Tècniques recursives de disseny i transformació d'algorismes

C. Roselló

Report LSI–89–28

Tècniques recursives de disseny i transformació d'algorismes

Celestí Rosselló

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
08028 Barcelona

Resum. Es presenten els principis del disseny recursiu d'algorismes junt amb els conceptes matemàtics en els que es basa: els preordres ben fonamentats i les demostracions per inducció noetheriana. Tot seguit s'exposen les tècniques de disseny i transformació basades en el concepte d'immersió. Per il·lustrar les diferents tècniques exposades a l'article es resolen diversos exemples.

Abstract. We show the principles of recursive design and its mathematical basis: the concepts of well founded set and noetherian induction. We also show recursive design and transformation techniques based on embedding. To illustrate each technique several examples are solved.

1. Introducció

La raó que m'ha mogut a escriure aquest report ha estat l'absència d'un treball de característiques similars. Així com existeixen els treballs de Dijkstra [Dij-76] i Gries [Gri-81] sobre disseny d'algorismes iteratius no hi ha treballs similars sobre disseny recursiu. El tema del disseny recursiu s'ha tractat des de diferents punts de vista p.e. [Sch-84], [BW-82], [CIP-85], ... però no n'hi ha cap que estudiï el disseny recursiu a partir d'una especificació pre-post.

En aquest treball es pretén donar a conèixer les diferents tècniques de disseny d'algorismes recursius. Algunes d'aquestes tècniques són ben conegudes i s'han fet servir en diferents camps i per a propòsits diferents; d'altres són aportacions personals al tema.

M'ha interessat veure totes aquestes idees des del mateix punt de vista, unificant la notació, la presentació, l'estil i el formalisme que es fa servir. Això m'ha permès el poder resaltar les diferents relacions que hi ha entre les tècniques. Al mateix temps, aquest treball servirà de base i referència sobre el tema.

L'estructura de l'article és la següent. En primer lloc es dedica una secció a definir els conceptes sobre els que es basa el disseny recursiu i les demostracions per inducció: les relacions de preordre i els conjunts ben fonamentats. La secció 3 es dedica a presentar els principis de disseny recursiu. Tot seguit es passa a exposar la tècnica d'immersió i com s'aplica al disseny recursiu. El problema de l'eficiència dels algorismes obtinguts es tracta a la secció 5. La immersió es fa servir altre cop per presentar mètodes de transformació que permeten obtenir algorismes eficients. Per acabar, es dedica una secció al tema de transformacions per a obtenir funcions recursives finals com a pas previ de la transformació en programes iteratius equivalents. Al llarg de l'article es resolen nombrosos exemples per il·lustrar les tècniques que es van exposant.

2. Conceptes bàsics

Es diu que una definició és recursiva si fa referència a si mateixa. Probablement, la definició del factorial o dels números combinatoris són els exemples més coneguts de definicions recursives.

Un cop se sap quina és la definició recursiva d'una funció, és trivial obtenir un algorisme que la calculi: només cal escriure la definició en una notació algorísmica enlloc de matemàtica. El problema és

obtenir la definició recursiva; en general, no hi ha mètodes per a fer-ho. És molt més fàcil especificar una funció: establir amb precisió quin és el seu domini i quins resultats esperem obtenir. Per exemple, és ben senzill especificar la funció arrel quadrada d'un nombre real, però no ho és tant donar-ne una definició recursiva.

Sigui $f: T_1 \to T_2$ una funció. Per a especificar-la definirem dos predicats que reben el nom de precondició i postcondició. La precondició $Q(x)$ serveix per a establir el domini de f: els valors $x \in T_1$ per als quals f està ben definida són aquells que fan que el predicat $Q(x)$ valgui *cert*. La postcondició $R(x, y)$ relaciona el resultat de la funció amb el seu argument. Un valor $y \in T_2$ és un resultat vàlid de $f(x)$ si el predicat $R(x, y)$ val *cert*. La funció f satisfà l'especificació $\langle Q(x), R(x, y)\rangle$ si i només si per a tots els valors $x \in T_1$ la següent proposició és una tautologia:

$$Q(x) \Rightarrow R(x, f(x)).$$

Per exemple, l'especificació de la funció arrel quadrada seria:

$$x \geq 0 \Rightarrow f(x)^2 = x.$$

Establim que la funció no està definida per a valors negatius i que el quadrat del resultat coincideix amb l'argument. Fixem-nos en que l'especificació defineix amb precisió una funció però no dóna cap mètode efectiu de calcular-la. Aquesta és precisament la tasca del disseny algorísmic en general i del disseny recursiu en particular: trobar un algorisme, o sigui, un mètode de calcular la funció.

Un cop s'hagi obtingut un algorisme ens interessarà comprovar que sempre dóna la resposta correcta. Per això caldrà comprovar si la definició recursiva a la que haurem arribat satisfà l'especificació. L'única manera de comprovar-ho es fent una demostració per inducció. La inducció no és res més que una manera recursiva de demostrar teoremes: es demostra el teorema pel valor 0 i, suposant vàlid el teorema per a $n-1$, es demostra que també és vàlid per a n. Així sabem que un determinat teorema és vàlid per a tots els naturals. El teorema que haurem de demostrar és: la funció satisfà l'especificació, o sigui, per a tots els valors del seu domini calcula el valor que esperavem. Més concretament: $\forall x: Q(x) \Rightarrow R(x, f(x))$. Com veurem al llarg de l'article, fer un disseny recursiu no és altra cosa que fer un raonament per inducció.

El principi d'inducció sobre els naturals només és suficient quan T_1 és el conjunt dels naturals. Altrament, cal utilitzar una generalització d'aquest principi que rep el nom d'inducció noetheriana. Abans, però, caldrà definir els conceptes de relació de preordre i de conjunt ben fonamentat.

Relacions de preordre

Una relació binària $\preceq$ definida sobre un conjunt A rep el nom de preordre si és reflexiva i transitiva:

- (reflexiva) $a \preceq a$ per a qualsevol $a \in A$.
- (transitiva) $(a \preceq b) \wedge (b \preceq c) \Rightarrow (a \preceq c)$ per a qualssevol $a, b, c \in A$.

Es diu que a precedeix b (o que b segueix a) si $a \preceq b$. Un conjunt A preordenat segons $\preceq$ s'escriu $(A, \preceq)$.

Com que en una relació de preordre no es requereix la propietat antisimètrica, pot haver elements diferents que es precedeixin mútuament. Aquests elements reben el nom d'elements similars. La seva definició és:

$$(a \equiv b) \iff (a \preceq b) \wedge (b \preceq a).$$

La relació $\equiv$ és d'equivalència. El preordre $(A, \preceq)$ indueix una relació d'ordre $\preceq_{/\equiv}$ sobre el conjunt quocient $A_{/\equiv}$:

$$\forall \bar{a}, \bar{b} \in A_{/\equiv}\ (\bar{a} \preceq_{/\equiv} \bar{b} \iff a \preceq b).$$

Tot preordre defineix un preordre estricte $\prec$ de la següent manera:

$$(a \prec b) \iff (a \preceq b) \wedge \neg(a \equiv b),$$

que és una relació irreflexiva i transitiva.

Un element m d'un conjunt preordenat $(A, \preceq)$ rep el nom d'element minimal si no hi ha cap altre element que el precedeixi estrictament: $\neg\exists a \in A\ (a \prec m)$. Un element m rep el nom de mínim si precedeix tots els altres elements del conjunt: $\forall a \in A\ (m \preceq a)$. D'acord amb aquestes definicions, tot element mínim és també minimal però no a l'inrevés.

Conjunts ben fonamentats

Un conjunt preordenat $(A, \preceq)$ és ben fonamentat si totes les successions estrictament descendents d'elements de A són finites. Entenem per successió estrictament descendent tota seqüència d'elements a_0, a_1, ... de A tal que

$$\forall i \geq 0 \; (a_{i+1} \prec a_i).$$

Una definició alternativa però equivalent és: un conjunt $(A, \preceq)$ és ben fonamentat si tots els subconjunts no buits tenen, si més no, un element minimal.

La demostració d'equivalència és la següent. Sigui $(A, \preceq)$ un conjunt en el que es puguin construir successions estrictament descendents infinites. El conjunt format per tots els elements d'una d'aquestes successions no té minimal. Efectivament, qualsevol element de la successió en té un altre que el precedeix estrictament i, per tant, no pot ser el minimal del conjunt. Igualment, suposem que algun subconjunt no buit de $(A, \preceq)$ no té minimal. Podem construir amb elements d'aquest subconjunt una successió a_0, a_1, ... estrictament descendent i infinita de la següent manera: a_0 és un element qualsevol del subconjunt (que no és buit). Com que el conjunt no té minimals, existeix un element que precedeix a_0 estrictament: aquest serà a_1. Repetint l'argument anterior tants cops com calgui escollim a_2, a_3, ... Per tant les dues definicions són equivalents.

Morfismes de preordres

El següent teorema ens permetrà de construir preordres ben fonamentats sobre qualsevol conjunt a partir d'algun preordre ben fonamentat que coneguem.

Sigui A_1 un conjunt qualsevol, $(A_2, \preceq_2)$ un preordre ben fonamentat i $f: A_1 \to A_2$ una funció. El conjunt A_1 junt amb la relació $\preceq_1$ definida:

$$(a \preceq_1 b) \iff (f(a) \preceq_2 f(b))$$

per a qualssevol $a, b \in A_1$ és ben fonamentat.

Notem que la definició anterior garanteix que:

$$(a \prec_1 b) \iff (f(a) \prec_2 f(b))$$

on $\prec_1$ i $\prec_2$ són els preordres estrictes corresponents a $\preceq_1$ i $\preceq_2$ respectivament.

Per demostrar el teorema, notem que A_1 no pot tenir successions estrictament descendents infinites perquè a cada successió a_0, a_1, ... de A_1 n'hi correspon una altra $f(a_0)$, $f(a_1)$, ... de A_2 que no es infinita ja que A_2 és ben fonamentat.

És simple ara definir relacions que convertiran un conjunt qualsevol en un conjunt ben fonamentat. Només cal saber algun conjunt ben fonamentat i aplicar el teorema precedent.

Un error fàcil de cometre quan s'aplica aquest teorema és no comprovar que la imatge de la funció f és un subconjunt de A_2. En particular això passa molt sovint quan A_2 són els naturals.

Alguns conjunts ben fonamentats

1. Qualsevol conjunt finit amb una relació de preordre.
2. Els números naturals $\mathbf{N}$ amb l'ordre habitual.
3. $\mathbf{N}^2$ amb $(a, b) \preceq (c, d) \iff (a < c) \vee ((a = c) \wedge (b \leq d))$ que rep el nom de preordre lexicogràfic.

El principi d'inducció sobre els naturals

Quan volem demostrar per inducció que tots els naturals posseeixen una propietat P qualsevol procedim de la següent manera:

a) (base) demostrem $P(0)$ i

b) (herència) suposant cert $P(n-1)$ demostrem $P(n)$.

No sempre n'hi ha prou amb establir la hipòtesi d'inducció sobre el natural precedent, de vegades s'ha de reforçar la hipòtesi i suposar que tots els naturals precedents satisfan la propietat. Això dóna lloc al principi general d'inducció sobre els naturals: si per a qualsevol $n \in \mathbf{N}$ podem demostrar que

$$(\forall i < n\ P(i)) \Rightarrow P(n)$$

llavors $\forall n\ P(n)$. Encara que no es requereix explícitament una demostració de $P(0)$, en general serà necessari perquè quan $n = 0$ l'antecedent de la implicació és trivialment cert i queda $cert \Rightarrow P(0)$ que és equivalent a demostrar $P(0)$.

El principi d'inducció noetheriana

Sigui $(A, \preceq)$ un conjunt ben fonamentat. Si per a qualsevol $a \in A$ podem demostrar que

$$(\forall b \prec a\ P(b)) \Rightarrow P(a)$$

llavors $\forall a\ P(a)$. Com es pot observar, és molt semblant al principi d'inducció dels naturals però amb $\prec$ enlloc de $<$.

La demostració d'aquest principi és fa per contradicció. Suposem que la hipòtesi és certa. Sigui S el conjunt d'elements de A que no satisfan P

$$S = \{\, x \in A \mid \neg P(x) \,\}$$

i suposem que no és buit: $S \neq \emptyset$. Ja que A és ben fonamentat podem escollir un element $m \in S$ que sigui minimal. Si apliquem la hipòtesi d'inducció a m veurem que l'antecedent és trivialment cert perquè m és minimal. Llavors, el conseqüent també ha de ser cert, o sigui, m satisfà la propietat P i, per tant, no pot pertanyer a S. La suposició de que S no era buit ha de ser forçosament errònia i això demostra el principi d'inducció.

3. Metodologia del disseny recursiu

Introduïm en aquesta secció la notació i el mètode a seguir per a efectuar dissenys recursius. Tot disseny recursiu requereix seguir una sèrie de passos que a continuació detallem:

1. Donar nom a la funció que es vol dissenyar (per poder-la referenciar) i establir clarament el seu perfil: definir els paràmetres, els resultats i els tipus als quals pertanyen.
2. Especificar la funció. Cal definir la precondició i la postcondició.
3. Dissenyar la funció. Separar el conjunt de casos possibles en dos: els simples o trivials, que admeten una solució directa; i els recursius, que requereixen l'activació recursiva de la funció. Definir quina és la solució per a cada cas d'acord amb l'especificació.
4. Demostrar la correcció del disseny efectuat. Normalment aquest pas és trivial perquè el disseny es fa de manera que satisfaci l'especificació. Sovint consisteix únicament en validar el raonament inductiu.

Una funció és recursiva simple si, en els casos recursius, només requereix una única activació. Si en requereix més s'anomena recursiva múltiple o composta.

L'algorisme genèric d'una funció recursiva simple és el següent:

$\{Q(x)\}$
func $f(x : T_1)$ **ret** $(y : T_2)$ **es**
  cas $b_s(x) \rightarrow$ **ret** $e(x)$
    [] $b_r(x) \rightarrow$ **ret** $c(f(s(x)), x)$
  fcas
ffunc
$\{R(x, y)\}$

Observem que la precondició només estableix restriccions sobre els paràmetres: defineix el domini de la funció. La postcondició relaciona els paràmetres amb els resultats i és, de fet, qui defineix la funció

realment. El cos de la funció està format per una estructura alternativa que diferencia entre els casos simples i els recursius. Els casos simples són aquells que satisfan l'expressió booleana b_s (s significa simple), la solució dels quals ve donada per la funció e. Els casos recursius vénen definits per l'expressió booleana b_r; la seva solució requereix activar recursivament la funció per a un valor diferent del paràmetre, que ve donat per la funció s (s de següent o successor). A partir del resultat d'aquesta activació es calcula el resultat de la funció per mitjà de la funció c (c de combinar). És clar que c només pot dependre del resultat de f i del paràmetre x. Si la funció c no és necessària, o sigui, si $f(x) = f(s(x))$ en el cas recursiu, la funció rep el nom de recursiva final.

La demostració de que una funció recursiva és correcta es fa per inducció noetheriana i té els següents passos:

1. Demostrar que f està definida per a tot valor del seu domini: $Q(x) \Rightarrow b_s(x) \vee b_r(x)$.
2. Establir la base de la inducció: $Q(x) \wedge b_s(x) \Rightarrow R(x, e(x))$
3. Establir l'herència:
   a. $Q(x) \wedge b_r(x) \Rightarrow Q(s(x))$, que la funció és activada dins el domini de definició.
   b. $Q(x) \wedge b_r(x) \wedge R(s(x), y) \Rightarrow R(x, c(y, x))$, suposant correcta la funció f per a $s(x)$, demostrar que també és correcta per a x.
4. Validar el raonament inductiu: $Q(x) \wedge b_r(x) \Rightarrow (s(x) \prec x)$, garantir que $s(x)$ precedeix x per algun preordre ben fonamentat definit sobre el domini de f.

És molt habitual que es defineixi un preordre sobre T_1 mitjançant una funció $p: T_1 \rightarrow \mathbf{N}$ i s'apliqui el teorema de la secció anterior. Per a poder-lo aplicar així, però, comprovar que la imatge de p sigui el conjunt ben fonamentat dels naturals, si més no, per als valors que satisfan la precondició. Concretant, cal demostrar que: $Q(x) \Rightarrow p(x) \in \mathbf{N}$. Llavors el punt 4 es pot reescriure: $Q(x) \wedge b_r(x) \Rightarrow (p(s(x)) < p(x))$.

Exemples de disseny recursiu

Ens proposem dissenyar una funció recursiva que calculi el producte de dos números naturals presuposant que només disposem de les operacions suma i desplaçament (productes i divisions per 2). La seva especificació és:

$\{cert\}$
func $mul(x, y : nat)$ **ret** $(z : nat)$
$\{z = x * y\}$

Al lector li pot semblar que hem fet trampes: hem definit la funció de multiplicació (mul) respecte de l'operació de multiplicació ($*$). En realitat, no hem fet res malament. Volem un algorisme per a multiplicar i sabem quines propietats té l'operació de multiplicar que hem representat amb el símbol $*$. En una especificació poden aparèixer operacions (o funcions) que no necessàriament hem de saber calcular: n'hi ha prou amb saber quines propietats tenen per a poder treballar-hi.

Exemple 1. Primera solució

Per a descobrir una relació de recurrència podem raonar així: el resultat de $mul(x, y)$ es pot obtenir sumant y cops x. Sumar y cops x és el mateix que sumar x al resultat de sumar $y - 1$ cops x. Però sumar $y - 1$ cops x és el resultat de $mul(x, y - 1)$. Dissortadament, no hem avançat gaire perquè no sabem calcular $mul(x, y - 1)$. Suposem, però, que hi ha alguna manera de fer-ho (de la qual ens ocuparem més endavant) i continuem. La relació $mul(x, y) = mul(x, y - 1) + x$ només és vàlida si $y \geq 1$ ja que, quan y val zero, $y - 1$ no és un natural.

L'elecció d'aquesta relació pot semblar arbitrària perquè igualment podriem haver triat $mul(x, y) = mul(x, y + 1) - x$ i suposar que hi haurà alguna manera de calcular $mul(x, y + 1)$. A més, aquesta relació val per a qualsevol valor de y (i no solament per a $y > 0$). Es pot argumentar que aquesta relació no val perquè fem servir una operació de la que no disposem: la resta de naturals; però hi ha una raó molt més important que veurem tot seguit.

La primera relació no val pel cas $y = 0$, però sabem que $mul(x, 0) = 0$. Això significa que ja sabem calcular $mul(x, 1)$. Segons la relació que teniem (que podem aplicar perquè y val 1), $mul(x, 1) = mul(x, 0) + x = 0 + x = x$. De la mateixa manera podem calcular $mul(x, 2)$, aplicant la relació un nombre

suficient de vegades fins que arribem al cas $y = 0$ pel qual sabem la resposta: $mul(x, 2) = mul(x, 1) + x = mul(x, 0) + x + x = 0 + x + x = 2 * x$. Aquest mateix raonament val també per a $mul(x, 3)$, $mul(x, 4)$, ... En definitiva, tenim una manera de calcular la funció per a qualsevol valor dels seus paràmetres.

En canvi, per a l'altra relació això no és possible. Suposar que sabem calcular la funció per a $y + 1$ ens obliga a suposar que hem de saber calcular-la per a $y + 2$ i per a $y + 3$ i així successivament. Mai arribarem a trobar un valor de y pel qual sapiguem la resposta sense fer novament referència a la funció.

Amb la primera relació hem tingut èxit perquè la suposició "això es pot calcular d'alguna manera" s'ha fet sobre uns valors "més petits", "més propers" al cas simple. Tècnicament, és diu que el conjunt dels valors dels paràmetres ($\mathbf{N}^2$ en aquest cas) és un conjunt ben fonamentat compatible amb la relació de recurrència. En el nostre exemple, el valor $\langle x, y-1 \rangle$ precedeix estrictament al valor $\langle x, y \rangle$ en la relació de preordre ben fonamentada definida sobre $\mathbf{N}^2$: $\langle x, y \rangle \preceq \langle x', y' \rangle \iff y \leq y'$. L'algorisme corresponent es pot veure tot seguit.

func $mul(x, y : nat)$ **ret** $(z : nat)$ **es**
  cas $y = 0 \rightarrow$ **ret** 0
  $\square$ $y > 0 \rightarrow$ **ret** $mul(x, y - 1) + x$
  fcas
ffunc

Pasem ara a comprovar que es correcte. La demostració pas per pas és la següent:

1. $cert \Rightarrow y = 0 \vee y > 0$
2. $cert \wedge y = 0 \Rightarrow 0 = x * y$

3a. $cert \wedge y > 0 \Rightarrow cert$

3b. $cert \wedge y > 0 \wedge (z = x * (y - 1)) \Rightarrow (z + x = x * y)$

4. $cert \wedge y > 0 \Rightarrow \langle x, y - 1 \rangle \prec \langle x, y \rangle$

S'ha fet una substitució literal de cada terme de la demostració genèrica perquè sigui fàcil la seva identificació. Es veu que tots els passos són prou trivials i no necessiten cap argumentació addicional.

Exemple 2. Segona solució al producte de dos naturals

Pel mateix problema anterior ens proposem de trobar una solució que també faci ús de les operacions de desplaçament: duplicar i dividir per dos.

Suposem que y és parell. Llavors, en lloc de sumar y cops x, podem sumar-ne només la meitat i duplicar el resultat, o també, sumar la meitat de cops el doble de x:

$$y \text{ és parell} \Rightarrow mul(x, y) = mul(x * 2, y/2)$$
$$y \text{ és parell} \Rightarrow mul(x, y) = mul(x, y/2) * 2$$

Ara bé quan y és senar això no va bé, entre altres coses perquè no està clar què significa dividir un número senar per dos (què en fem del residu de la divisió?). Notem, però, que quan y és senar $y - 1$ és parell i a partir de la primera relació obtenim:

$$y \text{ és senar} \Rightarrow mul(x, y) = mul(x * 2, (y - 1)/2) + x$$

Com que tot número natural és parell o senar estem temptats de concloure que les relacions anteriors són suficients per a obtenir un programa correcte; però no és així. Sabem que ha d'haver algun cas simple, altrament no estaria garantit l'acabament del càlcul. Com en la solució anterior prendrem $y = 0$ de cas simple. Per a qualsevol altre valor de y, segons sigui parell o senar s'aplicarà una relació o l'altra. Escrivim l'algorisme i passem a demostrar-lo.

func $mul2(x, y : nat)$ **ret** $(z : nat)$ **es**
  cas $y = 0 \rightarrow$ **ret** 0
  $\square$ $y > 0 \rightarrow$ **cas** $parell(y) \rightarrow$ **ret** $mul2(x * 2, y/2)$
    $\square$ $\neg parell(y) \rightarrow$ **ret** $mul2(x * 2, (y - 1)/2) + x$
    fcas
  fcas
ffunc

Els punts 1 i 2 de la demostració són idèntics als de l'exemple anterior. El punt 3a s'ha de fer dues vegades, una per a cada cas recursiu:

$$y > 0 \wedge parell(y) \Rightarrow \langle x * 2, y/2 \rangle \in \mathbf{N}^2$$
$$y > 0 \wedge \neg parell(y) \Rightarrow \langle x * 2, (y-1)/2 \rangle \in \mathbf{N}^2$$

El punt 3b s'ha de fer també dues vegades:

$$y > 0 \wedge parell(y) \wedge z = (y/2) * (x * 2) \Rightarrow z = x * y$$
$$y > 0 \wedge \neg parell(y) \wedge z = ((y-1)/2) * (x * 2) \Rightarrow z + x = x * y$$

Per a fer aquestes demostracions només cal fer simples manipulacions algebraiques.

Per a validar el raonament inductiu serveix el mateix preordre de l'exemple anterior. Aquest cop però la demostració és:

$$y > 0 \wedge parell(y) \Rightarrow y/2 < y$$
$$y > 0 \wedge \neg parell(y) \Rightarrow (y-1)/2 < y$$

Totes dues equacions són certes perquè la divisió per 2 redueix el valor de y sempre que y (o $y-1$) sigui parell i diferent de zero.

Exemple 3. Recursió múltiple

Volem resoldre el següent problema: donats dos números naturals, diferents de zero, n i s, $\mathcal{S}(n,s)$ és el número de maneres de sumar n amb s sumands tots diferents de zero. Per exemple, $\mathcal{S}(7,3) = 4$ ja que $7 = 1+1+5 = 1+2+4 = 1+3+3 = 2+2+3$. Es consideren iguals dues maneres que només es diferencien en l'ordre dels sumands. Dissenyar un algorisme per a la funció $\mathcal{S}$.

L'especificació de la funció serà la següent:

$\{n \geq 1 \wedge s \geq 1\}$
func $sumes(n, s : nat)$ **ret** $(r : nat)$
$\{r = \mathcal{S}(n,s)\}$

Començarem el disseny cercant casos simples. La manera de sumar n amb 1 sol sumand és única. Sumar un total de n amb n sumands només es pot aconseguir d'una única manera: amb n uns. D'altra banda, és impossible totalitzar n amb més de n sumands, o el que ve a ser el mateix, és impossible obtenir un valor inferior a s amb s sumands. Resumint:

$$\mathcal{S}(n,s) = \begin{cases} 1 & \text{si } s = 1, \\ 1 & \text{si } s = n, \\ 0 & \text{si } s > n. \end{cases}$$

Els casos que queden ($2 \leq s < n$) els tractarem recursivament.

Sigui $n = d_1 + \cdots + d_s$ una manera possible de sumar n amb s sumands. Si sumem 1 a cada sumand obtindrem una manera de sumar $n+s$ amb s sumands: $n+s = (d_1+1) + \cdots + (d_s+1)$. Aquesta manera té la particularitat de que cap sumand val 1. Igualment podem dir que, si tenim una manera de sumar $n-s$ amb s sumands, afegint 1 a cada sumand obtenim una manera de sumar n amb s sumands, tots diferents de 1. Això només té sentit si $n - s \geq 1$, o sigui, si és un cas recursiu. La relació és vàlida fins i tot quan $\mathcal{S}(n-s, s) = 0$, o sigui, quan $n - s < s$. Efectivament, amb s sumands diferents de 1 s'obté una suma que val $2s$ o més; si $n < 2s$ és impossible obtenir una suma de valor n.

Fins ara, sabem quantes sumes hi ha que no fan servir uns. Quantes n'hi ha que en facin servir? Igual que abans, si tenim una manera de sumar n amb s sumands i hi afegim un sumand 1, obtenim una manera de sumar $n+1$ amb $s+1$ sumands. Per tant, si $n-1 = d_1 + \cdots + d_{s-1}$ llavors $n = d_1 + \cdots + d_{s-1} + 1$. La particularitat d'aquesta descomposició és que té algun sumand 1 (si més no, l'últim). Això només tindrà sentit quan $n - 1 \geq 1$ i $s - 1 \geq 1$, però els casos recursius satisfan totes dues coses.

Si $2 \leq s < n$, qualsevol suma possible de n en s sumands es pot obtenir d'alguna de les dues formes anteriors. A més a més, cap suma es pot obtenir de les dues formes ja que l'una sempre produeix sumes sense uns i l'altra sumes amb uns. Per tant, el número de maneres de sumar n amb s sumands es pot obtenir sumant les possibles formes de fer-ho amb uns i sense uns. Resumint,

$$2 \leq s < n \Rightarrow \mathcal{S}(n,s) = \mathcal{S}(n-s,s) + \mathcal{S}(n-1,s-1)$$

L'algorisme que s'obté és el següent:

func *sumes*$(n, s : nat)$ **ret** $(r : nat)$ **es**
  cas $s = 1 \rightarrow$ **ret** 1
  $\square$ $s = n \rightarrow$ **et** 1
  $\square$ $s > n \rightarrow$ **ret** 0
  $\square$ $2 \leq s \wedge s < n \rightarrow$ **ret** *sumes*$(n-s,s) +$ *sumes*$(n-1,s-1)$
  fcas
ffunc

És la transcripció exacta de les propietats que acabem de deduir i que correspondrien als punts 2 i 3 de la demostració de correcció. Per estar segurs de que tot és correcte, cal demostrar que s'han tractat totes les situacions possibles (això és ben clar) i que existeix algun preordre compatible amb la relació de recurrència. Un preordre possible és l'induït per la funció $p(n,s) = n + s$. La imatge de la funció és el preordre ben fonamentat dels naturals, per tant podem aplicar el teorema de la secció anterior. El decreixement d'una activació a la següent es demostra tot seguit i cal fer-ho per a les dues activacions:

$$2 \leq s < n \Rightarrow p(n-1,s-1) < p(n,s) \Leftrightarrow n + s - 2 < n + s$$
$$2 \leq s < n \Rightarrow p(n-s,s) < p(n,s) \Leftrightarrow n < n + s$$

4. Disseny per immersió

Hi ha molts problemes per als quals no és fàcil trobar una relació de recurrència. Pot ser que no s'ens acudeixi o que, tal i com l'hem especificat, no admeti cap relació de recurrència. Això no vol dir de cap manera que aquest problema no pugui ser resolt recursivament; només que ens fa falta informació addicional per descobrir una relació de recurrència. La dificultat és que aquesta informació addicional no es troba a l'especificació. Per superar aquest entrebanc utilitzarem la tècnica de disseny per immersió.

La tècnica d'immersió es basa en dissenyar una funció diferent, més general que l'original, per a la qual sigui possible obtenir un disseny recursiu. Un cop obtingut l'algorisme d'aquesta nova funció només cal particularitzar-lo per a que calculi la funció original. Per una funció més general entenem una funció amb més paràmetres que, en determinades condicions, calcula el mateix que la funció original. Una funció així rep el nom de funció immersora o funció d'immersió. Concretant, la tècnica de disseny per immersió té quatre passos:

a) especificar la funció que es vol dissenyar;
b) generalitzar-la per obtenir la funció d'immersió i la seva especificació;
c) dissenyar la funció d'immersió;
d) particularitzar, si convé, l'algorisme obtingut.

Els passos a) i c) són els típics de qualsevol disseny recursiu. El pas b) és el que presenta més dificultats. A l'article [RBP-89] es proposa un mètode de derivació de l'especificació de la funció d'immersió en el cas de recursivitat simple. Aquí només farem un resum molt breu d'aquest mètode i tot seguit passarem a resoldre exemples.

Sigui $f(x)$ una funció i $g(x,w)$ una generalització de f amb les seves corresponents especificacions:

$\{Q(x)\}$	$\{Q'(x,w)\}$
func $f(x)$ **ret** (y)	**func** $g(x,w)$ **ret** (y)
$\{R(x,y)\}$	$\{R'(x,w,y)\}$

Per poder dissenyar g primer hem de trobar la seva especificació. Com que volem fer servir g per a calcular f hem d'aconseguir que g satisfaci la postcondició de f per alguna precondició adequada Q'. O sigui, g ha de calcular un valor $y = f(x)$ que satisfaci:

$$\forall x \; \exists w \; (Q'(x, w) \Rightarrow R(x, f(x)))$$

Per obtenir l'especificació de g debilitarem aquesta implicació. Hi ha dues maneres de fer-ho: debilitar la part dreta o reforçar la part esquerra.

En el primer cas (debilitar la part dreta) la postcondició R' de g s'obté al substituir una constant de R pel paràmetre w; s'obté, doncs, una postcondició més feble. La precondició Q' restringirà el domini de definició de g a aquelles combinacions dels paràmetres x i w per als quals la funció estigui ben definida. Cal evitar, en particular, aquells casos que fan que la postcondició valgui *fals*.

En el segon cas (reforçar la part esquerra) mantindrem la part dreta intacta, o sigui farem $R = R'$ i el paràmetre w no apareixerà a la postcondició de g. Per a que això tingui èxit, caldrà "moure" a Q' part de la informació que hi ha a R. Per saber quina informació cal moure es parteix de la demostració de correcció del cas simple: $Q'(x, w) \wedge b_s(x, w) \Rightarrow R(x, e(x))$. Cal expressar R en forma conjuntiva i assignar-ne una part a Q' i l'altra a b_s.

Es difícil donar una regla clara de quan s'ha de fer servir una debilitació o l'altra; de vegades el mateix problema admet les dues. En general escollir entre una o altra debilitació és una decisió molt lligada a l'experiència. Uns quants exemples ens ajudaran a aclarir-ho tot.

Exemples

Resoldrem ara el següent problema dos cops, un per a cada mètode de debilitament. Donat un número natural n, calcular la part entera de la seva arrel quadrada: $r = \lfloor \sqrt{n} \rfloor$.

Exemple 4. Solució per debilitament de la part dreta.

L'especificació és:

$\{cert\}$
func $rel(n : nat)$ **ret** $(r : nat)$
$\{r^2 \leq n < (r+1)^2\}$

on hem substituït la part entera per la seva definició i hem elevat al quadrat tots els membres de la desigualtat.

Substituïm a la postcondició la constant 1 per un nou paràmetre a i obtenim:

$$r^2 \leq n < (r+a)^2$$

La funció no està ben definida si $a < 1$ perquè llavors la postcondició és equivalent a *fals*. Restringirem, doncs, el domini d'aquest paràmetre a $a \geq 1$. L'especificació de la funció immersora serà:

$\{1 \leq a\}$
func $irel(n, a : nat)$ **ret** $(r : nat)$
$\{r^2 \leq n < (r+a)^2\}$

És evident que $irel(n, 1) = rel(n)$.

Quan la funció valgui zero la primera conjunció de la postcondició serà trivialment certa, i l'altra conjunció ens dirà en quins casos la funció val zero: $n < a^2$. Hem trobat, doncs, els casos simples. Prenem com a casos recursius tots els altres: $n \geq a^2$. Per aconseguir arribar a algun cas simple i, per tant, per poder definir un preordre ben fonamentat, caldrà buscar una recurrència sobre un subproblema amb una n menor o una a més gran.

Quan es dissenya per immersió és una bona idea cercar relacions de recurrència entre subproblemes que només es diferencien en el valor del paràmetre addicional. A la pràctica això no és una restricció excessiva perquè, sovint, el disseny per immersió s'intenta quan falla el disseny directe. A més, si hem afegit un nou paràmetre és amb la intenció de fer-lo servir, de que ens ajudi en el disseny.

Descartem, doncs, cercar recurrències sobre una n menor. Com que $a \geq 1$, una possible manera de fer-lo més gran és duplicar-lo. Per trobar una relació entre $irel(n, 2 * a)$ i $irel(n, a)$ podem raonar així: $irel(n, 2 * a)$ calcula un valor r' que satisfà

$$r'^2 \leq n < (r' + 2 * a)^2$$

que seria el valor de $irel(n, a)$ si $n < (r' + a)^2$. En cas contrari, podem escriure:

$$\begin{aligned} r'^2 \leq n < (r' + 2 * a)^2 \wedge n \geq (r' + a)^2 &\Rightarrow (r' + a)^2 \leq n < (r' + 2 * a)^2 \\ &\Leftrightarrow (r' + a)^2 \leq n < ((r' + a) + a)^2 \end{aligned}$$

o sigui que el valor de $irel(n, a)$ seria $r' + a$. Resumint:

$$irel(n, a) = \begin{cases} r' & \text{si } n < (r' + a)^2 \\ r' + a & \text{si } n \geq (r' + a)^2 \end{cases} \qquad \text{on } r' = irel(n, 2 * a)$$

L'algorisme que resulta és:

$\{1 \leq a\}$
func $irel(n, a : nat)$ **ret** $(r : nat)$ **es**
  cas $n < a^2 \rightarrow$ **ret** 0
    $\square$ $n \geq a^2 \rightarrow$ **sigui** $r' = irel(n, 2 * a)$ **en**
      cas $n < (r' + a)^2 \rightarrow$ **ret** r'
        $\square$ $n \geq (r' + a)^2 \rightarrow$ **ret** $r' + a$
      fcas
  fcas
ffunc
$\{r^2 \leq n \wedge n < (r + a)^2\}$

La demostració de correcció és bastant senzilla i es basa precisament en les propietats que acabem de deduir. Respecte del preordre ben fonamentat, la funció $p(n, a) = \lfloor n/a^2 \rfloor$ n'indueix un de compatible amb la recurrència. Comprovem-ho:

a) la imatge de la funció p és el conjunt del números naturals, que és ben fonamentat:

$$Q'(n, a) \Rightarrow p(n, a) \in \mathbf{N} \iff n \in \mathbf{N} \wedge 1 \leq a \Rightarrow \lfloor n/a^2 \rfloor \in \mathbf{N};$$

b) l'activació $irel(n, 2 * a)$ precedeix a la $irel(n, a)$ en el preordre induït per p:

$$Q'(n, a) \wedge b_r(n, a) \Rightarrow p(n, 2 * a) < p(n, a) \iff 1 \leq a \wedge n \geq a^2 \Rightarrow \lfloor n/(2 * a)^2 \rfloor < \lfloor n/a^2 \rfloor$$

Per a demostrar-ho farem el canvi $x = 4 * n/a^2$ que dóna: $x \geq 1/4 \Rightarrow \lfloor x \rfloor < \lfloor 4 * x \rfloor$. Per a $1/4 \leq x < 1$ és evidentment cert. Per a $x \geq 1$ n'hi ha prou amb notar que: $\lfloor x \rfloor < 4 * \lfloor x \rfloor \leq \lfloor 4 * x \rfloor$.

Exemple 5. Solució per reforçament de la part esquerra

Resoldrem ara el mateix problema fent una immersió diferent. La idea següent resulta molt adequada en general per a decidir quina immersió cal fer. Dedicarem un paràmetre addicional a "mantenir" el resultat de la funció, de manera que quan s'arribi a un cas simple aquest paràmetre tindrà el valor de la funció. Formalment això significa que (vegeu l'esquema genèric) $e(x, w) = w$. Intentem, doncs, una generalització del tipus:

$\{Q'(n, a)\}$
func $irel(n, a : nat)$ **ret** $(r : nat)$
$\{r^2 \leq n \wedge n < (r + 1)^2\}$

Per trobar Q' fem les consideracions següents. En el cas simple caldrà demostrar que $Q'(n,a) \wedge b_s(n,a) \Rightarrow R(n,e(n,a))$ però d'acord amb la idea anterior $e(n,a) = a$, i tenim

$$Q'(n,a) \wedge b_s(n,a) \Rightarrow a^2 \leq n < (a+1)^2.$$

Resoldre aquesta implicació és molt fàcil. Assignem una de les conjuncions de la part dreta a Q' i l'altra a b_s. Per exemple, triem $Q'(n,a) = a^2 \leq n$. Els casos simples seran $b_s(n,a) = n < (a+1)^2$.

Igual que en el cas anterior només cercarem relacions de recurrència en les que el paràmetre original de la funció (n en aquest cas) no canviï.

La manera més senzilla d'apropar-se al cas simple és cercar una relació de recurrència entre $irel(n,a)$ i $irel(n,a+1)$. Al no haver canviat la postcondició, qualsevol de les dues activacions anteriors la satisfà, suposant que satisfacin la precondició. L'activació $irel(n,a+1)$ satisfarà la precondició si $(a+1)^2 \leq n$. Com que aquesta condició és precisament la oposada de la dels casos simples, hem acabat el disseny. Vegeu l'algorisme tot seguit.

$\{a^2 \leq n\}$
func $irel(n, a : nat)$ **ret** $(r : nat)$ **es**
  cas $n < (a+1)^2 \rightarrow$ **ret** a
  $\square$ $n \geq (a+1)^2 \rightarrow$ **ret** $irel(n, a+1)$
  fcas
ffunc
$\{r^2 \leq n \wedge n < (r+1)^2\}$

La demostració de correcció no és necessària ja que hem dissenyat l'algorisme basant-nos en el que necessitem complir. Només manca validar el raonament inductiu. Un preordre ben fonamentat compatible amb la recurrència és l'induït per la funció $p(n,a) = n - a^2$. Q' garanteix que la imatge de la funció és sempre un natural (podem aplicar el teorema) i les equacions següents

$$\begin{aligned} & Q'(n,a) \wedge b_r(n,a) \Rightarrow p(n,a+1) < p(n,a) \\ \Leftrightarrow\ & a^2 \leq n \wedge (a+1)^2 \leq n \Rightarrow n - (a+1)^2 < n - a^2 \end{aligned}$$

demostren que l'activació $irel(n,a+1)$ precedeix en el preordre a l'activació $irel(n,a)$.

Ara la relació entre rel i $irel$ no és única com a la solució anterior. En particular, $irel(n,0) = rel(n)$. Però molts altres valors a més a més del 0 serveixen per a inicialitzar el paràmetre a de la funció $irel$: es podria calcular d'alguna manera una aproximació a l'arrel de n i fer-la servir de valor inicial de a sempre que satisfaci la precondició. Per exemple, la potència de dos més gran que no superi l'arrel de n.

Sempre que se segueixi aquest mètode s'obtindrà una funció recursiva final perquè la postcondició de la funció immersora no depén dels nous paràmetres, i els paràmetres que ja hi havia no són alterats d'una activació a la següent. Efectivament, el punt 3b de la demostració de correcció adaptat a la funció d'immersió és (recordem que la postcondició no depén dels nous paràmetres): $Q'(x,w) \wedge b_r(x,w) \wedge R(s(x),y) \Rightarrow R(x,c(y,x))$. Si els paràmetres de la funció original no varien tenim $s(x) = x$ i llavors podem escriure: $Q'(x,w) \wedge b_r(x,w) \wedge R(x,y) \Rightarrow R(x,c(y,x))$. La manera més simple de satisfer-la és prendre $c(y,x) = y$ i això fa que la funció sigui recursiva final.

5. Transformacions d'eficiència

La eficiència és un aspecte important en el disseny d'algorismes. D'entre dos algorismes correctes que resolen el mateix problema preferirem aquell que ho faci més ràpid. Sovint, quan se segueix un mètode formal de disseny, la solució a la que s'arriba no és la més eficient. Si per a obtenir un algorisme més eficient ens l'inventem pot deixar de ser correcte. La situació és clara: no estem disposats a tenir algorismes correctes si no són eficients. Però tampoc volem obtenir algorismes eficients i incorrectes. Veurem tot seguit que la tècnica d'immersió proporciona una manera d'aconseguir algorismes a la vegada eficients i correctes.

Entre altres raons, els algorismes són ineficients perquè calculen diverses vegades el mateix, o bé perquè fan servir operacions de cost elevat quan amb altres de cost menor n'hi hauria prou. En aquestes situacions, és útil mantenir la informació necessària per a evitar la repetició de càlculs. Aquesta idea és ben coneguda en programació funcional (vegeu [DM-87], [BMPP-89] o [FH-88]) i també en optimització de programes (vegeu [PK-82]). Un resum molt complet (però també molt breu) de les tècniques de transformació es pot trobar a [Fea-87].

En el disseny recursiu la informació necessària es manté en nous paràmetres que s'afegeixen a la funció. Es tracta, doncs, de fer la immersió adequada. De fet, no sempre és possible mantenir la informació necessària en els paràmetres i resulta útil ampliar la idea d'immersió a funcions amb més d'un resultat. Se li dóna el nom d'immersió de resultats per diferenciar-la de l'altra immersió, la de paràmetres.

Sigui f una funció recursiva com la de l'esquema genèric, el disseny de la qual ja s'ha fet i és correcte. Representem per $\phi(x)$ l'expressió que es vol evitar de calcular repetidament. Fem la següent immersió:

$\{Q(x) \wedge w = \phi(x)\}$
func $g(x, w)$ **ret** (y)
$\{R(x, y)\}$

El disseny recursiu de g serà el mateix que el de f però ara podem substituir totes les aparicions de $o(x)$ en g per w. Això manté la correcció de g perquè la seva precondició estableix la igualtat de w i $o(x)$. L'única diferència substancial de la demostració de correcció de g respecte de la de f és la següent. Per satisfer la precondició, l'activació recursiva de g haurà de ser $g(s(x), \phi(s(x)))$. L'operació o no ha desaparegut completament però intentarem expressar $\phi(s(x))$ d'una manera més simple. Sovint, és fàcil expressar $\phi(s(x))$ en funció de $\phi(x)$ i després substituir $\phi(x)$ per w. L'eficiència millorarà si calcular $\phi(s(x))$ en funció de $\phi(x)$ costa menys que el càlcul directe de $\phi(s(x))$.

Totes les expressions sospitoses de ser ineficients i que no depenen del valor que retorna l'activació recursiva es poden tractar així. En particular, i referint-nos a l'esquema genèric, les que formen part de b_s, b_r, e i s. En canvi, les que depenen del valor retornat (que necessàriament formaran part de la funció c de l'esquema genèric) no s'hi poden tractar. No serviria de res afegir un nou paràmetre perquè ens obligaria a mantenir calculat el valor que estem calculant: estariem fent dependre la precondició de la funció del seu resultat. Per poder aplicar la mateixa tècnica hem de fer servir una immersió de resultats. Un dels resultats coincidirà amb el de la funció inicial i els altres serviran per a mantenir informació.

Igual que abans, suposem que ja tenim dissenyada una funció f i que fem la següent immersió:

$\{Q(x)\}$
func $g(x)$ **ret** (y, z)
$\{R(x, y) \wedge z = \psi(x, y)\}$

Com en el cas d'immersió de paràmetres, el disseny de g és una còpia del de f. Després de l'activació recursiva de g s'obtenen un valors y' i z' que satisfan $R(s(x), y') \wedge z' = \psi(s(x), y')$. Les expressions c que formin part dels càlculs posteriors a l'activació recursiva es podran substituir per z'. Igual que abans la correcció és manté i només cal assegurar que la funció g calculi el valor adequat per z que, sovint, es pot calcular a partir de z'. Com el cas d'immersió de paràmetres, l'eficiència millorarà si el càlcul de z a partir de z' costa menys que calcular z directament.

Exemple 6. Eficiència per immersió de paràmetres

Prenem la segona solució al problema de l'arrel quadrada que hem obtingut en l'exemple 4. Notem que l'expressió $(a + 1)^2$ s'avalua un cop per activació recursiva.

Introduïm un paràmetre b per a substituir $(a + 1)^2$. L'especificació de la immersió serà

$\{a^2 \leq n \wedge b = (a + 1)^2\}$
func $iirel(n, a, b : nat)$ **ret** $(r : nat)$
$\{r^2 \leq n \wedge n < (r + 1)^2\}$

on s'ha afegit a la precondició la relació entre l'expressió i el nou paràmetre.

Per garantir la correcció hem de comprovar que, en el casos recursius, la funció s'activa amb uns valors que satisfan la precondició. Els valors que cal donar als paràmetres a i n són els mateixos que ja tenien a la funció original: pel que fa a ells l'especificació no ha canviat. Al nou paràmetre caldrà passar-li un valor b' que satisfaci la precondició. L'activació serà $iirel(a, n+1, b')$ i la precondició exigeix que

$$b' = ((a+1)+1)^2 = (a+1)^2 + 2*(a+1) + 1 = b + 2*a + 3.$$

Recollint tota aquesta informació s'arriba a l'algorisme següent.

func $iirel(n, a, b : nat)$ **ret** $(r : nat)$
  cas $n < b \longrightarrow a$
  $\square$ $n \geq b \longrightarrow iirel(n, a+1, b+2*a+3)$
  fcas
ffunc

El guany en eficiència que s'ha obtingut és molt petit. Al substituir un producte per una suma només guanyem en un factor constant. Sembla, doncs, que no val la pena fer aquest esforç si hem de guanyar tant poc. Veurem, però, que hi ha situacions en les que la immersió permet reduir el cost assimptòtic de l'algorisme.

Exemple 7. Eficiència per immersió de resultats

Vegem com aplicariem aquesta tècnica a la primera solució del problema de l'arrel quadrada (exemple 4). Hi ha dues expressions que requereixen l'ús de multiplicacions: a^2 i $(r'+a)^2$. Per a la primera farem una immersió de paràmetres; per a la segona, una immersió de resultats.

La immersió de paràmetres porta a la següent especificació:

$\{1 \leq a \wedge b = a^2\}$
func $iirel(n, a, b : nat)$ **ret** $(r : nat)$
$\{r^2 \leq n < (r+a)^2\}$

Substituïm les expressions i obtenim:

func $iirel(n, a, b : nat)$ **ret** $(r : nat)$ **es**
  cas $n < b \longrightarrow$ **ret** 0
  $\square$ $n \geq b \longrightarrow$ **sigui** $r' = iirel(n, 2*a, 4*b)$ **en**
    cas $n < r'^2 + 2*r'*a + b \longrightarrow$ **ret** r'
    $\square$ $n \geq r'^2 + 2*r'*a + b \longrightarrow$ **ret** $r' + a$
    fcas
  fcas
ffunc

L'expressió $(r+a)^2$ s'ha desenvolupat i el terme a^2 s'ha substituït igualment per b.

Afegim els resultats $s = r^2$ i $t = r*a$ per evitar el càlcul dels productes que surten a l'expressió $r'^2 + 2*r'*a + b$ de l'algorisme anterior. L'especificació serà:

$\{1 \leq a \wedge b = a^2\}$
func $iirel(n, a, b : nat)$ **ret** $(r, s, t : nat)$
$\{r^2 \leq n < (r+a)^2 \wedge s = r^2 \wedge t = r*a\}$

Si fem $\langle r', s', t' \rangle = iirel(n, 2*a, 4*b)$ tindrem:

$$r'^2 \leq n < (r' + 2*a)^2 \wedge s' = r'^2 \wedge t' = 2*r'*a$$

i podrem substituir les expressions r'^2 i $2*r'*a$ posteriors a l'activació per s' i t' respectivament. Per completar el disseny calculem els nous valors de s i t en funció de s' i t'. En el primer cas tenim $r = r'$ i, per tant

$$s = r^2 = r'^2 = s'$$
$$t = r*a = r'*a = t'/2.$$

Per a l'altre cas tenim $r = r' + a$ i llavors:

$$s = r^2 = (r' + a)^2 = s' + t' + b$$
$$t = r * a = (r' + a) * a = r' * a + a^2 = t'/2 + b.$$

Només resta decidir quins valors de s i t satisfan la postcondició pels casos simples. Com que $r = 0$, s i t valdran també 0. A continuació veiem el disseny complet de la funció on ja no surten sinó sumes i desplaçaments (productes i divisions per 2).

func *iiirel*$(n, a, b : nat)$ **ret** $(r, s, t : nat)$ **es**
  cas $n < b \rightarrow$ **ret** $\langle 0, 0, 0 \rangle$
  $\square$ $n \geq b \rightarrow$ **sigui** $\langle r', s', t' \rangle = iiirel(n, 2 * a, 4 * b)$ **en**
    $\{r'^2 \leq n < (r' + 2 * a)^2 \wedge s' = r'^2 \wedge t' = 2 * r' * a\}$
    cas $n < s' + t' + b \rightarrow$ **ret** $\langle r', s', t'/2 \rangle$
    $\square$ $n \geq s' + t' + b \rightarrow$ **ret** $\langle r' + a, s' + t' + b, t'/2 + b \rangle$
    fcas
  fcas
ffunc

Per saber la relació que hi ha entre aquestes funcions i les originals notem que els nous paràmetres i resultats estan sempre especificats respecte dels vells. Per tant, tindrem:

$$\begin{aligned} rel(n) &= irel(n, 1) \\ &= iirel(n, 1, 1)) \\ &= \pi_1(iiirel(n, 1, 1)) \end{aligned}$$

on π_1 és la funció de projecció de la primera component de *iiirel*. Notem que, el nostre propòsit és calcular $rel(n)$ i que totes les immersions han servit per fer el disseny i per obtenir eficiència, però no ens interessen per elles mateixes sinó per la seva relació amb *rel*. L'arrel quadrada de n sempre es calcularà fent l'activació $iiirel(n, 1, 1)$. D'aquí que sigui possible fer una simplicació d'aquest l'algorisme. Notem que el resultat t coincideix amb r quan $a = 1$: $\pi_1(iiirel(n, 1, 1)) = \pi_3(iiirel(n, 1, 1))$ (π_3 és la projecció de la tercera component). Per tant el resultat r no és necessari.

Com que cap expressió de la funció depén de r excepte la que calcula la pròpia r, podem suprimir-la. El paràmetre a només s'utilitzava en el càlcul de r i, per tant, es pot suprimir igualment. Tot seguit podem veure el resultat d'aquestes simplificacions. La postcondició s'ha reformulat tenint en compte que $t = r * a \Leftrightarrow t^2 = r^2 * a^2 = r^2 * b$. En definitiva tenim que $rel(n) = \pi_2(ivrel(n, 1))$.

$\{1 \leq b\}$
func *ivrel*$(n, b : nat)$ **ret** $(s, t : nat)$ **es**
  cas $n < b \rightarrow$ **ret** $\langle 0, 0 \rangle$
  $\square$ $n \geq b \rightarrow$ **sigui** $\langle s', t' \rangle = ivrel(n, 4 * b)$ **en**
    cas $n < s' + t' + b \rightarrow$ **ret** $\langle s', t'/2 \rangle$
    $\square$ $n \geq s' + t' + b \rightarrow$ **ret** $\langle s' + t' + b, t'/2 + b \rangle$
    fcas
  fcas
ffunc
$\{t^2 \leq b * n < (t + b)^2 \wedge t^2 = b * s\}$

Exemple 8. Il·lustració de tots els conceptes anteriors

Considerem el següent problema. Donada una constant natural n i un vector $a[1{:}n]$ de naturals, decidir si hi ha alguna manera de descomposar el vector en una part dreta i una part esquerra que sumin igual. Evidentment, aquestes parts poden ser de mides diferents.

L'especificació és:

$\{n \geq 0\}$
func *particio*$(a : vector; n : nat)$ **ret** $(b : bool)$
$\{b = (\exists i{:}0 \leq i \leq n{:}(\Sigma j{:}1 \leq j \leq i{:}a[j] = \Sigma j{:}i < j \leq n{:}a[j]))\}$

Dissenyarem l'algorisme per immersió en una funció de postcondició més feble. Substituirem la constant 0 per un paràmetre k. La nova funció té la següent especificació:

$\{n \geq 0 \wedge 0 \leq k \leq n+1\}$
func *iparticio*$(a : vector; n : nat; k : nat)$ **ret** $(b : bool)$
$\{b = (\exists i{:}k \leq i \leq n{:}(\Sigma j{:}1 \leq j \leq i{:}a[j] = \Sigma j{:}i < j \leq n{:}a[j]))\}$

El domini del paràmetre k va des de 0 (el valor substituït) fins a $n+1$ (el valor per al qual l'interval sobre el quantificador $\exists$ és nul). Per a valors de k superiors a $n+1$ el quantificador no està definit. Per a valors de k inferiors a zero la funció *iparticio* no està ben definida: els sumatoris no estarien ben definits o es referenciarien valors inexistents del vector a. La relació amb la funció inicial és:

$$particio(a, n) = iparticio(a, n, 0).$$

Per dissenyar la funció farem l'anàlisi de casos sobre el número de valors quantificats per $\exists$. Quan el número de valors sigui zero, és a dir, quan l'interval del quantificador sigui nul, la funció valdrà *fals*. Això és dedueix a partir de la definició del quantificador $\exists$ i correspon al cas en el que $k = n+1$. Els casos recursius seran tots els altres: aquells que tenen un o més valors quantificats i que corresponen als casos en els que $k < n+1$. Per tendir cap a un cas simple caldrà reduir el número de valors de l'interval de quantificació. Així podem establir la següent relació (on hem abreujat $(\Sigma j{:}1 \leq j \leq i{:}a[j] = \Sigma j{:}i \leq j \leq n{:}a[j])$ per $P(i)$):

$$\exists i{:}k \leq i \leq n{:}P(i) \Leftrightarrow (P(k) \vee \exists i{:}k+1 \leq i \leq n{:}P(i)).$$

O sigui, si hi ha una possible descomposició entre k i n, o bé la descomposició és a k o bé està entre $k+1$ i n. Resumint:

$$k = n+1 \Rightarrow iparticio(a, n, k) = fals$$
$$k < n+1 \Rightarrow iparticio(a, n, k) = P(k) \vee iparticio(a, n, k+1)$$

La funció *iparticio* queda:

func *iparticio*$(a : vector; n : nat; k : nat)$ **ret** $(b : bool)$ **es**
  cas $k = n+1 \rightarrow$ **ret** *fals*
  $\square$ $k < n+1 \rightarrow$ **ret** $(\Sigma j{:}1 \leq j \leq k{:}a[j] = \Sigma j{:}k+1 \leq j \leq n{:}a[j]) \vee iparticio(a, n, k+1)$
  fcas
ffunc

La demostració detallada de la correcció de l'algorisme es deixa al lector. Respecte del preordre, la funció $p(a, n, k) = n+1-k$ n'indueix un de compatible amb la recurrència.

La funció està inacabada perquè no s'ha donat cap manera de calcular els dos sumatoris que apareixen en el cas recursiu. Qualsevol implementació raonable trigaria un temps $\Theta(n)$ en calcular-los. El cost de la funció *iparticio* vindria expressat per la següent recurrència:

$$cost(a, n, k) = \begin{cases} c & \text{si } k = n+1, \\ cost(a, n, k+1) + c'n + c'' & \text{si } k < n+1; \end{cases}$$

que dóna: $cost(a, n, k) \in \Theta(n(n-k))$. El cost de la funció original, $particio(a, n)$, seria $\Theta(n^2)$.

Per obtenir un algorisme més eficient i igualment correcte farem una immersió que optimitzarà el càlcul dels dos sumatoris. Hi ha dues possibilitats: 1) es poden afegir dos paràmetres, un per a cada sumatori; 2) es pot afegir un paràmetre per a un dels sumatoris i un resultat per a l'altre.

En el primer cas definim dos paràmetres addicionals *se* i *sd* per a mantenir el valor dels dos sumatoris. L'especificació resultant és:

$\{n \geq 0 \wedge 0 \leq k \leq n+1 \wedge se = \Sigma j{:}1 \leq j \leq k{:}a[j] \wedge sd = \Sigma j{:}k+1 \leq j \leq n{:}a[j]\}$
func *iiparticio*$(a : vector; n, k, se, sd : nat)$ **ret** $(b : bool)$
$\{b = (\exists i{:}k \leq i \leq n{:}(\Sigma j{:}1 \leq j \leq i{:}a[j] = \Sigma j{:}i < j \leq n{:}a[j]))\}$

Ara cal substituir els sumatoris per els nous paràmetres i calcular els valors que han de rebre aquests paràmetres en les activacions recursives. Anomenem se' i sd' aquests valors. Tindrem:

$$se' = \Sigma j{:}1 \leq j \leq k+1{:}a[j] = se + a[k+1]$$
$$sd' = \Sigma j{:}k+2 \leq j \leq n{:}a[j] = sd - a[k+1]$$

i la funció serà:

func *iiparticio*$(a : vector; n, k, se, sd : nat)$ **ret** $(b : bool)$ **es**
  cas $k = n+1 \rightarrow$ **ret** *fals*
    $\square$ $k < n+1 \rightarrow$ **ret** $(se = sd) \vee$ *iiparticio*$(a, n, k+1, se + a[k+1], sd - a[k+1])$
  fcas
ffunc

Per calcular la relació d'aquesta funció amb la original recordem que k havia de valer 0. D'acord amb la precondició de *iiparticio* tenim:

$$se = \Sigma j{:}1 \leq j \leq k{:}a[j] = 0$$
$$sd = \Sigma j{:}k+1 \leq j \leq n{:}a[j] = \Sigma j{:}1 \leq j \leq n{:}a[j]$$

La relació entre aquesta última immersió i la funció original és:

$$particio(a, n) = iiparticio(a, n, 0, 0, \Sigma j{:}1 \leq j \leq n{:}a[j])$$

i caldrà calcular la suma de tot el vector abans d'activar la funció.

Noteu que el cost de la funció en el cas recursiu s'ha reduït; la recurrència és ara:

$$cost(a, n, k) = \begin{cases} c & \text{si } k = n+1, \\ cost(a, n, k+1) + c' & \text{si } k < n+1; \end{cases}$$

que dóna: $cost(a, n, k) \in \Theta(n-k)$; com que inicialment $k = 0$ queda un cost $\Theta(n)$. En aquest cost caldrà afegir-li el de la inicialització del paràmetre *sd*. Per a sumar tot el vector fa falta un cost linial, per tant, el cost de la funció $particio(a, n)$ serà $\Theta(n)$.

Amb aquest exemple hem vist com una immersió pot aconseguir mantenir la correcció i rebaixar el cost assimptòtic.

L'altra possibilitat que haviem anunciat era mantenir un sumatori en un paràmetre i l'altre en un resultat. Mantindrem en un nou paràmetre *se* el mateix sumatori que en la solució precedent i intentarem mantenir l'altre sumatori en un resultat. Això ens porta a l'especificació i l'algorisme següents:

$\{n \geq 0 \wedge 0 \leq k \leq n+1 \wedge se = \Sigma j{:}1 \leq j \leq k{:}a[j]\}$
func *iiiparticio*$(a : vector; n, k, se : nat)$ **ret** $(b : bool; sd : nat)$ **es**
  cas $k = n+1 \rightarrow$ **ret** $\langle fals, ? \rangle$
    $\square$ $k < n+1 \rightarrow$ **sigui** $\langle b', sd' \rangle =$ *iiiparticio*$(a, n, k+1, se + a[k+1])$ **en**
      ret $\langle (se = \Sigma j{:}k+1 \leq j \leq n{:}a[j]) \vee b', ? \rangle$
  fcas
ffunc

Hem posat signes d'interrogació als llocs on aniran els valors del nou resultat, que encara no sabem perquè no hem decidit quina serà la postcondició. Respecte del resultat b la postcondició serà, òbviament,

la mateixa. Respecte de sd seria bo que sd' fos el sumatori que encara no hem substituït. Això simplificaria, si més no, aquest càlcul. D'acord amb el cas recursiu tenim que sd haurà de ser una expressió tal que si substituïm k per $k+1$ obtinguem sd' o sigui que:

$$sd = \Sigma j{:}k \leq j \leq n{:}a[j]$$

i això és el que posarem a la postcondició junt amb que ja hi havia anteriorment sobre b. Per a completar el disseny només cal veure quins valors cal posar en el lloc dels interrogants. En el cas simple tenim $k = n+1$ i per tant:

$$sd = \Sigma j{:}n+1 \leq j \leq n{:}a[j] = 0$$

i en el cas recursiu, $k < n+1 \Leftrightarrow k \leq n$ i tenim:

$$sd = \Sigma j{:}k \leq j \leq n{:}a[j] = a[k] + \Sigma j{:}k+1 \leq j \leq n{:}a[j] = a[k] + sd'$$

Hem completat el disseny de la funció que podem veure tot seguit:

$\{n \geq 0 \wedge 0 \leq k \leq n+1 \wedge se = \Sigma j{:}1 \leq j \leq k{:}a[j]\}$
func *iiiparticio*$(a : vector; n, k, se : nat)$ **ret** $(b : bool; sd : nat)$ **es**
  cas $k = n+1 \rightarrow$ **ret** $\langle fals, 0\rangle$
  $\square$ $k < n+1 \rightarrow$ **sigui** $\langle b', sd'\rangle =$ *iiiparticio*$(a, n, k+1, se + a[k+1])$ **en**
    ret $\langle (se = sd') \vee b', sd' + a[k]\rangle$
  fcas
ffunc
$\{b = (\exists i{:}0 \leq i \leq n{:}(\Sigma j{:}1 \leq j \leq i{:}a[j] = \Sigma j{:}i < j \leq n{:}a[j])) \wedge sd = \Sigma j{:}k \leq j \leq n{:}a[j]\}$

Notem que aquest algorisme té també un cost linial. A més a més, la inicialització és molt més simple ja que no cal sumar tot el vector: $particio(a, n) = \pi_1(iiiparticio(a, n.0, 0))$.

6. Recursió final

L'interés de tenir dissenys recursius finals ve del fet que és molt simple transformar-los en algorismes iteratius equivalents (o sigui, que satisfan la mateixa especificació). Tota funció recursiva final té un algorisme iteratiu equivalent que té un sol bucle.

Aquests algorismes iteratius són, en general, més eficients que els recursius; si bé només en un factor constant. S'ha de tenir en compte que les instruccions de control de bucle són més ràpides que les d'activació recursiva i de retorn d'una activació.

A més a més, no hi ha ordinadors "recursius" i els compiladors dels llenguatges que tenen recursivitat generen un codi màquina que simula la recursió amb l'ajut d'una pila. En aquesta pila s'hi guarden els paràmetres i els valors de les variables locals de cada una de les activacions recursives que s'han iniciat i s'han deixat pendents per fer una altra activació. Com més activacions hi ha més gran és la pila que es necessita. En els algorismes iteratius no es necessita pila, només una variable per a cada paràmetre de la funció independentment de quantes activacions es facin. L'estalvi de memòria pot arribar a ser molt gran. De fet tot el problema ve de que els compiladors simulen la recursió enlloc de traduir automàticament els algorismes recursius en els seus equivalents iteratius.

Quan el disseny que hem obtingut no és recursiu final caldrà fer algun tipus de transformació que, mantenint la correcció, ens doni una funció final equivalent. Aquesta transformació es farà amb l'ajut d'una immersió. La resta d'aquesta secció està dedicada a explicar quina és la immersió adequada en cada cas.

De la recursió simple a la final

Per aconseguir convertir una funció recursiva simple en recursiva final ens hem de desempallegar dels càlculs posteriors a l'activació recursiva (representat per la funció c a l'esquema genèric) i efectuar-los abans de les activacions. Això no és sempre possible però quan ho sigui caldran un o més paràmetres addicionals per fer aquests càlculs. La manera d'aconseguir la definició d'aquests paràmetres es força mecànica i es troba a [AK–82]. La proposta és, esencialment, la següent.

Sigui f una funció recursiva no final. Substitueixi's a l'expressió del cas recursiu de f tots els termes que no continguin ni f ni c per variables. L'expressió resultant és la definició de la funció immersora que cal fer servir.

És clar que diferents funcions c donaran lloc a diferents generalitzacions. Nogensmenys, hi ha un cas particular que mereix una atenció especial donada la freqüència amb la que sol presentar-se: quan l'expressió del cas recursiu és $f(s(x)) \circ d(x)$, o sigui, quan c és una operació binària entre el resultat de l'activació recursiva i alguna funció de l'argument.

En aquest cas l'expressió generalitzada és: $f(v) \circ w$ que és precisament la definició de la funció immersora (diguem-ne g): $g(v, w) = f(v) \circ w$. D'acord amb aquesta definició i amb l'anàlisi de casos de la funció tenim que:

$$\begin{aligned} b_s(v) \Rightarrow g(v,w) = f(v) \circ w &= e(v) \circ w \\ b_r(v) \Rightarrow g(v,w) = f(v) \circ w &= (f(s(v)) \circ d(v)) \circ w \\ &= f(s(v)) \circ (d(v) \circ w) \\ &= g(s(v), d(v) \circ w) \end{aligned}$$

L'última igualtat requereix que $\circ$ sigui una operació associativa. Fixem-nos en que la funció g és recursiva final.

En realitat no s'ha fet altra cosa que una immersió. El paràmetre w ha permès d'evitar els càlculs posteriors a l'activació recursiva sempre i quan $\circ$ sigui una operació associativa. Per saber quina és la precondició de g seguirem la tècnica d'immersió que ja coneixem. L'equació que hem de plantejar és:

$$\begin{aligned} & Q'(x,v,w) \Rightarrow g(v,w) = f(x) \\ \Leftrightarrow\ & Q'(x,v,w) \Rightarrow (f(v) \circ w) = f(x) \end{aligned}$$

o sigui, que Q' ha de contenir la definició de g i, a més, els predicats de domini que fan que tot això tingui sentit:

$$Q'(x,v,w) = Q(x) \wedge D(v) \wedge D(w) \wedge (f(v) \circ w = f(x))$$

Ara és fàcil veure quina és la inicialització adequada dels paràmetres de g per tal de que calculi f: $g(x, n_\circ) = f(x)$; on $n_\circ$ és un neutre de l'operació $\circ$, o sigui, $w \circ n_\circ = w$. De fet n'hi ha prou amb que $n_\circ$ "es comporti com un neutre" per a x.

No sempre és possible d'obtenir una funció final equivalent. La cadena d'igualtats que ens permet d'obtenir la definició de g és el que ens permet saber si es pot o no. Recordem que, en el cas recursiu es fan 4 passos:

1. (generalització) $g(x, w)$ s'expressa en funció de $f(x)$.
2. (desplegat) $f(x)$ s'expressa en funció de $f(s(x))$.
3. (reorganització) L'expressió resultant s'escriu de manera que sigui possible efectuar el pas següent.
4. (plegat) S'identifica l'expressió anterior amb la funció g.

Si es pot reorganitzar l'expressió en el pas 3 per a plegar-la en el 4 la generalització era correcta i es pot obtenir una funció final equivalent. En cas contrari, o bé la generalització no era la correcta, o bé no es pot obtenir una funció final equivalent. Els noms "plegat" i "desplegat" provenen d'un article sobre transformació de programes [BD–77] en el que ja es proposava la tècnica que hem presentat però on no es mencionava com obtenir una generalització adequada.

Exemple 9. Cas amb operació binària associativa

Recordem la primera solució que hem obtingut del problema del producte de dos números naturals:

func $mul(x, y : nat)$ **ret** $(z : nat)$ **es**
  cas $y = 0 \rightarrow$ **ret** 0
    $\square$ $y > 0 \rightarrow$ **ret** $mul(x, y - 1) + x$
  fcas
ffunc

Notem que és un dels casos que discutiem: la forma de la funció pel cas recursiu és la desitjada, $+$ és una operació binària i associativa. La immersió adequada serà, doncs: $gmul(u, v, w) = mul(u, v) + w$. Seguint l'anàlisi de casos de *mul* tenim:

$$\begin{aligned} v = 0 \Rightarrow gmul(u, v, w) &= mul(u, v) + w = 0 + w = w \\ v > 0 \Rightarrow gmul(u, v, w) &= mul(u, v) + w = (mul(u, v - 1) + u) + w \\ &= mul(u, v - 1) + (u + w) \\ &= gmul(u, v - 1, u + w) \end{aligned}$$

I d'acord amb la discussió anterior la precondició de g serà:

$$Q'(x, y, u, v, w) = (mul(x, y) = mul(u, v) + w)$$

ja que totes les funcions de domini valen *cert* en aquest cas. A partir de la precondició anterior es pot observar que: $mul(x, y) = gmul(x, y, 0)$. L'algorisme per a la funció *gmul* es pot veure a continuació.

func $gmul(u, v, w : nat)$ **ret** $(z : nat)$ **es**
  cas $v = 0 \rightarrow$ **ret** w
    $\square$ $v > 0 \rightarrow$ **ret** $gmul(u, v - 1, u + w)$
  fcas
ffunc

Exemple 10. Cas general de transformació a recursió final

El següent exemple il·lustrarà el cas en el que la funció c no és una operació binària.

Hem de dissenyar una funció per avaluar un polinomi. Els coeficients estan disponibles en un vector $a[0{:}n]$ amb $n \geq 0$. L'especificació pot ser:

$\{n \geq 0\}$
func $poli(a : vector; n : ent; x : ent)$ **ret** $(y : ent)$
$\{y = \Sigma\, i{:}0 \leq i \leq n{:}\, a[i] * x^i\}$

A partir de la postcondició observem que és trivial calcular el valor d'un polinomi de grau zero: val $a[0]$. Aquest podria ser el cas simple. Solucionar recursivament els altres casos voldrà dir que hem d'expressar el valor d'un polinomi de grau n en termes d'un de grau $n - 1$, per exemple:

$$\Sigma\, i{:}0 \leq i \leq n{:}\, a[i] * x^i = (\Sigma\, i{:}0 \leq i \leq n - 1{:}\, a[i] * x^i) + a[n] * x^n$$

Aquesta expressió ens donaria directament el disseny del cas recursiu. Fixem-nos però que calcular x^n costaria $\Theta(n)$ ja que l'exponenciació no és pas una operació primitiva del nostre llenguatge.

La idea inicial de reduir el grau del polinomi en 1 és bona però es pot aplicar millor. Provem de reduir el grau del polinomi "per l'altra banda", o sigui:

$$\Sigma\, i{:}0 \leq i \leq n{:}\, a[i] * x^i = a[0] + \Sigma\, i{:}1 \leq i \leq n{:}\, a[i] * x^i$$

El sumatori involucra un coeficient menys però no és (encara) un polinomi de grau $n-1$. N'hi haurà prou amb treure x com a factor comú:

$$\begin{aligned}\Sigma i{:}0 \le i \le n{:}\, a[i] * x^i &= a[0] + x * \Sigma i{:}1 \le i \le n{:}\, a[i] * x^{i-1} \\ &= a[0] + x * \Sigma k{:}0 \le k \le n-1{:}\, a[k+1] * x^k\end{aligned}$$

Ara ja tenim un polinomi de grau $n-1$ però surt un nou problema: els coeficients no estan en els llocs 0 al $n-1$ de la taula. Per solucionar-ho farem una immersió de la funció original en una altra funció que ens permeti de tenir els coeficients a partir de l'índex que ens convingui. L'especificació serà:

$\{n \ge 0 \wedge 0 \le j \le n\}$
func *ipoli*$(a : vector; j, n : ent; x : ent)$ **ret** $(y : ent)$
$\{y = \Sigma i{:}0 \le i \le n{:}\, a[i+j] * x^i\}$

El lector pot estar pensant ara: quin ha estat el mètode de debilitament que hem aplicat? Només hem debilitat la part dreta: hem substituït constants pel paràmetre j. Ara bé, les constants no es veien a la postcondició perquè eren zeros que estaven sumant (o restant). Aquesta manera de fer les coses, cercar una possible relació de recurrència i després encaixar-hi la immersió adequada, és perfectament vàlida i molt útil per aquells problemes en els quals no està gaire clar quin mètode s'ha d'aplicar i com.

Igual que en el primer intent de solució el cas simple serà el polinomi de grau zero. Pel cas recursiu ja hem trobat abans la recurrència que ens convenia però particularitzada en el cas $j = 1$. Treballant amb la nova especificació trobarem la relació que necessitem:

$$\begin{aligned}\Sigma i{:}0 \le i \le n{:}\, a[i+j] * x^i &= a[j] + x * \Sigma i{:}1 \le i \le n{:}\, a[i+j] * x^{i-1} \\ &= a[j] + x * \Sigma k{:}0 \le k \le n-1{:}\, a[k+j+1] * x^k\end{aligned}$$

Per tant tenim les següents propietats que porten a escriure l'algorisme següent.

$$\begin{aligned}n = 0 &\Rightarrow ipoli(a, j, n, x) = a[j] \\ n > 0 &\Rightarrow ipoli(a, j, n, x) = a[j] + x * ipoli(a, j+1, n-1, x)\end{aligned}$$

func *ipoli*$(a : vector; j, n : ent; x : ent)$ **ret** $(y : ent)$ **es**
  cas $n = 0$ — **ret** $a[j]$
  □ $n > 0$ — **ret** $a[j] + x * ipoli(a, j+1, n-1, x)$
  fcas
ffunc

La comprovació detallada de la correcció de l'algorisme, incloent el preordre, es deixa d'exercici pel lector.

Notem que l'expressió del cas recursiu no s'ajusta al patró que s'ha donat a l'inici d'aquesta secció. Aquí tenim una combinació de dues operacions ($*$ i $+$). Malgrat l'aparença complicada, és fàcil aplicar el mètode de generalització per a obtenir una funció recursiva final equivalent.

La generalització s'obté substituint per variables totes les expressions que no involucrin ni la funció ni les operacions $*$ i $+$. Obtindrem:

$$gpoli(a', j', n', x', u, v) = u + v * ipoli(a', j', n', x')$$

Hem preferit posar uns noms als paràmetres de *gpoli* que recordin els corresponents de *ipoli* perquè els raonaments que segueixen siguin més fàcils de seguir.

La definició recursiva de *gpoli* l'obtindrem a partir de la cadena d'igualtats següent:

$$\begin{aligned}n > 0 \Rightarrow gpoli(a', j', n', x', u, v) &= u + v * ipoli(a', j', n', x') \\ &= u + v * (a'[j'] + x' * ipoli(a', j'+1, n'-1, x')) \\ &= (u + v * a'[j']) + (v * x') * ipoli(a', j'+1, n'-1, x')) \\ &= gpoli(a', j'+1, n'-1, x', u + v * a'[j'], v * x')\end{aligned}$$

Observem que ha estat possible efectuar la reorganització de l'expressió i identificar-la, a l'última igualtat, amb la funció *gpoli*. Pel cas simple s'obté: $gpoli(a, j, n, x, u, v) = u + v * a[j]$. L'algorisme de *gpoli* completat amb la precondició és:

$\{0 \leq n \wedge 0 \leq j \leq n \wedge ipoli(a, j, n, x) = u + v * ipoli(a', j', n', x')\}$
func $gpoli(a' : vector; j', n' : ent; x' : ent; u, v : ent)$ **ret** $(y : ent)$ **es**
  cas $n' = 0$ — **ret** $u + v * a'[j']$
    □ $n' > 0$ — **ret** $gpoli(a', j' + 1, n' - 1, x', u + v * a'[j'], v * x')$
  fcas
ffunc

La postcondició és la mateixa que la de *ipoli*.

A partir de la precondició podem deduir que: $ipoli(a, j, n, x) = gpoli(a, j, n, x, 0, 1)$. I si ajuntem aquest resultat amb $ipoli(a, 0, n, x) = poli(a, n, x)$ tenim: $poli(a, n, x) = gpoli(a, 0, n, x, 0, 1)$.

7. Referències

[AK–82] Arsac, J.; Kodratoff., Y. Some Techniques for Recursion Removal from Recursive Functions. *ACM Trans. on Programming Languages and Systems* 4, 2 (Apr. 1982), 295–322.

[BW–82] Bauer, F. L.; Wössner, H. *Algorithmic Language and Program Development*, Springer-Verlag 1982.

[BMPP–89] Bauer, F. L.; Möller, B.; Partsch, H.; Pepper, P. Formal Program Construction by Transformations—Computer-Aided, Intuition-Guided Programming. *IEEE Trans. on Software Engineering* 15, 2 (Feb. 1989), 165–180.

[BD–77] Burstall, R.; Darlington, J. A Transformation System for Developing Recursive Programs. *Journal of the ACM* 24, 1 (Jan. 1977), 44–67.

[CIP–85] The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L. Lecture Notes in Computer Science 183, Springer-Verlag 1985.

[Dij–76] Dijkstra, E. *A Discipline of Programming*, Prentice-Hall 1976.

[DM–87] Dosch, W.; Möller, B. Seminar on Functional Programming, Madrid, October 1987.

[Fea–87] Feather, M. S. A Survey and Classification of some Program Transformation Approaches and Techniques. In L. Meertens (ed.): *Program Specification and Transformation*. North-Holland 1987, 165–195.

[FH–88] Field, A.; Harrison, P. *Functional Programming*, Addison-Wesley 1988.

[Gri–81] Gries, D. *The Science of Programming*, Springer-Verlag 1981.

[PK–82] Paige, R.; Koenig, S. Finite Differencing of Computable Expressions, *ACM Trans. on Programming Languages and Systems* 4, 3 (July 1982), 402–454.

[RBP–84] Rosselló, C.; Balcázar, J. L.; Peña, R. Deriving Specifications of Embeddings in Recursive Program Design, *Structured Programming* 10, 3 (Aug. 1989), 133–145.

[Sch–84] Scholl, P. *Algorithmique et représentation des données 3. Récursivité et arbres*, Masson 1984.