrendatzea ahalbidetzen baitigu, hots, $\mathbf{A} = \text{ran}(\mathbf{f}) = \{ \mathbf{f}(0), \mathbf{f}(1), \mathbf{f}(2), \mathbf{f}(3), \mathbf{f}(4), \ldots \}$. Intuizioz honako irakaspena atera dezakegu: Σ_1-eko multzoak, beren elementu guztiak *zerrendatzeko* metodo konputagarria edukitzeagatik bereizten dira. Honek esan nahi du $\mathbf{x} \in \mathbf{A}$ bada, orduan noizbait zerrenda horretan azalduko dela (posizio batean baino gehiagotan agertzea litekeena da), eta horregatik **f**-ri **A** multzoaren *zerrendatze funtzioa* deitzen zaio.

Ezaugarri honen alderdi interesgarrienetako bat emango diogun erabilpena da, diagonalizazio metodoarekin konbinatuz, multzo bat sasierabakigarria ez dela frogatzeko erabiliko baitugu.

6.3.4 Diagonalizazioa eta ez-sasierabakigarritasuna

Orain arte, zenbait multzo erabakigarri edo sasierabakigarri direla frogatzeko baliagarriak izan daitezkeen propietateak enuntziatu eta frogatzen jarri dugu arreta, baina Konputagarritasunaren Teoriaren ikuspuntutik gehien interesatuko zaigun erronka kontrakoa izatea, hain zuzen, itxaron daiteke: **A** multzo bat erabakiezina dela frogatzea, edo sasierabakigarria ez dela ere. Horretarako bere ezaugarri funtzioa, $\mathbf{C_A}$, edo bere erdiezaugarri funtzioa, χ_A, konputaezina dela frogatu beharko dugu, eta konputaezintasunak erakusteko dugun metodo bakarra diagonalizazioarena denez, mekanismo horri eutsi beharko diogu.

Hala ere, xehetasunetan arreta pixka bat jarriz gero, diagonalizazio metodoa orain arte funtzio osoen (predikatuak, oro har) konputaezintasuna frogatzeko aplikatu izan dela behatuko dugu. Era berdinean egiten jarraituz gero ezaugarri

funtzioentzat metodoak ondo funtzionatuko duela itxarotea arrazoizkoa dirudi, horiek osoak baitira, baina, hala ez diren erdiezaugarri funtzioentzat portaera berdina edukitzea espero al dezakegu? Erantzuna ezezkoa da, eta arrazoia oso sinplea da: ez-osoa den ψ funtziotik hasten bagara, zeinen konputaezintasuna erakutsi nahi dugun, δ funtzio diagonal bat eraiki behar dugu, eta horretarako ψ funtzioak kasu bakoitzean eskaintzen digun informazioa erabili behar dugu. Eta zer gertatzen da $\psi(\mathbf{x})\uparrow$ betetzen den kasuetan? Bada informazio hori ez dela existitzen eta ondorioz ezin dela erabili.

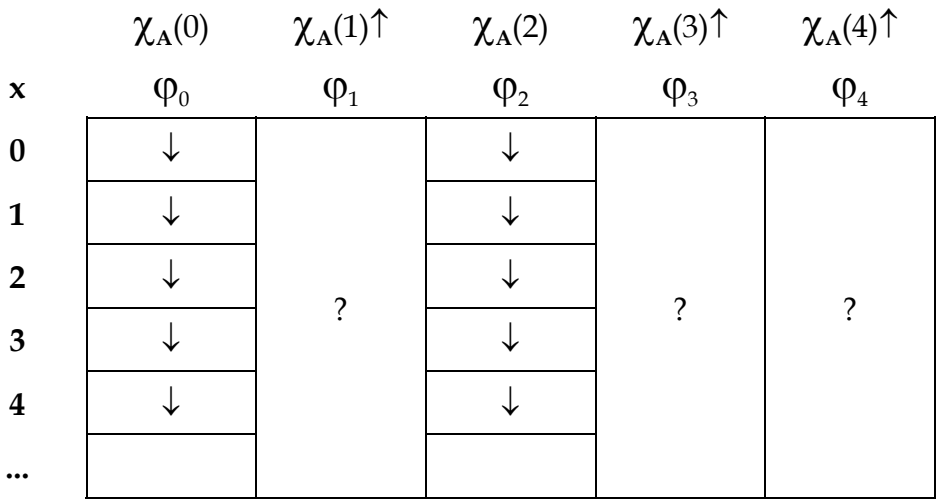
Adibide bat ikus dezagun: bedi **A** multzoa, $\mathbf{A} = \{ \mathbf{x}\in\sum^*: \varphi_\mathbf{x}$ funtzioa osoa eta mugatua da $\} = \{ \mathbf{x}\in\sum^*: \exists\mathbf{k}\forall\mathbf{y}\ \varphi_\mathbf{x}(\mathbf{y})<\mathbf{k} \}$, eta sasierabakigarria ez dela erakutsi nahi dugula demagun. Horretarako ondoko funtzioa, hots, **A**-ren erdiezaugarri funtzioa konputagarria ez dela frogatu behar dugu.

$$\chi_\mathbf{A}(\mathbf{x}) \cong \begin{cases} \text{true} & \mathbf{x} \in \mathbf{A} \Leftrightarrow \varphi_\mathbf{x} \text{ osoa eta mugatua da} \\ \perp & \text{bestela} \end{cases}$$

Demagun konputagarria dela, eta $\varphi_\mathbf{x}$ funtzio bakoitzarekin zer gertatzen denari buruz $\chi_\mathbf{A}(\mathbf{x})$ balioak eskaintzen digun informazioa aztertzen saia gaitezen:

- $\chi_\mathbf{A}(\mathbf{x})$ = true bada, edozein **y** sarreratarako $\varphi_\mathbf{x}(\mathbf{y})$-k konbergitzen duela jakingo dugu, eta gainera $\varphi_\mathbf{x}$ funtzioak berezkoa duen muga zehatz bat baino txikiagoa den balioa itzuliko duela. Orduan, δ funtzio diagonala eraikitzeko, **x** puntuan desberdina izan dadin, alternatiba ugari edukiko genuke: $\delta(\mathbf{x}) = \varphi_\mathbf{x}(\mathbf{x})+1$ egin dezakegu, edo $\delta(\mathbf{x}) \cong \perp$ ere, $\varphi_\mathbf{x}(\mathbf{x})$-k konbergitzen duela dakigulako.
- $\chi_\mathbf{A}(\mathbf{x})\uparrow$ bada, $\varphi_\mathbf{x}(\mathbf{x})$-rekin zer gertatzen denari buruz ez dugu informaziorik edukiko. Erantzunik jasotzen ez dugunez, ez dakigu ezta $\varphi_\mathbf{x}$ funtzioa osoa eta mugatua denentz ere.

Ondo uler dezagun, egoera hau ezin da 5.5.3 ataleko diagonalizazio asimetrikoaren kasuetan gertatzen zitzaigunarekin alderatu. *Orduan jasotzen genuen informazioa eskasa edo urria zen*, eta horregatik ezin genuen erantzunaren emaitza aprobetxatu $\varphi_\mathbf{x}(\mathbf{x})$-ren desberdina izango zen $\delta(\mathbf{x})$-ren balio egokia erabakitzeko. *Orain ez dugu inolako erantzunik jasotzen*, eta horregatik $\chi_\mathbf{A}(\mathbf{x})$-rekin zer gertatzen den galdetze hutsak, $\delta(\mathbf{x})$ balioa zehazteko gure prozesuak ere amaierarik ez edukitzea eragiten du. Horrela, $\chi_\mathbf{A}$ funtzioa tranpa bihurtzen da, dibergentea den kasuetan bertatik atera ezinik geratzen gara. Okerrena da $\delta(\mathbf{x})$ balioa zehaztugabe suertatzen dela unerik desegokienean: $\mathbf{x}\notin\mathbf{A}$ betetzen denean gertatzen da hain zuzen ere, eta beraz $\varphi_\mathbf{x}$ funtzioa, arazorik gabe, **x** puntuan dibergitzen duen funtzio ez osoa izan daiteke, orduan $\varphi_\mathbf{x}$ eta δ funtzioak desberdinak izateko inolako bermerik ez daukagu. Deskribatu dugun egoera hori 6.2 irudiko adibide-taulan adierazten da.



6.2 irudia: Diagonalizazio metodoa, orain arte ezagutzen dugun bezala, funtzio ez-oso batekin ezin dugula aplikatu erakusten duen adibide-taula.

Hala ere, kasu honetan eta antzekoetan aplikatu ahal izateko, diagonalizazio metodoa alda dezakegu. Ideia kontraesana beste bideren batetik bilatzean datza. **A** sasierabakigarria dela suposatzen dugunean, kontraesana bilatzeko χ_A funtzioa hartu ordez, Σ_1 klasearen karakterizazio teoremaren **d)** atala erabiliko dugu. Honek, $A \neq \emptyset$ betetzen den guztietan, **f** funtzio oso eta konputagarri bat existitzen dela esaten digu, bere heina **A**-ren berdina duena, hain zuzen ere. Funtzio osoa izatean **f** maneiatzea askoz seguruagoa izango da.

Arrazonamendu ildo hau ezarri ondoren $A = \text{ran}(f) = \{ f(0), f(1), f(2), f(3), f(4), \ldots \}$ dela argumenta dezakegu, eta ondorioz, $\{ \varphi_{f(0)}, \varphi_{f(1)}, \varphi_{f(2)}, \varphi_{f(3)}, \varphi_{f(4)}, \ldots \}$ listan osoak eta mugatuak diren funtzio konputagarri guztien zerrendatze osoa daukagula (funtzio horiek konputatzen dituzten programa guztiak **A**-n daudelako). Iskin egin ondoren, diagonalizazio metodoa aplikatzea planteatzen dugu, baina funtzio konputagarri guztien zerrenda hartu eta horien guztien desberdina den δ funtzioa definitzen saiatu ordez, funtzio konputagarri, oso eta mugatuak bakarrik hartzen dituen lista murriztu berria erabiliko dugu, **f** funtzioak ematen digun lista hain zuzen, eta gero aurreko kasuetan aplikatutako prozesu bera jarraituko dugu.

Ondoren metodoaren aldaera hau argituko dugu aipatu, **A** multzoa, hots, $A = \{ x \in \Sigma^* : \varphi_x \text{ funtzioa osoa eta mugatua da} \}$, sasierabakigarria ez dela frogatzeko.

JUSTIFIKAZIOA: Intuitiboki, $A \notin \Sigma_1$ dela susmatzea arrazoizkoa dirudi, $x \in A$ dela detektatzeko $\exists k \forall y \ \varphi_x(y) < k$ betetzen dela frogatu beharko baikenuke, eta horrek infinitu egiaztaketa egin behar izatea ekarriko luke, denbora-tarte finituan erantzun bat ematea eragozten diguna.

PLANIFIKAZIOA: Hipotesi moduan, **A** multzoa sasierabakigarria dela suposatzen dugu. Badakigu **A** ez dela hutsa, funtzio konstanteren bat konputatzen

duten programa guztiak bertan daudelako. Orduan, karakterizazio teoremarengatik, **A** multzoa **f** funtzio konputagarri eta osoaren heinarekin bat datorrela ere badakigu. Funtzio horrek funtzio konputagarri, oso eta mugatu guztien zerrendatze konputagarria eskaintzen digu, hau da, $\mathcal{L} = \{ \varphi_{\mathbf{f}(0)}, \varphi_{\mathbf{f}(1)}, \varphi_{\mathbf{f}(2)}, \varphi_{\mathbf{f}(3)}, \varphi_{\mathbf{f}(4)}, \ldots \}$ zerrendatzea.

Orain **d** funtzio diagonala definituko dugu, alde batetik *konputagarria*, *osoa* eta *mugatua* izan dadin, baina bestaldetik, $\mathcal{L}$ zerrendako funtzio guztien desberdina ere izan dadin. Denak osoak izateagatik $\mathcal{L}$-ko funtzioak maneiatzea oso erraza denez, diagonaleko balioa erabili ahal izango dugu, $\varphi_{\mathbf{f}(\mathbf{x})}(\mathbf{x})\downarrow$ beti beteko dela ziur jakinda. Horrela **d** funtzioa $\varphi_{\mathbf{f}(0)}$-tik 0 puntuan desberdinduko da, $\varphi_{\mathbf{f}(1)}$-etik 1 puntuan, etab. Lehen azaldu dugun bezala, funtzioak desberdinduarazteko modu zehatza era desberdin askotara ezar daiteke, baina kontu handiz betiere. Ezin dugu $\mathbf{d}(\mathbf{x})\uparrow$ jarri, orduan **d** funtzioa ez bailitzateke osoa izango eta gure arrazoiketa pikutara joango litzateke. Aldaketak egiteko erabilitako mekanismo klasikoetakoren bat, esaterako $\mathbf{d}(\mathbf{x}) = \varphi_{\mathbf{f}(\mathbf{x})}(\mathbf{x})+1$, ere ezin dugu sartu, orduan **d** ez delako funtzio mugatua izango[10].

Mugatua izatea lortzeko $\mathbf{d}(\mathbf{x})$-ri bi balio bakarrik hartzeko aukera emango diogu. Oro har, $\mathbf{d}(\mathbf{x}) = 3$ definituko dugu, $\varphi_{\mathbf{f}(\mathbf{x})}(\mathbf{x})$-ren balioa 3 ez den bitartean betiere, eta kasu horretan 0 balioa esleituko diogu, desberdina izan dadin. Horrela, **d** osoa eta mugatua izango da, 3-ren berdinak edo txikiagoak diren balioak itzuliz, beti konbergituko baitu. Ideia 6.3 irudiko adibide-taulan erakusten da.

ERAIKETA: **d** funtzioa formalki definituko dugu:

$$\mathbf{d}(\mathbf{x}) = \begin{cases} 0 & \varphi_{\mathbf{f}(\mathbf{x})}(\mathbf{x}) = 3 \\ 3 & \text{bestela} \end{cases}$$

f konputagarria denez, **d** ere hala izango da, honako programak erakusten duen bezala:

```
R := Φ(f(X1), X1);
X0 := 3;
if R = 3 then X0 := 0; end if;
```

Funtzio konputagarria izateagatik, $\mathbf{e} \in \sum^*$ indizea egongo da, $\mathbf{d} \cong \varphi_{\mathbf{e}}$ beteko duena. Baina, egia esan, interesatzen zaiguna ez da **e** indizea funtzio konputagarrien C_1 zerrendan dagoela egiaztatzea, baizik eta $\mathcal{L}$ zerrendan dagoela

[10] Hala da, zerrendako funtzio guztiak mugatuak egon arren, horrek ez du esan nahi horietarako guztietarako muga komuna dagoenik. $\mathbf{d}(\mathbf{x}) = \varphi_{\mathbf{f}(\mathbf{x})}(\mathbf{x})+1$ jarriko bagenu ez litzateke mugatua izango, balizko edozein **k** muga harturik $\mathbf{g}(\mathbf{x}) = \mathbf{k}$ funtzio konstantea edukiko genukeelako, konputagarria, osoa eta mugatua izanik $\mathcal{L}$ zerrendaren barnean egongo litzatekeena. Orduan **g** funtzioa, **a** balioren batentzat, $\varphi_{\mathbf{f}(\mathbf{a})}$ moduan jar litekeela esan dezagun, eta ondorioz $\mathbf{d}(\mathbf{a}) = \varphi_{\mathbf{f}(\mathbf{a})}(\mathbf{a})+1 = g(\mathbf{a})+1 = \mathbf{k}+1$ beteko litzateke, **k** muga gainditzen duen balioa, hain zuzen.

erakustea, funtzio konputagarri, oso eta mugatuen zerrendan, hain zuzen. Funtzioa definitzean $\forall \mathbf{y}\ (\mathbf{d}(\mathbf{y})=3 \vee \mathbf{d}(\mathbf{y})=0)$ betetzen dela arrazoitu dugu. Beraz, bere **e** indizea funtzio oso eta mugatu bati dagokio, eta ondorioz $\mathbf{e} \in \mathbf{A}$ da. **A** multzoa, berriz, **f**-ren heina denez, $\mathbf{m} \in \sum^*$ hitza badagoela daukagu, $\mathbf{f}(\mathbf{m})=\mathbf{e}$ betetzen duena. Hortaz, funtzio bat eraiki dugu zeinaren programa **A** multzoan dagoen, baina bestaldetik, bere eraiketan erabili dugun teknikak ziurtatzen digu programa horren portaera diagonaleko zerrendan dauden guztien portaeraren desberdina dela. Honek kontraesan bat ezartzen utziko digu.

d	**x**	$\varphi_{\mathbf{f}(0)}$	$\varphi_{\mathbf{f}(1)}$	$\varphi_{\mathbf{f}(2)}$	$\varphi_{\mathbf{f}(3)}$	$\varphi_{\mathbf{f}(4)}$	$\varphi_{\mathbf{f}(5)}$
3	**0**	0	≤ 25	≤ 3	≤ 31	≤ 6	≤ 1917
3	**1**	≤ 0	20	≤ 3	≤ 31	≤ 6	≤ 1917
0	**2**	≤ 0	≤ 25	3	≤ 31	≤ 6	≤ 1917
3	**3**	≤ 0	≤ 25	≤ 3	2	≤ 6	≤ 1917
0	**4**	≤ 0	≤ 25	≤ 3	≤ 31	3	≤ 1917
3	**5**	≤ 0	≤ 25	≤ 3	≤ 31	≤ 6	733

6.3 irudia: **f**-ren heinak eskaintzen duen funtzio konputagarrien zerrenda mugatuaren gainean **d** funtzio diagonal bat eraikitzeko adibide-taula. **f** funtzioa konputagarria eta osoa izanik $\varphi_{\mathbf{f}(\mathbf{x})}(\mathbf{x})$ balioa beti kalkula dezakegu, eta hau berriz, beti egongo da definituta, $\varphi_{\mathbf{f}(\mathbf{x})}$ funtzioa osoa eta mugatua izateagatik. Honek $\mathbf{d}(\mathbf{x})$-rentzat balio desberdin bat ematea ahalbidetuko digu.

KONTRAESANA: Gure arrazonamenduagatik ere, **e** indize hori existitzea ezinezkoa dela ikus dezagun. **d** eraikitzean kontu handia jarri dugu konputagarria ($\mathbf{d} \cong \varphi_{\mathbf{e}}$) eta gainera osoa eta mugatua ($\mathbf{f}(\mathbf{m})=\mathbf{e}$) izan dadin. Aldi berean $\varphi_{\mathbf{f}(\mathbf{x})}$ funtzio oso eta mugatu guztien desberdina egin dugu, **x** puntuan hain zuzen. Ondorioz, $\varphi_{\mathbf{f}(\mathbf{m})}$ funtziotik ere desberdinduko da, **m** puntuan. **d** funtzioa **m** puntuan aplikatzen badugu kontraesana lortzen dugula ikus dezagun:

$$\underline{\text{1. kasua:}}\ \mathbf{d}(\mathbf{m})=3 \overset{\text{def. } \mathbf{d}}{\Rightarrow} \neg\varphi_{\mathbf{f}(\mathbf{m})}(\mathbf{m})=3 \overset{\mathbf{f}(\mathbf{m})=\mathbf{e}}{\Rightarrow} \neg\varphi_{\mathbf{e}}(\mathbf{m})=3 \overset{\mathbf{d}\cong\varphi_{\mathbf{e}}}{\Rightarrow} \neg\mathbf{d}(\mathbf{m})=3\ \lightning$$

$$\underline{\text{2. kasua:}}\ \mathbf{d}(\mathbf{m})=0 \overset{\text{def. } \mathbf{d}}{\Rightarrow} \varphi_{\mathbf{f}(\mathbf{m})}(\mathbf{m})=3 \overset{\mathbf{f}(\mathbf{m})=\mathbf{e}}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{m})=3 \overset{\mathbf{d}\cong\varphi_{\mathbf{e}}}{\Rightarrow} \mathbf{d}(\mathbf{m})=3\ \lightning$$

Kontraesana hasierako hipotesitik, hots, **A** sasierabakigarria dela suposatzetik, sortzen dela ondoriozta dezakegu eta horrela $\mathbf{A} \notin \Sigma_1$ dela frogatuta geratzen da.

Diagonalizazio metodoaren hedapen honen bitartez multzo anitz ez-sasierabakigarri bezala sailkatzeko aukera izango dugu.

7. Ondorioak

Orrialde hauetan zehar Informatikaren oinarri teorikoak ulertzeko funtsezkoa den galdera bati erantzun diogu: algoritmoen bitartez ebazteko aukera izango ez dugun problemak ba al daude? Problema horiek egon badaudela frogatuz baietz erantzutea ez da guretzat nahikoa izan, eta zenbait adibide zehatz ere aztertu ditugu, geratze-problema oinarrizko bezala nabarmenduz (hau da, programa batek bere exekuzioa noiz bukatuko duen aurretik, *a priori*, zehaztearen problema). Azken finean, konputazioaren oinarrizko lege bat baliozkotzeko aukera izan dugu: programen portaera aurreratzeko laburbiderik ez dagoela, hots, portaera, berez, aurretik ezagutzea ezinezkoa dela, esaten duen legea. Konputazio baten emaitza jakiteko erarik zuzenekoena, *oro har*, hura pausoz pauso jarraitzea da. Emaitza horrekin programa bukatzen dela egiazta dezakegu, baina ez dela bukatzen frogatzeko metodo finiturik ez dago.

Aurretik ezin jakite horrek lehen mailako ondorio praktiko batzuk dauzka. Adibidez, badakigu konpilatzaileak programetako zenbait huts-egite mota (errore sintaktikoak deitzen direnak) detektatzeko oso eraginkorrak direla, eta, aldiz, itxuraz oso konplexuak ere ez diruditen beste zenbait gertakizun (adibidez, taula baten indizea bere tartetik atera dela) aurreikusteko ezgauza ageri dira. Honen arrazoia da exekuzioan gertatzen diren erroreen izaeran geratze-problemaren erabakiezintasun bera dagoela: errore horiek gertatzen diren ala ez jakiteko metodo orokor bakarra programaren exekuzioa burutzea da, eta hau da, azken finean, konpilatzaileak egiten duena: aipatu errore horiek aurretik zehaztea ezinezkoa denez, programa itzultzen digu, erroreekin exekuzio garaian topa gaitezen, ematen badira betiere!

Programen portaeren aurreikuspenarekin oso erlazionatuta dauden beste problema batzuk existitzen dira, *software*aren ekoizpen prozesuari lotutakoak zuzen-zuzenean. Programa batek eman zaizkion zehaztapenei kasu egingo dien (hau da, bera diseinatzeko orduan aurreikusi den hori egingo duenentz) erabakitzea oro har ez da posible. Ezin da ezta bi programak helburu berdinetarako balioko ote duten ere zehaztu, eta hortaz ezin ditugu tresnak eraiki zeintzuek, programa bati azken birfinketak edo aldaketak egin ondoren, lehenago zeukan sarrera/irteera portaera berdina izaten jarraituko ote duen egiaztatuko luketen.

Programen semantikarekin erlazionatutakoa eta konputaezina izatea gertatu den beste problema bat honakoa da: programa bat emanik, problema berdina ebatziko luken beste bertsio motzago bat aurkitu. Eta horrela, hainbat adibide zerrendatzen jarraitu genezakeen (ez ahaztu erabakiezinak diren infinitu multzo existitzen direla).

Berez, frogatzen zaila ez den oso emaitza deigarria dago, programen portaera funtzionalarekin erlazionatutako propietate ***guztiak*** erabakiezinak direla esaten duena. Hau da, algoritmo baten sarreren eta irteeren arteko erlazioari dagokion edozein galdera ezingo dela algoritmikoki ebatzi. Neurri batean Ingeniaritza Informatikorako paradoxa gertatzen da, hainbat eremutako problemak ebazteko hain tresna baliogarria dela frogatu ondoren, bere area bera izatea bere muga edo murriztapenak gehiago sufritzen dituena.

Problema horien guztien konputaezintasunari heltzeko diagonalizazioaren teknika berriz, motz geratzen da. Kasu askotan beste bat behar izaten da, ***laburtze teknika***, honek problema batzuen erabakiezintasuna beste batzuenarekin erlazionatzea ahalbidetzen du, eta "zailtasun konputazional"aren hierarkia ezartzen doa. Hierarkia horren azpiko elementuak dira idazlan honetan azaldu ditugunak. Horrela, multzo erabakigarriak posible diren "errazenak" dira, eta beraien gainetik erabakigarriak ez diren problema sasi-erabakigarriak geratzen dira. Baina aurrerago ere joan daiteke, sasierabakigarriak ez diren multzoen artean ere "zailagoak" eta "errazagoak" direnak ere badaudelako, hierarkia infinitu bat ezarriz berez. Hau da, oso arrotza dirudien arren, beste batzuk baino "erabakiezinagoak" diren multzoak badaude, eta gainera "erabakiezintasun-mailetan" nahi adina gora aurki ditzakegu.

Hala ere, konputaezintasunaren munduan murgiltzea eta bertan aurki ditzakegun problemen kopuru itzelaz ohartzea harrigarria suertatzen den arren, konputagarriak diren problemen inguruan sortu diren hainbat ideia ez dira gutxiestekoak, inondik ere. Ez dugu ahaztu behar konputazioak problema zailak ebazteko erabiltzen ohituta gauden tresna boteretsua izaten jarraitzen duela. Gertatzen dena da pixka bat desberdina den optika batekin ikus dezakegula orain, erabat erraza dela gertutik egiaztatu dugulako: erabateko xumea den lengoaia programazio ingurune barrokoenak adieraz ditzakeen algoritmo berak adierazteko gai da. Informatikan den esperientziak frogatu du oso funtsezkoa den eragiketa-multzoak funtzio konputagarri guztiak programatzeko nahikoa dela; eta funtzio horiek lista baten moduan zerrenda daitezke. Programazioko beste mekanismo batetik edo konputazioko beste eredu alternatibo batzuetatik abiatu izan bagina, ***programazio-sistema unibertsal eta onargarri*** batera helduko ginen ere. Desberdina izango zen, baina hemen azaldu ditugun emaitza berak ezartzeko bidea emango zigun betiere.

A eranskina: while-programen lengoaia

While-programak, $\sum = \{\mathbf{a_1}, \ldots, \mathbf{a_n}\}$ alfabetotik abiatuz eraikitako hitzen gainean lan egiten duten aginduz osatuta daude. Hitz horiek erreferentziatzeko, programak **X0**, **X1**, **X2**, ... erako aldagaiak erabiltzen ditu. Programa batek erabil dezakeen aldagai-kopuruaren inguruan mugarik ez dago, ezta programaren exekuzioan zehar hitz horiek har dezaketen tamainan ere.

While-programen multzoa $\mathbb{W}$ ikurraren bidez adierazten dugu, eta induktiboki era honetan definitzen da:

Baldin ...	**orduan...**
$\mathbf{i} \in \mathbb{N}$	XI := ε;
$\mathbf{i}, \mathbf{j} \in \mathbb{N}$ $\mathbf{s} \in \sum$	XI := $\mathbf{cons_s}$(XJ);
$\mathbf{i}, \mathbf{j} \in \mathbb{N}$	XI := cdr (XJ);
$\mathbf{P_1}, \mathbf{P_2} \in \mathbb{N}$	P_1 P_2
$\mathbf{i} \in \mathbb{N}$ $\mathbf{s} \in \sum$ $Q \in \mathbb{W}$	**if** $\mathbf{car_s}$?(XI) **then** Q **end if**;
$\mathbf{i} \in \mathbb{N}$ $Q \in \mathbb{W}$	**while** nonem?(XI) **loop** Q **end loop**;

... while-programa da

While-programa baten semantika honakoa da:

1. Programak **n** datu baditu, eragiketak egiten hastean horiek jadanik **X1**, **X2**, …, **XN** aldagaietan biltegiratuta daudela suposatzen da
2. Programako gainontzeko aldagaiak ε (hitz hutsa) balioarekin hasieratzen dira
3. Programako aginduak aldagaien gainean egikaritzen dira
4. Programa bukatzen denean (hala eginez gero), **X0** aldagaiaren bukaerako edukia konputazioaren emaitza bakartzat hartzen da
5. Aldiz, programa bukatzen ez bada, konputazioaren emaitza indefinitutzat hartzen da

Definitutako programa bakoitzaren esanahi informala honakoa da:

XI := ε;	**XI** aldagaian hitz hutsa sartu
XI := $cons_s$(XJ);	**XJ**-n dagoen hitzaren ezkerretik **s** ikurra erantsi eta lortutako balioa **XI**-ri esleitu
XI := cdr (XJ);	**XJ**-n dagoen hitzaren ezkerreko lehen ikurra ezabatu eta lortutako balioa **XI**-ri esleitu
P_1 P_2	$\mathbf{P_1}$ eta $\mathbf{P_2}$ programak sekuentzialki egikaritu
if car_s?(XI) **then** Q **end if**;	**Q** programa egikaritu, baldin eta **XI**-ren ezkerreko lehen ikurra **s** bada
while nonem?(XI) **loop** Q **end loop**;	**XI**-ren edukia hitz hutsa ez den bitartean, **Q** egikaritu

B eranskina: Makroak

Makroa while-programentzako laburdura mekanismoa da, kodearen errepikaketa ekiditen duena. Makro mota bakoitzaren definizioa osatzeko beharrezkoa da bere ***hedapena*** ematea, hau da, makro hori while-programa baliokide batean bihurtzeko era [IIS 98]. Hortaz, makroa while-programa horren laburdura besterik ez dela kontsideratzen da.

Makroak erabiliz honako formatua duten aginduak idatz ditzakegu:

- $U := \psi(V_1, ..., V_K)$;

  non $\mathbf{U}$, $\mathbf{V_1}, ..., \mathbf{V_K}$ edozein aldagai edo identifikatzaile izan daitezkeen (identifikatzaileei ***makroaldagai*** deitzen diegu eta idazkera erraztearren erabiltzen ditugu), eta ψ derrigorrez while-konputagarria izan behar den funtzioa den. Makro hauek (zeintzuen eskuineko aldeari ***makroadierazpen*** esaten diogun) erabilgarriak dira programatzen goazen funtzioak, jadanik lengoaian sartuta egongo balira bezala, berrerabiltzen joateko aukera ematen dutelako[11]. Dokumentu honen programetan erabilitako funtzio gehienak C eranskineko zerrendan biltzen ditugu, beren while-konputagarritasuna jadanik [IIS 98] dokumentuan frogatua izan zen.

- **if** $R(V_1, ..., V_K)$ **then** Q **end if**;
  edo
  while $R(V_1, ..., V_K)$ **loop** Q **end loop**;

  non **R** derrigorrez while-erabakigarria den predikatua den, eta lehen bezala $\mathbf{V_1}, ..., \mathbf{V_K}$ aldagaiak edo makroaldagaiak izan daitezkeen, eta **Q** makroprograma bat den. Makro hauetan sartzen diren ***makrobaldintzen*** erabilgarritasuna, programatzen goazen predikatuak, jadanik lengoaian sartuta egongo balira bezala, berrerabiltzen joateko aukera ematean datza[11]. Dokumentu honen programetan erabilitako predikatu gehienak C eranskineko zerrendan biltzen ditugu, beren while-erabakigarritasuna jadanik [IIS 98] dokumentuan frogatua izan zen.

---

[11] Makroadierazpenetan funtzio while-konputagarriak eta makrobaldintzetan predikatu while-erabakigarriak erabiltzean, programazio-lengoaietan ohikoenak diren, eta era berean hitzarmen matematikoetatik heredatutakoak diren, notazioak ahalik eta gehien errespetatzen saiatzen gara. Horrela, & kateamendu funtzioa edo = berdinketa predikatua bezalako eragiketak beren ohiko notazio infixu edo artizkiarekin erabiltzen dira: adibidez, X7:=AUX & X2; jartzen dugu X7:=&(AUX, X2); honen ordez. Era berean, while-konputagarriak diren funtzio konstanteei eta identitate-funtzioari deiak ohiko eran egiten dira: horrela, X4:=IND; eta P:='abbac'; idatziko dugu.

- **if** B_1 **then** Q_1 **elsif** B_2 **then** Q_2 **... elsif** B_n **then** Q_n **else** Q_{n+1} **end if**;

  non $\mathbf{B_1}$, ..., $\mathbf{B_n}$ makroadierazpenak diren eta $\mathbf{Q_1}$, ..., $\mathbf{Q_{n+1}}$ makroprogramak diren.

- **for** V **in** A..B **loop** Q **end loop**;

  non **V** aldagaia edo makroaldagaia den, **A** eta **B** makroadierazpenak diren eta **Q** makroprograma den, **V** alda dezakeen esleipenik ez daukana.

Agindu hauei esker, idazten ditugun programek, Ingeniaritza Informatikoko ikasleentzat ohikoagoak diren programazio-lengoaien bitartez eraikitakoetatik hurbilago dagoen itxura hartzen dute, baina makroprograma bakoitzarentzat while-programa baliokidea existitzen dela gogoratzea garrantzitsua da.

C eranskina: Funtzio konputagarrien eta predikatu erabakigarrien zerrenda

Jarraian, testuan erabiltzen diren funtzioen eta predikatuen zerrenda ematen dugu, beren while-konputagarritasuna eta while-erabakigarritasuna frogatutzat emanez [IIS 98]. Lehenik eta behin, $\sum = \{\mathbf{a_1}, \ldots, \mathbf{a_n}\}$ alfabeto orokor baten gaineko hitzekin lan egiteko erabiltzen ditugun eragiketak sartzen ditugu.

a) Hitzekin erabilitako eragiketa ohikoenak

- $cons_s : \sum^* \to \sum^*$ funtzioa definitzen da

$$\text{cons}_{\mathbf{s}}(\mathbf{w}) = \mathbf{s} \bullet \mathbf{w}$$

- $cdr : \sum^* \to \sum^*$ funtzioa definitzen da

$$\text{cdr}(\mathbf{w}) = \begin{cases} \varepsilon & \mathbf{w} = \varepsilon \\ \mathbf{v} & \exists \mathbf{s}\ \mathrm{w} = \mathbf{s} \bullet \mathbf{v} \end{cases}$$

- $car_{\mathbf{s}}? : \sum^* \to \mathbb{B}$ predikatua definitzen da

$$\text{car}_{\mathbf{s}}?(\mathbf{w}) = \begin{cases} \text{true} & \mathbf{w} = \mathbf{s} \bullet \mathbf{v} \\ \text{false} & \text{bestela} \end{cases}$$

- $nonem? : \sum^* \to \mathbb{B}$ predikatua definitzen da

$$\text{nonem?}(\mathbf{w}) = \begin{cases} \text{false} & \mathbf{w} = \varepsilon \\ \text{true} & \mathbf{w} \neq \varepsilon \end{cases}$$

- $\perp\!\!\!\perp : \sum^* \to \sum^*$ funtzio *hutsa* definitzen da

$$\perp\!\!\!\perp(\mathbf{w}) \cong \perp$$

- $\cdot^R : \sum^* \to \sum^*$ *alderantzizko* funtzioa definitzen da

$$\mathbf{w}^R = \begin{cases} \varepsilon & \mathbf{w} = \varepsilon \\ \mathbf{u}^R \bullet \mathbf{s} & \mathbf{w} = \mathbf{s} \bullet \mathbf{u} \end{cases}$$

- $\& \cdot : \sum^* \times \sum^* \to \sum^*$ *kateamendu* funtzioa definitzen da

$$\varepsilon \,\&\, \mathbf{y} = \mathbf{y}$$

$$\mathbf{s} \,\&\, \mathbf{y} = \text{cons}_{\mathbf{s}}(\mathbf{y})$$

$$\text{cons}_s(\mathbf{x}) \,\&\, \mathbf{y} = \text{cons}_s(\mathbf{x} \,\&\, \mathbf{y})$$

Funtzio hau testuan, programetatik kanpo, ere erabili ohi da, notazio algebraikoa jarraituz, hots, **x**&**y** adierazpenaren ordez **x**•**y** jarriz.

- *lehena* : $\sum^* \to \sum^*$ funtzioa definitzen da

$$\text{lehena}(\mathbf{w}) = \begin{cases} \varepsilon & \mathbf{w} = \varepsilon \\ \mathbf{s} & \mathbf{w} = \mathbf{s} \bullet \mathbf{u} \end{cases}$$

- **w** hitz bakoitzarentzat, $K_{\mathbf{w}} : \sum^* \to \sum^*$ funtzio *konstantea* definitzen da

$$K_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}$$

- *hur* : $\sum^* \to \sum^*$ funtzioa, zeinek hitz bat harturik honako baldintzak betetzen dituen hurrengoa itzultzen duen: a) luzera errespetatzen duen ordena jarraitzen da eta b) luzera bereko bi hitzen artean ordena lexikografikoa mantentzen da. Hots, funtzioa honela definitzen da:

$$\text{hur}(\varepsilon) = '\mathbf{a_1}'$$

$$\text{hur}(\mathbf{w} \bullet \mathbf{a_i}) = \begin{cases} \mathbf{w} \bullet \mathbf{a}_{\mathbf{i+1}} & \mathbf{i} < \mathbf{n} \\ \text{hur}(\mathbf{w}) \bullet \mathbf{a}_1 & \mathbf{i} = \mathbf{n} \end{cases}$$

- *aurre* : $\sum^* \to \sum^*$ funtzioa, zeinek $\sum^*$ multzoan definitutako ordenaren arabera aurreko hitza itzultzen duen, honela definitzen da

$$\text{aurre}(\mathbf{x}) = \begin{cases} \varepsilon & \mathbf{x} = \varepsilon \\ \text{aurre}^{-1}(\mathbf{x}) & \mathbf{x} \neq \varepsilon \end{cases}$$

b) Kodeketa funtzioak

kod^2 : $\sum^* \times \sum^* \to \sum^*$ funtzioa definitzen dugu, ***kodeketa*** funtzioa, zeinek hitz-bikote bakoitzarekin hau era unibokoan adierazten duen hitz bakar bat lotzea ahalbidetzen duen. Funtzioa induktiboki honela definitzen da:

$$\text{kod}^2(\varepsilon,\varepsilon) = \varepsilon$$

$$\text{kod}^2(\text{hur}(\mathbf{w}), \varepsilon) = \text{hur}(\text{kod}^2(\varepsilon, \mathbf{w}))$$

$$\text{kod}^2(\mathbf{v}, \text{hur}(\mathbf{w})) = \text{hur}(\text{kod}^2(\text{hur}(\mathbf{v}), \mathbf{w})$$

Gure lengoaian datu-egiturak adierazteko, kodeketa funtzioa guztiz baliozkoa da.

Bikote bati dagokion kodetik abiaturik, bere osagaiak berreskuratzeko alderantzizko bi **deskodeketa** funtzio, hots, $deskod_{2,1}$, $deskod_{2,2} : \sum^* \to \sum^*$, definitzen dira. Bi funtzio hauek betetzen dute:

$$kod^2(deskod_{2,1}(\mathbf{w}), deskod_{2,2}(\mathbf{w})) = \mathbf{w}$$

Ideia bera **k** hitzeko tuplen kodeketa definitzeko zabal dezakegu. $\mathbf{k}\geq 1$ bakoitzerako $kod^k : \sum^{*\mathbf{k}} \to \sum^*$ kodeketa funtzioak argumentu-kopuruaren arabera era induktiboan definitzen dira:

$$kod^1(\mathbf{z}) = \mathbf{z}$$

$$kod^{k+1}(\mathbf{z_1}, \ldots, \mathbf{z_{k+1}}) = kod^2(\mathbf{z_1}, kod^k(\mathbf{z_2}, \ldots, \mathbf{z_{k+1}})) = kod^2(\mathbf{z_1}, kod^2(\mathbf{z_2}, \ldots kod^2(\mathbf{z_{k-1}}, \mathbf{z_k})\ldots)))$$

Kodeketa funtzio guztiak bijektiboak (hots, osoak, injektiboak eta supraiektiboak) dira [IIS 98] eta beren $deskod_{k,i}$ deskodeketa funtzio propioak dituzte. $deskod_{k,i} : \sum^* \to \sum^*$ funtzioak, hitzen batek kodetzen duen **k**-tuplari dagokion **i**-garren elementua itzultzen du (non, $1 \leq \mathbf{i} \leq \mathbf{k}$), hots, honela definitzen da:

$$deskod_{\mathbf{k},\mathbf{i}}(\mathbf{w}) = \begin{cases} \mathbf{w} & \mathbf{i} = 1 \wedge \mathbf{k} = 1 \\ deskod_{2,1}(\mathbf{w}) & \mathbf{i} = 1 \wedge \mathbf{k} > 1 \\ deskod_{\mathbf{k}-1,\mathbf{i}-1}(deskod_{2,2}(\mathbf{w})) & \mathbf{i} > 1 \wedge \mathbf{k} > 1 \end{cases}$$

c) Beste datu-mota batzuei lotutako funtzioak

Gure makroprogrametan hitzenak ez ezik, beste datu-mota batzuei dagozkien eragiketak erabili ahal izateko badugu beste hitzarmen bat. Datu-mota horiek zenbaki arruntena $\mathbb{N}$, boolearrena $\mathbb{B}$, pilena $\mathbb{P}$, bektore dinamikoena $\mathbb{V}$ eta while-programena $\mathbb{W}$ dira eta aipatu hitzarmen horri esker mota horien balioak gure programetan erabili eta maneiatu ahal izango ditugu. Noski, balioak hitzen laburdurak besterik ez dira, hitzek balio horiek adierazten baitituzte.

Zenbaki arruntekin lan egiteko, 1 batzen duen *suc* eragiketa eraikitzaileaz gain oinarrizko eragiketetako batzuk erabiltzen ditugu: *pred* (1 kentzen duena, eta $pred(0)=0$ egiten duena), batuketa (+), biderketa (*), zatiketa (/), berreketa (**). Ordenaren konparaketarako eragileak ($<$, $\leq$, $>$, $\geq$) ere erabiliko ditugu.

Boolearrak eragiketa logikoak inplementatzeko definitzen dira eta horien artean honakoak erabiltzen ditugu: ezeztapena ($\neg$), konjuntzioa ($\wedge$), disjuntzioa ($\vee$), baliokidetasuna edo berdintza logikoa ($\leftrightarrow$).

Pilak eta bektore dinamikoak datuak, kasu honetan hitzak, biltegiratzeko gailuak dira. Lehenengoak, karaktere-kate batzuen agregazio ordenatuaren bitartez

eratzen dira (katerik ez egotea ere onartzen da) eta bere osagaiak "<" eta "]" ikurren artean zerrendatuz errepresentatzen ditugu, adibidez < **aba**, **bb**, ε, **a**] edo <] (azken honi *hitz hutsa* esaten diogu). Bektore dinamikoetan, piletan ez bezala, edozein osagai edozein unetan atzi daiteke, horretarako bere indizea erabiliz. Bektoreen indizeak 0-tik aurrera hasten dira edo bektore guztiek gutxienez osagai bat dute. Bektore dinamikoa errepresentatzeko bere osagaiak "(" eta ")" ikurren artean zerrendatzen ditugu, adibidez (**aba**, **bb**, ε, **a**).

Pila datu-motarentzat, ohikoak diren bere oinarrizko eragiketak erabiltzen ditugu:

- *pilaratu* : $\sum^* \times \mathbb{P} \to \mathbb{P}$ funtzioa, zeinek pila baten tontorrean hitz berria gehitzen duen:

$$\text{pilaratu}(\mathbf{v}, <\mathbf{z_1}, \ldots, \mathbf{z_n}]) = <\mathbf{v}, \mathbf{z_1}, \ldots, \mathbf{z_n}]$$

- *P_hutsa?* : $\mathbb{P} \to \mathbb{B}$ predikatua, pila batek noiz ez daukan osagairik adierazten duena:

$$\text{P\_hutsa?}(<\mathbf{z_1}, \ldots, \mathbf{z_n}]) = \text{true} \Leftrightarrow \mathbf{n}=0$$

- *tontorra* : $\mathbb{P} \to \sum^*$ funtzioa, zeinek pila batean sarturiko azken elementua ateratzen duen:

$$\text{tontorra}(<]) \cong \perp$$

$$\text{tontorra}(\text{pilaratu}(\mathbf{v}, <\mathbf{z_1}, \ldots, \mathbf{z_n}])) = \mathbf{v}$$

- *despilatu* : $\mathbb{P} \to \mathbb{P}$ funtzioa, pila baten tontorra ezabatzen duena:

$$\text{despilatu}(<]) \cong \perp$$

$$\text{despilatu}(\text{pilaratu}(\mathbf{v}, <\mathbf{z_1}, \ldots, \mathbf{z_n}])) = <\mathbf{z_1}, \ldots, \mathbf{z_n}]$$

Bektore dinamikoei dagokionez, honako eragiketak kontuan har ditzagun:

- *azken_indizea* : $\mathbb{V} \to \mathbb{N}$ funtzioa, zeinek bektore batean indize bidez atzi daitekeen azken elementuaren posizioa (hots, bere osagai-kopurua baino bat gutxiago) ematen duen:

$$\text{azken\_indizea}((\mathbf{z_0}, \mathbf{z_1}, \ldots, \mathbf{z_n})) = \mathbf{n}$$

- *edukia* : $\mathbb{V} \times \mathbb{N} \to \sum^*$ funtzioa, bektorearen osagai bat bere indizearen bitartez berreskuratzeko aukera ematen duena:

$$\text{edukia}((\mathbf{z_0}, \mathbf{z_1}, \ldots, \mathbf{z_i}, \ldots, \mathbf{z_n}), \mathbf{i}) = \mathbf{z_i} \quad \text{baldin } 0 \le \mathbf{i} \le \mathbf{n}$$

$$\text{edukia}((\mathbf{z_0}, \mathbf{z_1}, ..., \mathbf{z_n}), \mathbf{i}) \cong \perp, \quad \text{baldin } \mathbf{i>n}$$

Makroprogrametan, edukia(**v**, **i**) adierazteko, oro har, **v**(**i**) adierazpena erabiliko da.

- *aldatu* : $\mathbb{V} \times \mathbb{N} \times \sum^* \to \mathbb{V}$ funtzioa, zeinek bektore batean, esandako posizioan, balio berria sartu eta lortutako bektorea itzultzen duen:

$$\text{aldatu}((\mathbf{z_0}, \mathbf{z_1}, ..., \mathbf{z_i}, \ldots, \mathbf{z_n}), \mathbf{i}, \mathbf{w}) = ((\mathbf{z_0}, \mathbf{z_1}, ..., \mathbf{z_{i-1}}, \mathbf{z_i}, \mathbf{z_{i+1}}, \ldots, \mathbf{z_n}) \quad \text{si } \mathbf{i} \leq \mathbf{n}$$

$$\text{aldatu}((\mathbf{z_0}, \mathbf{z_1}, ..., \mathbf{z_n}), \mathbf{i}, \mathbf{w}) \cong (\mathbf{z_0}, \mathbf{z_1}, ..., \mathbf{z_n}, \varepsilon, \ldots, \varepsilon, \mathbf{w}) \quad \text{si } \mathbf{i>n}$$

Nabarmentzekoa da, indizea onartutakoa baino handiagoa denean *edukia* eragiketak errorea sortzen duen bitartean, *aldatu*-ren kasuan egoera horrek, esandako posizioan balio berria sartu ahal izateko, bektore dinamikoa zabaltzea eragiten duela.

Ohikoak diren V:= aldatu(V, I, W); erakoa esleipenak, V(I):= W; makroarengatik ordezkatuak izango dira.

d) Makroprogrametan eragiketak erabiltzeko beste era batzuk

Zenbait funtzio eta predikatu edozein datu-motarentzat defini daitezke, esaterako, funtzio hutsa ($\perp\!\!\!\perp$) eta berdintza (=) eta desberdintza predikatuak (≠).

Ezagunak diren beste batzuen konposaketa eginez funtzio while-konputagarri berriak lor ditzakegu. Era berean, eragiketa while-konpugarriak eta osoak predikatu while-erabakigarriekin ere konposa daitezke, makrobaldintzatan erabili ahal izateko. Azkenik, oraindik konplexuagoak diren makrobaldintzak eraikitzeko **not**, **and**, **or** eragile logikoak erabil ditzakegu. Azken batez, programak eraikitzeko habiatutako adierazpenak jar daitezke, honakoan egin dugun modura:

```
while  not P_hutsa?(PILA) or LAG<K**2  loop
      PILA := despilatu (PILA);
      LAG := LAG+1;
end loop;
```

Makroaldagai baten edukia datu-mota baten edozein elementu izan daitekeela kontuan hartuz, eta gainera edozein funtzio while-konputagarri aplikatzearen emaitza eslei diezaiokegula, honako esleipena idaztea ere egokia da

```
LAG:= X5=28 ;
```

LAG aldagaiari boolear motako balio bat esleitzen ariko ginatekeelako. Orduan, adierazpen boolearrak baldintzetan nahiz esleipenen eskuineko aldetan baliozkoak dira.

e) While-programei lotutako funtzioak

$\mathbb{W}$ datu-mota while-programen multzoak osatzen du, eta hauen definizioa A eranskinean ikus daiteke. Mota honetako datuekin lan egiten duten eragiketen artean honakoak bereiz ditzakegu:

- Eraiketan parte hartzen duten aldagaien datuetatik, ikurretatik edota azpiprogrametatik abiatuz, $\mathbb{W}$ motako objektuak eratzeko aukera ematen diguten funtzio eraikitzaileak.

$$\text{esleipen\_hutsa\_eratu} : \mathbb{N} \to \mathbb{W}$$

$$\text{esleipen\_cons\_eratu} : \mathbb{N} \times \sum^* \times \mathbb{N} \to \mathbb{W}$$

$$\text{esleipen\_cdr\_eratu} : \mathbb{N} \times \mathbb{N} \to \mathbb{W}$$

$$\text{konposaketa\_eratu} : \mathbb{W} \times \mathbb{W} \to \mathbb{W}$$

$$\text{baldintzazkoa\_eratu} : \sum^* \times \mathbb{N} \times \mathbb{W} \to \mathbb{W}$$

$$\text{iterazioa\_eratu} : \mathbb{N} \times \mathbb{W} \to \mathbb{W}$$

Adibidez, baldintzazkoa_eratu(**b**, 4, X0:=ε;) eragiketarekin honako programa sortuko da: **if** $\text{car}_{\mathbf{b}}$?(X4) **then** X0:=ε; **end if**;

- Ikuskatze-predikatuak, zeintzuk programetara aplikatzen diren eta emaitza boolearra itzultzen duten, emandako programa mota zehatz batekoa denentz adieraziz.

$$\text{esleipen\_hutsa?} : \mathbb{W} \to \mathbb{B}$$

$$\text{esleipen\_cons?} : \mathbb{W} \to \mathbb{B}$$

$$\text{esleipen\_cdr?} : \mathbb{W} \to \mathbb{B}$$

$$\text{konposaketa?} : \mathbb{W} \to \mathbb{B}$$

$$\text{baldintzazkoa?} : \mathbb{W} \to \mathbb{B}$$

$$\text{iterazioa?} : \mathbb{W} \to \mathbb{B}$$

- Atzipen-eragiketak, zeintzuek while-programa baten osagai gisa parte hartzen duten elementuen berri ematen diguten. *aldagai_indize* : $\mathbb{W} \to \mathbb{N}$ eta *aldagai_indize_2* : $\mathbb{W} \to \mathbb{N}$ funtzioek, programa baten eraiketan goreneko mailan erabilitako aldagaien indizea itzultzen dute. Era berean *ikur_erabili* : $\mathbb{W} \to \sum^*$ funtzioak ikurra ateratzea uzten digu eta *barne_agindu* : $\mathbb{W} \to \mathbb{W}$ eta *barne_agindu_2* : $\mathbb{W} \to \mathbb{W}$ funtzioek, berriz, barneko aginduak.

$$\text{aldagai\_indize}(\text{XI:=}\varepsilon;) = \mathbf{i}$$

$$\text{aldagai\_indize}(\text{XI:=cons}_{a_k}\text{(XJ);}) = \mathbf{i}$$

$$\text{aldagai\_indize}(\text{XI:=cdr(XJ);}) = \mathbf{i}$$

$$\text{aldagai\_indize}(\textbf{if } \text{car}_{a_k}\text{?(XI) } \textbf{then } \text{P } \textbf{end if};) = \mathbf{i}$$

$$\text{aldagai\_indize}(\textbf{while } \text{nonem?(XI) } \textbf{loop } \text{P } \textbf{end loop};) = \mathbf{i}$$

$$\text{aldagai\_indize}(P_1\ P_2) \cong \perp$$

Aipatzekoa da azken kasu honetan *aldagai_indize* indefinituta dagoela. Programak aldagairen bat eduki behar duen arren, hau ez da konposaketari dagokion goreneko mailan erabili.

$$\text{aldagai\_indize\_2}(\text{XI:=cons}_{a_k}\text{(XJ);}) = \mathbf{j}$$

$$\text{aldagai\_indize\_2}(\text{XI:=cdr(XJ);}) = \mathbf{j}$$

Gainontzeko kasuetan *aldagai_indize_2* funtzioa indefinituta geratzen da

$$\text{ikur\_erabili}(\text{XI :=cons}_{a_k}\text{(XJ);}) = \mathbf{a_k}$$

$$\text{ikur\_erabili}(\textbf{if } \text{car}_{a_k}\text{?(XI) } \textbf{then } \text{P } \textbf{end if};) = \mathbf{a_k}$$

Gainontzeko kasuetan *ikur_erabili* funtzioa indefinituta geratzen da

$$\text{barne\_agindu}(P_1\ P_2) = \mathbf{P_1}$$

$$\text{barne\_agindu}(\textbf{if } \text{car}_{\mathbf{s}}\text{?(XI) } \textbf{then } \text{P } \textbf{end if};) = \mathbf{P}$$

$$\text{barne\_agindu}(\textbf{while } \text{nonem?(XI) } \textbf{loop } \text{P } \textbf{end loop};) = \mathbf{P}$$

Gainontzeko kasuetan *barne_agindu* funtzioa indefinituta geratzen da

$$\text{barne\_agindu\_2}(P_1\ P_2) = \mathbf{P_2}$$

Gainontzeko kasuetan *barne_agindu_2* funtzioa indefinituta geratzen da

Erreferentziak

[Gar 84] M. GARDNER. ¡Ajá! Paradojas. 2ª edición. Ed. Labor S.A. 1984.

[Har 87] D. HAREL. Algorithmics. The spirit of Computing. Addison-Wesley, 1987

[IIS 98] J. IBAÑEZ; A. IRASTORZA; A. SANCHEZ. While Programak. Konputagarritasun Teoria oinarritzeko tresna. Barne txostena UPV/EHU/LSI/TR 3-98.

[IIS 00] J. IBAÑEZ; A. IRASTORZA; A. SANCHEZ. Konputaezintasun frogapen batzuk diagonalizazio teknika erabiliz. Barne txostena UPV/EHU/LSI/TR 11-2000.

[MAK 88] R. N. MOLL; M. A. ARBIB; A. J. KFOURY. A programming approach to computablity. Springer-Verlag, 1988

[SW 88] R. SOMMERHALDER; S. C. van WESTRHENEN The theory of computability. Programs, Machines, Effectiveness and Feasibility. Addison-Wesley 1.988

[Sta 81] G. STAHL. *El método diagonal en teoría de conjuntos.* Teorema. Vol 11, nº1, pp 27-35, 1981