Implementierung von Genotype und Phenotype im Sinne des Gene Expression Programming in KIV

Prüfungsrelevante Studienleistung
zur Lehrveranstaltung Programmverifikation
Prof. Dr. U. Petermann

vorgelegt von Matthias Jauernig
Fachbereich Informatik, Mathematik und Naturwissenschaften
Hochschule für Technik, Wirtschaft und Kultur Leipzig

Leipzig, den 12. Februar 2007

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Aufgabenstellung

Die Lehrveranstaltung Programmverifikation behandelt Methoden der Spezifikation und Verifikation von Algorithmen und Datentypen unter Zuhilfenahme des Verifikationssystems KIV. Mit Hilfe dieser Software waren die Datentypen Genotype und Phenotype im Sinne des *Gene Expression Programming* zu spezifizieren und die gegenseitige Umwandlung zu verifizieren.

1.2 Gene Expression Programming

Bei Gene Expression Programming (GEP, siehe [1, 2]) sollen Computerprogramme unter Zuhilfenahme evolutionärer Algorithmen entwickelt werden. Die Individuen bestehen hier aus Genotypen und Phenotypen. Genotypen stellen in der Biologie das Erbgut eines Organismus dar, während durch den Phenotyp dessen Erscheinungsbild festgelegt wird.

1.3 Genotype

Genotypen sind Chromosomen fixer Länge, welche aus einem oder mehreren Genen bestehen können. Kurz zum Aufbau eines Chromosoms im Sinne des GEP: es besitzt eine feste Länge und besteht aus einzelnen Zeichen, die entweder n-stellige Funktionssymbole (FS) oder (0-stellige) Terminalsymbole (TS) darstellen. Ein Chromosom als Ganzes ist somit eine Zeichenkette, kann logisch jedoch aus mehreren Genen bestehen. Solch ein Gen ist somit wiederum eine Zeichenkette aus FS und TS und besteht strukturell aus einem Kopf sowie einem Rest. Im Kopf können FS und TS beliebig aneinander gereiht vorkommen, wohingegen im Rest nur TS erlaubt sind. Zwischen der *Länge des Kopfes* h und der *Länge des Restes* t besteht folgende Beziehung (n sei dabei die *höchste Arität eines FS*):

$$t = h * (n - 1) + 1$$

Dadurch wird gesichert, dass jedes Gen mit einer solchen Struktur durch einen ET dargestellt werden kann. Die Länge des Kopfes ist a priori festzulegen, wodurch die Länge des Restes und schließlich auch die Länge des ganzen Genes berechnet werden kann.

1.4 Phenotype

Phenotypen stellen die Repräsentation der Chromosomen (Genotypen) als Ausdrucksbäume (*expression tree*, ET) dar. Ein Phenotyp lässt sich aus einem Genotyp als ET gewinnen. Speziell kann für jedes Gen eines Chromosoms ein eigener Unterbaum (Sub-ET) erzeugt werden (solche Unterbäume lassen sich dann wieder zusammenfügen, doch dies soll hier nicht weiter Bestandteil der Betrachtung sein).

1.5 Ordnung in Genotype und Phenotype

Die Ordnung im Genotype und wie sie im Phenotype wider gespiegelt wird stellt eine wichtige Eigenschaft bei der gegenseitigen Umwandlung der beiden Datenstrukturen dar. Ein Baum wird aus einem Gen durch *ebenenweises Auffüllen* noch zu besetzender Söhne erzeugt, wobei eine Ebene von n Knoten genau n zusammen hängenden Symbolen im Gen entspricht. Das folgende Beispiel verdeutlicht diese Ordnungseigenschaft:

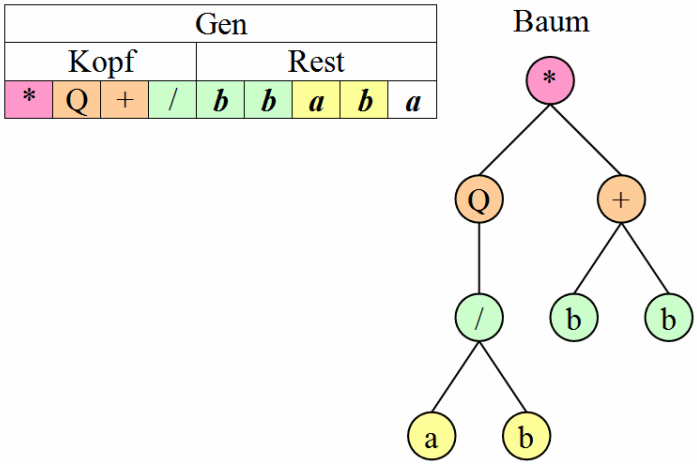


Abbildung 1.1: Beispiel zur Ordnungseigenschaft

Unter Voraussetzung dieses Ordnungsbegriffes lassen sich nun die Umwandlungen von Phenotype in Genotype und umgekehrt definieren.

1.6 Transformation von Genotype in Phenotype

Betrachtet werden soll das folgende Beispiel eines Genotypes (Chromosoms):

Chromosom																								
Gen 1									Gen 2								Gen 3							
Kopf1				Rest1					Kopf2				Rest2				Kopf3				Rest3			
/	*	a	Q	*a*	*b*	*a*	*a*	*a*	+	a	b	/	*a*	*a*	*b*	*b*	*b*	*	Q	+	/	*b*	*b*	*a*	*b*	*a*

Abbildung 1.2: Genotype-Beispiel

Zur Umwandlung eines Genotypes in einen Phenotype werden alle Gene unabhängig voneinander in den jeweiligen Ausdrucksbaum (expression tree, *ET*) überführt. Gene fixer Länge können Ausdrucksbäume unterschiedlicher Länge erzeugen, abhängig von den Aritäten der verwendeten Gensymbole. Wieviele Symbole eines Genes tatsächlich in einen Ausdrucksbaum überführt werden, wird durch den *kodierenden Teil* (coding part, *CP*) des Genes festgelegt, der von den Aritäten der darin befindlichen Symbole abhängt. Ein Symbol *s* mit der Arität n muss im Ausdrucksbaum durch einen Knoten mit n Söhnen repräsentiert werden, demzufolge müssen mindestens n weitere Symbole nach *s* im Gen betrachtet werden. Gehört *s* zum CP eines Genes, dann auch die Symbole, welche dessen Söhne im Ausdrucksbaum repräsentieren. Das erste Symbol eines Genes, also die Wurzel des zugehörigen ET, ist auf jeden Fall Bestandteil des CP.

Die Transformation kann wie folgt durchgeführt werden. Es wird sequentiell jedes Gensymbols eines Gens betrachtet und in einen Knoten umgewandelt. Das erste Symbol eines Gens entspricht der Baumwurzel. Deren Söhne müssen im weiteren Verlauf aufgefüllt werden. So wird stets das nächste Gensymbol aus dem Gen entnommen und an der nächsten offenen Stelle im Baum (dort wo noch Söhne fehlen) eingefügt, wobei die bereits dargestellte Ordnungs-Eigenschaft verwendet wird. Sind keine Söhne mehr zu besetzen, so befinden sich in der Blattebene nur noch Terminalsymbole und das Verfahren terminiert. Die Anzahl der betrachteten Gensymbole als auch die Anzahl der Knoten im erzeugten Baum entspricht der Länge des CP.

Die Transformation des in Abbildung 1.2 dargestellten Beispiels führt zu den Bäumen in Abbildung 1.3 des nächsten Abschnitts.

1.7 Transformation von Phenotype in Genotype

Betrachtet werden soll das folgende Beispiel eines Phenotypes, der aus 3 Ausdrucksbäumen besteht:

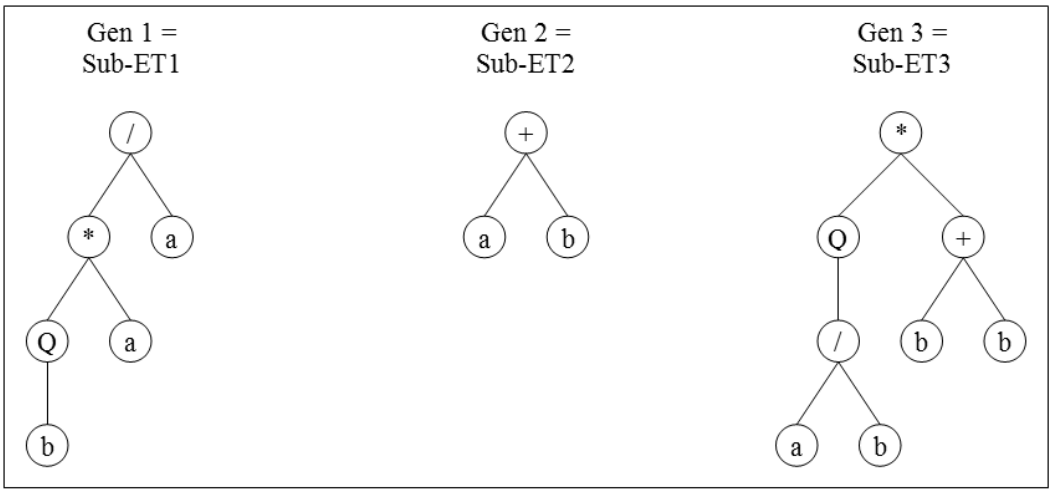


Abbildung 1.3: Phenotype-Beispiel

Zusammenfassen lassen sich diese 3 Bäume als eine Liste von Bäumen. Zur Umwandlung des Phenotypes in den entsprechenden Genotype ist jeder Baum der Liste in ein entsprechendes Gen zu überführen.

Auch bei der Umwandlung von Phenotype in Genotype ist deren ebenenweise Ordnungs-Beziehung zu berücksichtigen. So könnte das Gen aus einem Baum durch eine Art Breitensuche erzeugt werden, indem jede Ebene sequentiell durchlaufen wird und das Gen durch Anfügen des Gensymbols aus dem jeweils betrachteten Knoten aufgebaut wird. Ist die Prozedur beendet, so erhält man ein Gen, dessen Länge der Anzahl der Knoten im Baum entspricht.

Durch Umwandlung des in Abbildung 1.3 dargestellten Phenotypes erhält man den folgenden Genotype:

Chromosom																
Gen 1						Gen 2			Gen 3							
/	*	a	Q	a	b	+	a	b	*	Q	+	/	b	b	a	b

Abbildung 1.4: Rücktransformation des Phenotypes

Der Phenotype aus Abbildung 1.3 entstand durch Transformation des Genotypes aus Abbildung 1.2. Die Gene in Abbildung 1.4 entsprechen den Anfangsstücken der Gene aus Abbildung 1.2 und stellen deren kodierende Teile dar. Der Ausgangs-Genotype entspricht nicht dem der Rücktransformation, sodass die beiden Operationen der Umwandlung von Genotype in Phenotype und von Phenotype in Genotype nicht als invers anzusehen sind.

Kapitel 2

Das KIV-Projekt GenoPhenoType

2.1 Grundlegender Überblick

Auf Grundlage der ursprünglichen Beschreibung des Gene Expression Programming nach [2] wurde ein KIV-Projekt „GenoPhenoType“ angelegt. Dieses beinhaltet zum einen die formale Spezifikation der Datenstrukturen Genotype und Phenotype, was nur auf Grundlage weiterer Sorten möglich war. Zum anderen wurden die Operationen der gegenseitigen Transformation dieser Datenstrukturen spezifiziert und über zwei Module implementiert, deren Korrektheit in Bezug auf die zugrunde gelegten Export-Spezifikationn verifiziert werden konnten.

Es wurde versucht, soviele Bibliotheken wie möglich zu nutzen, auf deren Korrektheit sich verlassen werden konnte. Dies war jedoch nicht in allen Fällen möglich. Große Mühe machte die Spezifikation der Datenstruktur *tree*, über welche allgemeine Baumstrukturen dargestellt werden können. Die Söhne eines Baumknotens sind wieder als Liste von Baumknoten aufzufassen. Sinnvoll wäre es gewesen, dafür die Sorte *list* aus der Bibliothek zu verwenden. Dies war jedoch nicht möglich, da rekursive Datenstrukturen (in diesem Fall ein Baumknoten und die Liste von Söhnen, die ebenfalls wieder Baumknoten sind) innerhalb einer Datei spezifiziert werden müssen und sich dafür eben keine „externen“ Spezifikationen nutzen lassen. Sowohl hier als auch bei der Spezifikation der zahlreichen weiteren benötigten Sorten musste sehr viel Zeit investiert werden.

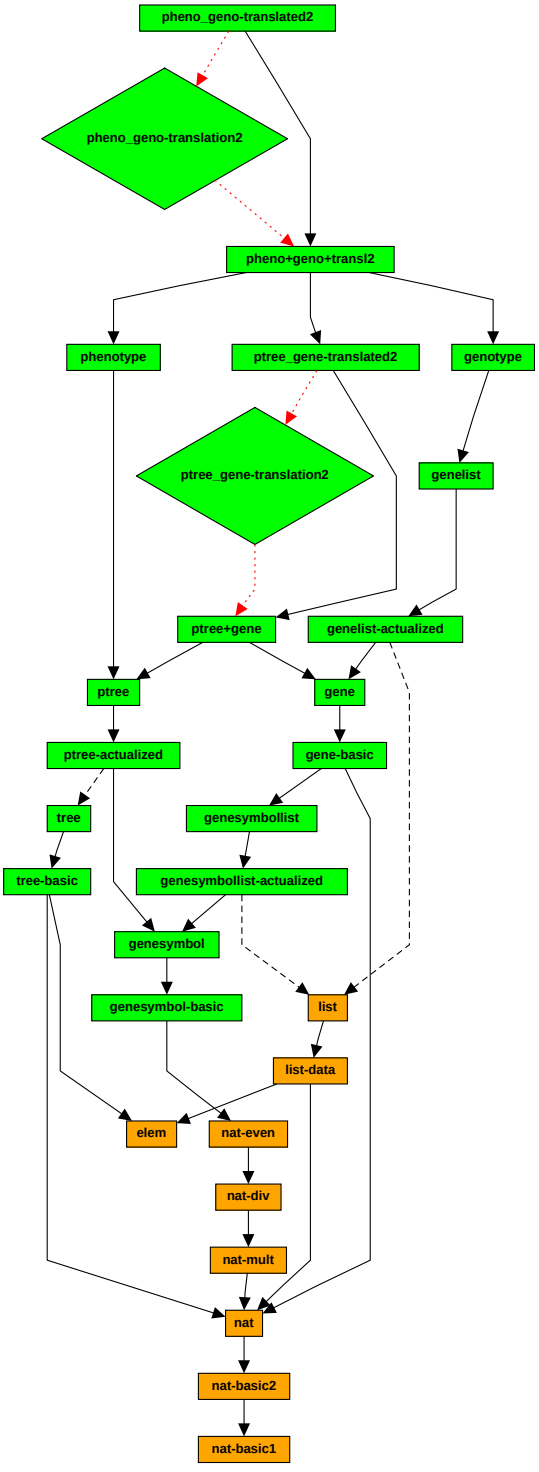
2.2 Verwendete Bibilotheks-Spezifikationen

Folgende Bibliotheken fanden in diesem Projekt Verwendung:

nat-basic-1
nat-basic-2
nat
nat-mult
nat-div
nat-even
elem
list-data
list

elem wurde dabei beispielsweise für die generische Spezifikation der Datenstruktur Baum verwendet, *list* fand sowohl bei der Definition eines Gens als eine Liste von Gensymbolen als auch bei der Datenstruktur Genotype als eine Liste von Genen Verwendung.

2.3 Der Projektgraph



2.4 Spezifikationen

2.4.1 Datenstrukturen

Genotype-spezifische Datenstrukturen

---

genesymbol-basic =
data specification
    using nat-even
    genesymbol = makeGS (. .symbol : nat ; . .arity : nat ;);
    variables gs, gs1, gs2: genesymbol;
end data specification

Generelle Beschreibung:
Diese Spezifikation beschreibt den Datentyp des Gensymbols. Ein Gensymbol besteht aus einem Symbol, welches in der Realität einem Operator oder einem Operanden entspricht und einer Stelligkeit (Arität) dieses Symbols. Funktionssymbole haben eine Arität ¿ 0, während Terminalsymbole Operanden darstellen und somit die Stelligkeit 0 besitzen. Als Datentyp des Symbols wurde *nat* gewählt, welches im Gegensatz zum auch denkbaren *char* einen unbegrenzten Umfang bietet. Nachteilig dabei ist, dass sich von den Symbolen nicht auf die dahinter stehenden Bedeutungen schließen lässt.

---

genesymbol =
enrich genesymbol-basic **with**
    predicates
        . .isTerminal : genesymbol;
        . .isFunction : genesymbol;

axioms

IsTerminal : gs.isTerminal ↔ gs.arity = 0;

IsFunction : gs.isFunction ↔ ¬gs.isTerminal;

end enrich

Generelle Beschreibung:
Die Spezifikation **genesymbol-basic** wird hierdurch angereicht durch die Funktionen *.isTerminal* und *.isFunction* sowie dazu gehörige Axiome. Ein Gensymbol ist genau dann ein Terminalsymbol (*.isTerminal*, ausgedrückt durch das Axiom *IsTerminal*), wenn es die Arität 0 hat. Ansonsten handelt es sich um ein Funktionssymbol (*.isFunction*, Axiom *IsFunction*).

---

genesymbollist-actualized =
actualize list **with** genesymbol **by morphism**
elem → genesymbol; list → genesymbollist; x → gsl; x0 → gsl0; x1 → gsl1;
x2 → gsl2; y → gsl3; y0 → gsl4; y1 → gsl5; y2 → gsl6; z → gsl7; z0 → gsl8;
z1 → gsl9; z2 → gsl10; a → gs; a0 → gs0; b → gs1; c → gs2
end actualize

Generelle Beschreibung:
Hierbei handelt es sich lediglich um eine Aktualisierung durch Morphismus der generischen Spezifikation *list* mit *genesymbol*, wobei *list* in die neue Sorte *genesymbollist* überführt wird. D. h. dass es sich bei *genesymbollist* nachfolgend um eine Liste handelt, die Gensymbole als Elemente enthält.

---

genesymbollist =
enrich genesymbollist-actualized **with**

functions

inv	:	genesymbollist	→	genesymbollist	;
. .highestArity	:	genesymbollist	→	nat	;
#cp	:	genesymbollist	→	nat	;
#cpRec	:	genesymbollist × nat	→	nat	;

predicates

. .isEmpty	:	genesymbollist;
isEndList	:	genesymbollist × genesymbollist;

axioms

IsEmpty : gsl.isEmpty ↔ gsl = [];

IsEndList : isEndList(gsl, gsl1) ↔ (∃ gsl2. gsl = gsl2 + gsl1);

Inv1 : inv([]) = [];

Inv2 : inv(gs ' + gsl) = inv(gsl) + gs ';

HighestArity1 : [].highestArity = 0;

HighestArity2 : (gs + gsl).highestArity = max(gs.arity, gsl.highestArity);

HighestArity3 : (gsl + gsl1).highestArity = max(gsl.highestArity, gsl1.highestArity);

CodingPart1 : #cp([]) = 0;

CodingPart2 : #cp(gs + gsl) = #cpRec(gsl, gs.arity) + 1;

CodingPart3 : #cpRec([], n) = 0;

CodingPart4 : #cpRec(gsl, 0) = 0;

CodingPart5 : n > 0 → #cpRec(gs + gsl, n) = #cpRec(gsl, n + gs.arity - 1) + 1;

end enrich

Generelle Beschreibung:
Hierdurch wird die in *genesymbollist-basic* eingeführte Sorte *genesymbollist* um weitere Funktionen und Prädikate sowie Axiome für diese angereichert.

Funktionen:

- **inv:** Diese einstellige Funktion übernimmt eine *genesymbollist* und invertiert diese. Die beiden Axiome *Inv1* und *Inv2* beschreiben diesen Prozess rekursiv. Bei *Inv2* wird das erste Element der übergebenen Liste abgetrennt und es findet ein rekursiver Aufruf für die Restliste statt. An dessen Ergebnis wird das abgetrennte Element hinten angefügt, wodurch sich die Invertierung ergibt.
- **.highestArity:** Hierbei handelt es sich um eine einstellige (Postfix-) Funktion, welche die höchste in einer *genesymbollist* vorkommende Arität ermittelt und zurückgibt. Die Axiome *HighestArity1* und *HighestArity2* beschreiben diesen Prozess rekursiv unter Nutzung der Funktion *max* aus der Bibliotheks-Spezifikation *nat-even*. *HighestArity3* beschreibt das Erhalten der höchsten Arität von zwei miteinander verketteten Listen.
- **#cp:** Hierdurch wird die Länge des *kodierenden Teils* (coding part, *CP*) einer *genesymbollist* berechnet und zurück gegeben. Dabei handelt es sich um genau die Anzahl von Gensymbolen, die bei einer Transformation in den Phenotype in dessen Baum auftreten. Der CP wird wird durch die Funktion *#cpRec* rekursiv berechnet. Die Axiome *CodingPart1* und *CodingPart2* sind für die Spezifikation von *#cp* zuständig, wobei die Länge des CP einer leeren Liste 0 ist und anderenfalls der rekursive Aufruf für das erste Symbol der Liste mit dessen Arität stattfindet. Die Arität gibt an, wieviele Söhne für einen Baumknoten basierend auf diesem Symbol zu schließen sind, wieviele weitere Symbole der Liste also mindestens noch betrachtet werden müssen.
- **#cpRec:** Hierdurch wird die Länge des CP einer *genesymbollist* rekursiv berechnet und zurückgegeben. Die Axiome *CodingPart3* und *CodingPart4* stellen die Endpunkte der Rekursion dar: ist die Liste leer oder sind keine Symbole mehr zu betrachten, so wird 0 für die aktuelle Länge des CP zurück gegeben. Axiom *CodingPart5* ist für die Rekursion bei der Berechnung verantwortlich. Die Länge des CP wird berechnet, indem die *genesymbollist* sequentiell durchlaufen wird (durch die rekursiven Aufrufe von *#cpRec*. Bei jedem rekursiven Aufruf, der nicht zu einer Termination der Rekursion führt, wird das zweite Argument um die Stelligkeit des aktuellen Gensymbols erhöht und 1 subtrahiert. Dieses zweite Argument stellt die aktuelle Anzahl noch zu betrachtender Gensymbole der Liste dar. Sie korrespondiert mit der Anzahl von Sohn-Knoten, die bei einer Transformation in den Phenotype in der jeweiligen Situation noch besetzt werden müssen. Durch den aktuellen Aufruf wird ein

Knoten besetzt, daher kann 1 subtrahiert werden. Andererseits werden bei Funktionssymbolen (im Gegensatz zu Terminalsymbolen) neue Knoten geöffnet, die wiederum besetzt werden müssen. Daher wird die Arität des Symbols addiert. Die Struktur eines Gens sichert ab, dass 0 als Wert des zweiten Arguments auf jeden Fall erreicht wird (spätestens nach der Betrachtung des gesamten Gens), da im Gen-Rest nur Terminalsymbole vorkommen und dieser genügend lang ist.

Prädikate:

- **.isEmpty:** Ist erfüllt, wenn die bezeichnende *genesymbollist* leer ist. Spezifikation über das Axiom *IsEmpty*.
- **isEndList:** Übernimmt zwei *genesymbollist*'s als Argumente. Das Prädikat ist wie in Axiom *IsEndList* spezifiziert erfüllt, wenn die zweite Liste gsl1 Teilliste der ersten gsl ist und an deren Ende vorkommt, es also eine Liste gsl2 gibt sodass gilt:
  $gsl = gsl2 + gsl1$.

---

gene-basic =
data specification
using genesymbollist, nat
gene = makeGene (. .list : genesymbollist ; . .headLength : nat ;);
variables g, g1, g2: gene;
end data specification

Generelle Beschreibung:
Hier wird die Sorte *gene* spezifiziert. Ein Gen wird bestimmt durch die eigentliche Liste von Gensymbolen (*genesymbollist*) sowie die Länge des Gen-Kopfes, welche a priori zu übergeben ist und im in Abschnitt 1.3 beschriebenen Verhältnis zur Länge des Gen-Restes stehen muss.

---

gene =
enrich gene-basic **with**
functions

. .tailLength	:	gene	→	nat	;
. .highestArity	:	gene	→	nat	;
# .	:	gene	→	nat	;
. .first	:	gene	→	genesymbol	;
. .rest	:	gene	→	gene	;

predicates

. ∈ .	:	genesymbol × gene;
correctGene	:	gene;

axioms

Element : gs ∈ g ↔ gs ∈ g.list;

GeneFirst : g.list ≠ [] → g.first = g.list.first;

GeneRest : g.list ≠ [] ∧ g1.list = g.list.rest → g.rest = g1;

GeneLength : # g = # g.list;

TailLength : g.tailLength = g.headLength ∗ (g.highestArity - 1) + 1;

HighestArity :
g.highestArity = m ↔ (∃ gs. gs ∈ g ∧ gs.arity = m) ∧ (∀ gs. gs ∈ g → ¬gs.arity > m);

HighestArity1 : g.highestArity = g.list.highestArity;

CorrectGene :
correctGene(g)
↔ # g = g.headLength + g.tailLength
∧ (∀ gsl1, gsl2.
gsl1 = g.headLength ∧ # gsl2 = g.tailLength ∧ g.list = gsl1 + gsl2
→ ∀ gs. gs ∈ gsl2 → gs.isTerminal);

end enrich

Generelle Beschreibung:
Hier wird die in *gene-basic* erzeugte Sorte *gene* um weitere Funktionen und Prädikate angereichert. Die meisten davon stellen Abbildungen auf die darunter liegende Datenstruktur *genesymbollist* dar und sollen hier nicht weiter beschrieben werden. Dabei handelt es sich um *.highestArity*, *#*, *.first*, *.rest* und ∈. Übrig bleibt zum einen die Funktion *.tailLength*, welche über das Axiom *TailLength* die Länge des Gen-Rests nach der in Abschnitt 1.3 angegebenen Formel berechnet. Zum anderen gilt das Prädikat *correctGene* zu nennen, welches die Korrektheit eines erzeugten Genes prüft. Ein Gen gilt als korrekt, wenn sich seine Länge aus der Länge des Kopfes und der des Restes ergibt und wenn es sich bei allen Symbolen im Gen-Rest um Terminalsymbole (die eine Arität 0 besitzen) handelt.

---

genelist-actualized =
actualize list **with** gene **by morphism**
elem → gene; list → genelist; a → g; a0 → g0; b → g1; c → g2; x → gl; x0 → gl0; x1 → gl1; x2 → gl2; y → gl3; y0 → gl4; y1 → gl5; y2 → gl6; z → gl7; z0 → gl8; z1 → gl9; z2 → gl10
end actualize

Generelle Beschreibung:
Durch diese Spezifikation wird die Datenstruktur der Genliste (Sorte *genelist*) in Vorbereitung der Definition des Genotypes eingeführt. Eine Genliste besteht aus einer Liste einzelner Gene. Dazu wird die generische Bibliotheks-Spezifikation *list* mit der Sorte *gene* durch Morphismus aktualisiert.

---

genelist =

---

enrich genelist-actualized **with**

end enrich

Generelle Beschreibung:
Diese Spezifikation wurde angelegt, um der Sorte *genelist* weitere Funktionen und Prädikate sowie entsprechende Axiome hinzuzufügen. Es bestand im Nachhinein keine Notwendigkeit von dieser Möglichkeit Gebrauch zu machen, wodurch diese Anreicherung leer geblieben ist.

---

genotype =
data specification
using genelist
genotype = makeGT (. .genes : genelist ;);
variables gt, gt1, gt2: genotype;
end data specification

Generelle Beschreibung:
Diese Spezifikation beinhaltet abschließend die Einführung der Sorte *genotype* als Abbildung eines Genotypes. Ein Genotype wird lediglich durch eine *genelist* als Liste von Genen gekennzeichnet, was die Einführung dieser neuen Sorte getrennt von *genelist* eigentlich überflüßig macht, zur Wahrung der Analogie zu *phenotype* jedoch beibehalten wurde.

Phenotype-spezifische Datenstrukturen

---

tree-basic =
generic data specification
parameter elem **using** nat
tree = empty
| node (. .value : elem ; . .subtrees : treelist ;) **with** is_node
;
treelist = []
| . + . **prio** 9 (. .first : tree ; . .rest : treelist ;) **prio** 9
;
variables

t, t0, t1, t2, t3: tree;
tl, tl0, tl1, tl2, tl3: treelist;
size functions # . : treelist → nat ;
order predicates . < . : treelist × treelist;
end generic data specification

Generelle Beschreibung:
Hierdurch findet die generische Datentyp-Spezifikation einer allgemeinen Baumstruktur statt. Der Parametertyp dabei ist *elem* und kann durch Morphismus aktualisiert werden. Er gibt den Datentyp des Inhaltes eines Baumknotens an. Es werden zwei Sorten eingeführt: *tree* und *treelist*. *tree* stellt einen Baumknoten dar und kann generiert werden als leer (*empty*) oder als Knoten (*node*) mit einem Wert (*.value*) und einer Liste von Sohnknoten (*subtrees*). *treelist* stellt eine Listenstruktur dar, die Elemente vom Typ *tree* aufnehmen kann. Es ist zu beachten, dass die Definition von *tree* und *treelist* rekursiv erfolgt, da ein Baumknoten als Söhne eine Liste aufnimmt, deren Elemente wiederum Bäume sind. Genau aus diesem Grund konnte auch nicht die Bibltiotheks-Spezifikation *list* für die *treelist* verwendet werden, da bei rekursiven Definitionen die Sorten-Spezifikationen innerhalb derselben Datei erfolgen müssen. Das ist bedauerlich, da so die Listenstruktur erneut eingeführt werden muss und benötigte Axiome sowie Lemmas neu definiert bzw. bewiesen werden müssen.

---

tree =
enrich tree-basic **with**
functions

. ’	:	tree	→	treelist	;
. + .	:	treelist × treelist	→	treelist	**prio** 9;
. + .	:	treelist × tree	→	treelist	**prio** 9;
#nl	:	tree	→	nat	;
#nl	:	treelist	→	nat	;
#n	:	tree	→	nat	;
#n	:	treelist	→	nat	;
#l	:	tree	→	nat	;
#l	:	treelist	→	nat	;
inv	:	treelist	→	treelist	;

predicates

. ∈ .	:	tree × treelist;
. ∈ .	:	elem × tree;
. ∈ .	:	elem × treelist;
. .isLeaf	:	tree;

axioms

Nil : [] + tl = tl;

Cons : (t + tl) + tl1 = t + tl + tl1;

One : t ' = t + [];

Last : tl + t = tl + t ';

In : t ∈ tl ↔ (∃ tl1, tl2. tl = tl1 + t + tl2);

Len-1 : # tl = # tl0 → (tl + tl1 = tl0 + tl2 ↔ tl = tl0 ∧ tl1 = tl2);

IsLeaf : node(a, tl).isLeaf ↔ tl = [];

Nodes2 : #nl(node(a, tl)) = #nl(tl) + 1;

Nodes3 : #nl([]) = 0;

Nodes4 : #nl(t + tl0) = #nl(t) + #nl(tl0);

Nodes5 : #nl(tl + tl0) = #nl(tl) + #nl(tl0);

INodes1 : t.isLeaf → #n(t) = 0;

INodes2 : ¬node(a, tl).isLeaf → #n(node(a, tl)) = #n(tl) + 1;

INodes3 : #n([]) = 0;

INodes4 : #n(t + tl) = #n(t) + #n(tl);

INodes5 : #n(tl + tl0) = #n(tl) + #n(tl0);

Leaves : #l(t) = #nl(t) - #n(t);

Leaves1 : #l(tl) = #nl(tl) - #nl(tl);

In1 : a = t.value → a ∈ t;

In2 : t.value ≠ a ∧ a ∈ t.subtrees → a ∈ t;

In3 : a = t.value → a ∈ t + tl;

In4 : a ≠ t.value ∧ a ∈ tl → a ∈ t + tl;

In5 : ¬a ∈ [];

Inv1 : inv([]) = [];

Inv2 : inv(t ' + tl) = inv(tl) + t ';

Inv3 : inv(tl) = inv(tl0) ↔ tl = tl0;

end enrich

Generelle Beschreibung:
Durch diese Spezifikation findet eine Anreicherung der Sorten *tree* und *treelist* mit weiteren Funktionen, Prädikaten und Axiomen statt. Vor allem für *treelist* ist dies notwendig und wichtig, um eine mit der Bibliotheks-*list* vergleichbare Basisfunktionalität zu schaffen. Dazu wurden die Axiome *Nil, Cons, One, Last* und *In* aus *list* entnommen und hier eingeführt. Das Axiom *Len-1* stellt ein Lemma aus *list* dar, welches dort bewiesen wurde. Aufgrund der Schritte, die dazu erforderlich gewesen wären (es hätte ein Großteil der Lemmas aus *list* übernommen und ebenfalls bewiesen werden müssen), wurde es hier als Axiom definiert.

Funktionen:

- **':** Dies stellt die Erzeugung der Einerliste bestehend aus dem übergebenen *tree* dar.

- **+:** Hierdurch werden *treelist*'s und *tree*'s miteinander verknüpft.
- **inv:** Mit dieser Funktion wird die übergebene Liste invertiert. Die Axiome *Inv1* und *Inv2* beschreiben diesen Vorgang rekursiv: mit der leeren Liste endet die Rekursion, ansonsten ist das erste Element abzutrennen, der Rest der Liste zu invertieren und das Element dann von hinten anzufügen. Axiom *Inv3* beschreibt die offensichtliche Beziehung, dass zwei Listen genau dann gleich sind, wenn auch ihre Invertierten übereinstimmen.
- **#nl:** Hierdurch werden alle Knoten (also innere Knoten und Blätter) eines *tree* bzw. einer *treelist* gezählt. Die Axiome *Nodes3* und *Nodes4* beschreiben diesen Vorgang rekursiv für *treelist*: bei einer leeren Liste ist die Anzahl der Knoten 0, ansonsten wird der erste Baumknoten von der Liste abgetrennt, die Anzahl der Knoten ergibt sich dann aus der Anzahl der Knoten dieses ersten Baumknotens addiert mit der Anzahl der Knoten der Restliste. Axiom *Nodes5* definiert die Anzahl der Knoten für zwei miteinander verkettete Listen. Durch das Axiom *Nodes2* wird die Zählung der Knoten rekursiv für *tree* beschrieben. So wird die Zählung eines mit *node* erzeugten Baumknotens für die Sohnliste aufgerufen und für den aktuellen Knoten 1 addiert.
  → ***Bemerkung:*** Die Funktion *#nl* ist elementar für die Verifikation der Transformation von Phenotype zu Genotype. Dabei wird eine Induktion über die Länge einer *treelist* geführt. In jedem Schritt wird der erste Knoten der *treelist* entfernt und dessen Söhne hinten angehangen. Durch das bewiesene Lemma *nl-1* konnte gezeigt werden, dass bei diesem Schritt die Anzahl der Knoten der *treelist* um 1 verringert wird, womit eine Induktion geführt werden kann.
- **#n:** Hierdurch werden die inneren Knoten (also ohne Blätter) eines *tree* bzw. einer *treelist* gezählt. Die Axiome *INodes3* und *INodes4* beschreiben diesen Vorgang rekursiv für *treelist*: bei einer leeren Liste ist die Anzahl der Knoten 0, ansonsten wird der erste Baumknoten von der Liste abgetrennt, die Anzahl der inneren Knoten ergibt sich dann aus der Anzahl der inneren Knoten dieses ersten Baumknotens addiert mit der Anzahl der inneren Knoten der Restliste. Axiom *INodes5* definiert die Anzahl der inneren Knoten für zwei miteinander verkettete Listen. Durch die Axiome *INodes1* und *INodes2* wird die Zählung der inneren Knoten rekursiv für *tree* beschrieben. Ist der betreffende Baumknoten ein Blatt, so wird 0 zurück gegeben, da Blätter nicht mitgezählt werden sollen. Ansonsten wird die Zählung für die Sohnliste fortgesetzt und für den aktuellen Knoten 1 addiert.
- **#l:** Mit dieser Größenfunktion werden die Blätter in einem *tree* bzw. einer *treelist* gezählt. Die Axiome *Leaves* und *Leaves1* zeigen, dass dazu auf die Funktionen *#nl* sowie *#n* zurück gegriffen wird, um sich eine erneute rekursive Definition zu ersparen.

Prädikate:

- **∈:** Hierdurch und spezifiziert durch die Axiome *In, In1-In5* wird ausgedrückt, wann ein Element in einer Liste enthalten ist.
- **.isLeaf:** Dieses Prädikat ist genau dann erfüllt, wenn es sich bei dem bezeichnenden *tree* um einen Blattknoten handelt. Dies wird durch das Axiom *IsLeaf* ausgedrückt: ein Baumknoten ist genau dann Blatt, wenn die Liste seiner Söhne leer ist.

---

ptree-actualized =
actualize tree **with** genesymbol **by morphism**
elem → genesymbol; tree → ptree; treelist → ptreelist
end actualize

Generelle Beschreibung:
Hierbei wird die generische Sorte *tree* mit *genesymbol* durch Morphismus aktualisiert, d. h. es wird eine neue Sorte *ptree* eingeführt, welche einen Baum darstellt, deren Knoten als Inhalt Gensymbole besitzen. Die Sorte *treelist* wird weiterhin abgebildet auf *ptreelist*. Die Namen *ptree* sowie *ptreelist* stehen für „phenotype tree“ und „phenotype treelist“.

---

ptree =
enrich ptree-actualized **with**
predicates correctPNode : ptree;

axioms

CorrectPNode : correctPNode(node(gs, tl)) ↔ # tl = gs.arity;

end enrich

Generelle Beschreibung:
Hierdurch findet eine Anreicherung der Sorte *ptree* mit weiteren Syntax-Elementen statt. Genauer gesagt wird hier das Prädikat *correctPNode* eingeführt, welches beschreibt, wann es sich bei einem Knoten (*ptree*) um einen korrekten Baumknoten handelt. Dies ist genau dann der Fall, wenn die Länge der Sohnliste der Arität des Gensymbols des Baumknotens entspricht.

---

phenotype =
data specification
using ptree
phenotype = makePT (. .ptrees : ptreelist ;);
variables pt, pt1, pt2: phenotype;
end data specification

Generelle Beschreibung:
Diese Spezifikation beinhaltet abschließend die Einführung der Sorte *phenotype* als Abbildung eines Phenotypes. Ein Phenotype wird dabei gekennzeichnet durch eine *ptreelist*, also eine Liste von Bäumen, welche auf einen Genotypen als eine Liste einzelner Gene abgebildet werden können.

2.4.2 Transformationen zwischen Genotype und Phenotype

---

ptree+gene = ptree + gene

Generelle Beschreibung:
Dies stellt lediglich eine Vereinigung der Spezifikationen *ptree* und *gene* dar, um diese beiden Sorten bei ihrer gegenseitigen Transformation in *ptree_gene-translated2* nutzen zu können. Sie dient zudem als Import-Spezifikation des Moduls *ptree_gene-translation2*.

---

ptree_gene-translated2 =
enrich ptree+gene **with**
functions

toGene	:	ptree	→	gene	;
toGeneRec	:	ptreelist	→	genesymbollist	;
toPTree	:	gene	→	ptree	;
toPTreeRec	:	genesymbollist × nat	→	ptreelist	;

axioms

ToPTree :
  # g.list > 0
  ∧ correctGene(g)
  ∧ tl = toPTreeRec(g.list.rest, #cp(g.list) - 1)
  ∧ # tl > 0
→ toPTree(g) = node(g.list.first, inv(tl));

ToPTreeRec1 : toPTreeRec(gsl, 0) = [];

ToPTreeRec2 :
  n > 0
  ∧ tl3 = toPTreeRec(gsl, n - 1)
  ∧ # tl3 ≥ gs.arity
  ∧ tl3 = inv(tl1) + tl2
  ∧ # tl1 = gs.arity
→ toPTreeRec(gs + gsl, n) = tl2 + node(gs, tl1);

ToGene :
  correctPNode(node(gs, tl)) ∧ toGeneRec(tl) = gsl
→ toGene(node(gs, tl)) = makeGene(gs + gsl, 0);

ToGeneRec1 : toGeneRec([]) = [];

ToGeneRec2 :

correctPNode(node(gs, tl)) ∧ gsl = toGeneRec(tl1 + tl)
→ toGeneRec(node(gs, tl) + tl1) = gs + gsl;

end enrich

Generelle Beschreibung:
Diese Spezifikation enthält Funktionen und Axiome zur Beschreibung der gegenseitigen Transformation von Genen (*gene*) und Bäumen (*ptree*), die den Ausgangspunkt für die Umwaldungen von Genotype und Phenotype darstellen. Die SPezifikation stellt ebenso die Export-Spezifikation des Moduls *ptree_gene-translation2* dar, welches die Transformation implementiert. Die Funktionen *toGene* und *toPTree* stellen hierbei den Ausgangspunkt dar. Durch sie werden über die Axiome *ToPTree* und *ToGene* die rekursiven Funktionen zur Transformation aufgerufen.

Als Ausgangspunkt für die nachfolgenden Beschreibungen werden die Begriffe und Beschreibungen des Kapitels 1 (Einleitung) vorausgesetzt. Die Transformationen sind rekursiv definiert, der Algorithmus zur Umwandlung von *ptree* zu *gene* lehnt sich dabei stark an die in Prolog gehaltene Version aus [3] an, in der Gegenrichtung konnte das dort verwendete Verfahren nur teilweise übernommen werden.

Beschreibung der Transformation von *ptree* zu *gene*:

Es ist der gesamte Baum zu durchlaufen und aus den Baumknoten unter Einhaltung des zugrunde gelegten Ordnungsbegriffs das entsprechende Gen zu erzeugen. Da zwischen Gen und Baum eine ebenenweise Ordnungs-Beziehung besteht, muss der Baum auch ebenenweise abgearbeitet werden, um eine korrekte Reihenfolge im Gen zu erhalten. Die grundlegende Idee zur Realisierung dessen ist das Führen einer Liste *tl* von Baumknoten, die noch nicht betrachtet wurden.

Die Verfahrensweise ist rekursiv: zu Beginn enthält tl lediglich die Baumwurzel. In jedem Rekursionsschritt wird das erste Element aus tl entnommen und seine Söhne an tl angehangen. Mit dieser neuen Liste findet der rekursive Aufruf statt. Dadurch (Entnahme des ersten Elements und Anfügen seiner Söhne an tl) wird der Baum ebenenweise abgearbeitet. In jedem Rekursionsschritt wird die Summe aller Knoten der in tl befindlichen Teilbäume um 1 erniedrigt, wie durch das Lemma *nl-1* der Spezifikation *tree* bewiesen werden konnte. Dadurch terminiert der Algorithmus nach endlich vielen Schritten, was durch die Verifikation seiner Implementierung auch gezeigt werden konnte. Ist die Liste tl leer, so findet kein weiterer rekursiver Aufruf statt und es kann in der Rückwärtsphase ein direkter Aufbau des Gens erfolgen.

Die Rückkehr aus einem rekursiven Aufruf liefert ein Teil-Gen, welches das Ende des tatsächlichen Gens darstellt. Um dieses Gen mit dem aktuell betrachteten Baumknoten zu erweitern, wird dessen Inhalt von vorn an das Gen angefügt und das so erweiterte Teil-Gen wird zurück gegeben.

Die eben beschriebene Vorgehensweise wird durch genau 3 Axiome spezifiziert. *ToGene* stellt den Ausgangspunkt für die Rekursion dar und beschreibt das Ergebnis der Funktion *toGene*. Die Wurzel des übergebenen Baumes (*node(gs, tl)*) wird entnommen, wobei tl als Sohnliste der Wurzel das erste Argument für den Aufruf der rekursiven Funktion *toGeneRec* darstellt. Bei deren Rückkehr ergibt sich das gewünschte Gen aus gs und der zurück

gegebenen *genesymbollist*. Die Axiome *ToGeneRec1* und *ToGeneRec2* stellen die Spezifikation der rekursiven Funktion *toGeneRec* dar. *ToGeneRec1* beschreibt das Rekursionsende bei einer leeren Liste als Argument, *ToGeneRec2* ist für die rekursiven Aufrufe und die Erzeugung der korrekten Ergebnisliste verantwortlich.

Die gesamte Vorgehensweise lässt sich anschaulich an einem Beispiel verdeutlichen, welches in Abbildung 2.1 dargestellt ist:

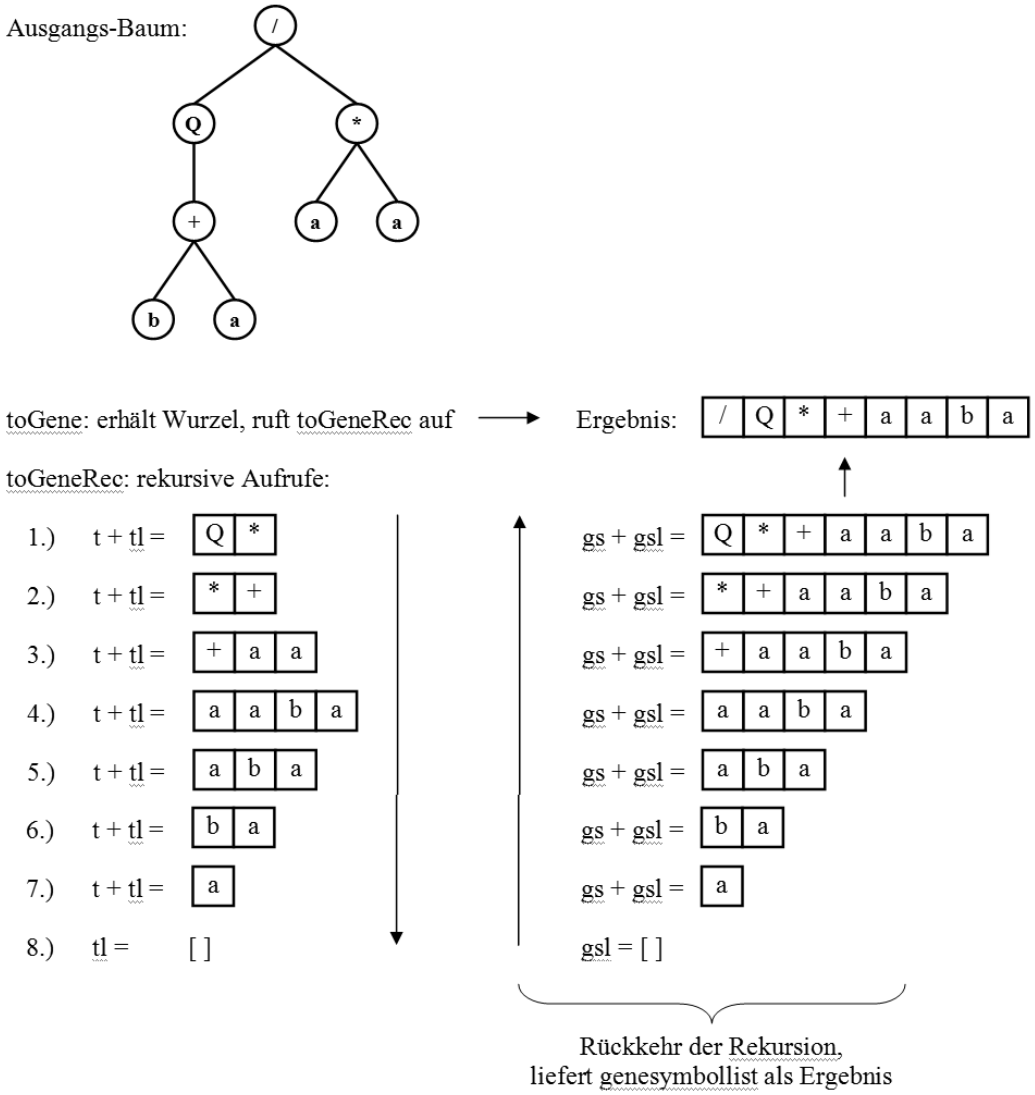


Abbildung 2.1: Beispiel zur Transformation *ptree* → *gene*

Beschreibung der Transformation von *gene* zu *ptree*:

Die ebenenweise Repräsentation zusammenhängender Symbole des Gens im Baum macht diese Richtung der Transformation etwas unhandlich. Zudem muss man wissen, bis zu welchem Symbol ein Gen transformiert wird, da i. Allg. nicht das ganze Gen in einen Baum überführt wird, sondern nur die ersten n Elemente. Die Länge des „kodierenden Teils“

(coding part, *CP*) gibt die Anzahl der Elemente an, welche mit der Anzahl der Knoten in den zu überführenden Baum übereinstimmen. Axiomatisch definiert wurde dies bereits durch die Funktion *#cp* der Spezifikation *genesymbollist*, sodass sie hier verwendet werden kann.

Es wird folgendermaßen vorgegangen: zunächst findet ein rekursiver Abstieg im zu transformierenden Gen statt. Dabei wird in jedem Rekursionsschritt der Liste des *gene* das vorderste Element entnommen, bis das Ende des kodierenden Teils des Gens erreicht ist. Die entfernten Elemente stellen genau die Symbole dar, welche durch Knoten im zu erstellenden Baum repräsentiert werden.

Bei der Rückkehr der rekursiven Aufrufe wird nun der Baum von unten (den Blättern) nach oben (der Wurzel) ebenenweise aufgebaut. Ebenenweise deswegen, weil der kodierende Teil des Gens über die rekursiven Aufrufe sequentiell abgearbeitet wurde und zusammenhängende Genteile jeweils eine Ebene im Baum repräsentieren. Jeder rekursive Aufruf gibt eine *ptreelist* zurück, deren Elemente Söhne darstellen, die auf oberen Baumebenen noch mit ihren Elternknoten verbunden werden müssen. Dabei gilt: je tiefer der rekursive Aufruf statt gefunden hat, umso eher ist das betreffende Gensymbol beim Rückkehr der Rekursion als Sohn einem Elternknoten hinzuzufügen. Darauf basiert der Aufbau der zurück gegebenen *ptreelist*: Knoten, die als erstes einem Vater hinzuzufügen sind, stehen ganz vorn an der Liste. Bei jeder Rückkehr aus einer Rekursion werden die ersten Elemente der *ptreelist* entnommen und dem aktuellen Symbol als Sohnknoten hinzugefügt. Die Anzahl der entnommenen Elemente entspricht dabei der Arität des aktuellen Symbols. Dabei muss die Liste der so entnommenen Sohnknoten noch invertiert werden, da sie von rechts (also in umgekehrter Ebenenrichtung) an die *ptreelist* angefügt wurden. An die Restliste ohne die entnommenen Sohnknoten wird der aktuell erzeugte (um die Söhne erweiterte) Baumknoten von rechts angefügt und das Ergebnis zurück gegeben. Auf diese Art kehren alle rekursiven Aufrufe zurück und wird die Wurzel erreicht, so entspricht die zurück gegebene *ptreelist* genau der Sohnliste der Wurzel in umgekehrter Reihenfolge.

Die Spezifikation des eben beschriebenen Algorithmus' findet über 3 Axiome statt. *ToPTree* stellt mit der Definition der Funktion *toPTree* den Ausgangspunkt dar. Das erste Element des Gens (die Wurzel des Baums) wird entnommen und mit der Länge *#cp* des kodierenden Gen-Teils (vermindert um 1, da das erste Element bereits betrachtet wurde) sowie mit dem Rest-Gen die Rekursion über die Funktion *toPTreeRec* gestartet. Bei deren Rückkehr erhält man eine *ptreelist*, welche genau der Sohnliste der Wurzel in umgekehrter Reihenfolge entspricht, womit die Baumwurzel erzeugt und als Ergebnis zurück geliefert werden kann.

Die Axiome *ToPTreeRec1* und *ToPTreeRec2* spezifizieren die Funktion *toPTreeRec*, welche den Baum nach der obigen Beschreibung rekursiv aufbaut. *toPTreeRec* übernimmt zwei Argumente. Das erste ist die noch zu betrachtende Restliste, das zweite stellt die Anzahl noch zu betrachtender Elemente des kodierenden Teils des Gens dar. Beträgt diese Anzahl 0, so ist das Ende der Rekursion erreicht, was durch *ToPTreeRec1* zum Ausdruck kommt. Das Axiom *ToPTreeRec2* beschreibt hingegen das Verhalten, wenn der kodierende Gen-Teil noch nicht komplett abgearbeitet wurde. In diesem Fall findet zunächst der rekursive Aufruf mit dem Rest-Gen und der verminderten Größe des kodierenden Teils statt. Dieser liefert eine Liste tl3 zurück. Ist die Länge dieser Liste größer oder gleich der Arität des abgetrennten Gensymbols gs (was aufgrund der Berechnung des kodierenden Teils auf jeden

Fall eintritt), so werden die ersten Elemente abgetrennt und in umgekehrter Reihenfolge als Sohnliste des aktuellen Knotens verwendet. Die Länge dieser Liste tl1 entspricht dabei der Arität von gs. Das Ergebnis des Aufrufs setzt sich zusammen aus der Restliste tl2, an die von rechts der aktuell erzeugte Baumknoten (bestehend aus gs und tl1) angefügt wird.

Die gesamte Vorgehensweise lässt sich anschaulich an einem Beispiel verdeutlichen, welches in Abbildung 2.2 dargestellt ist:

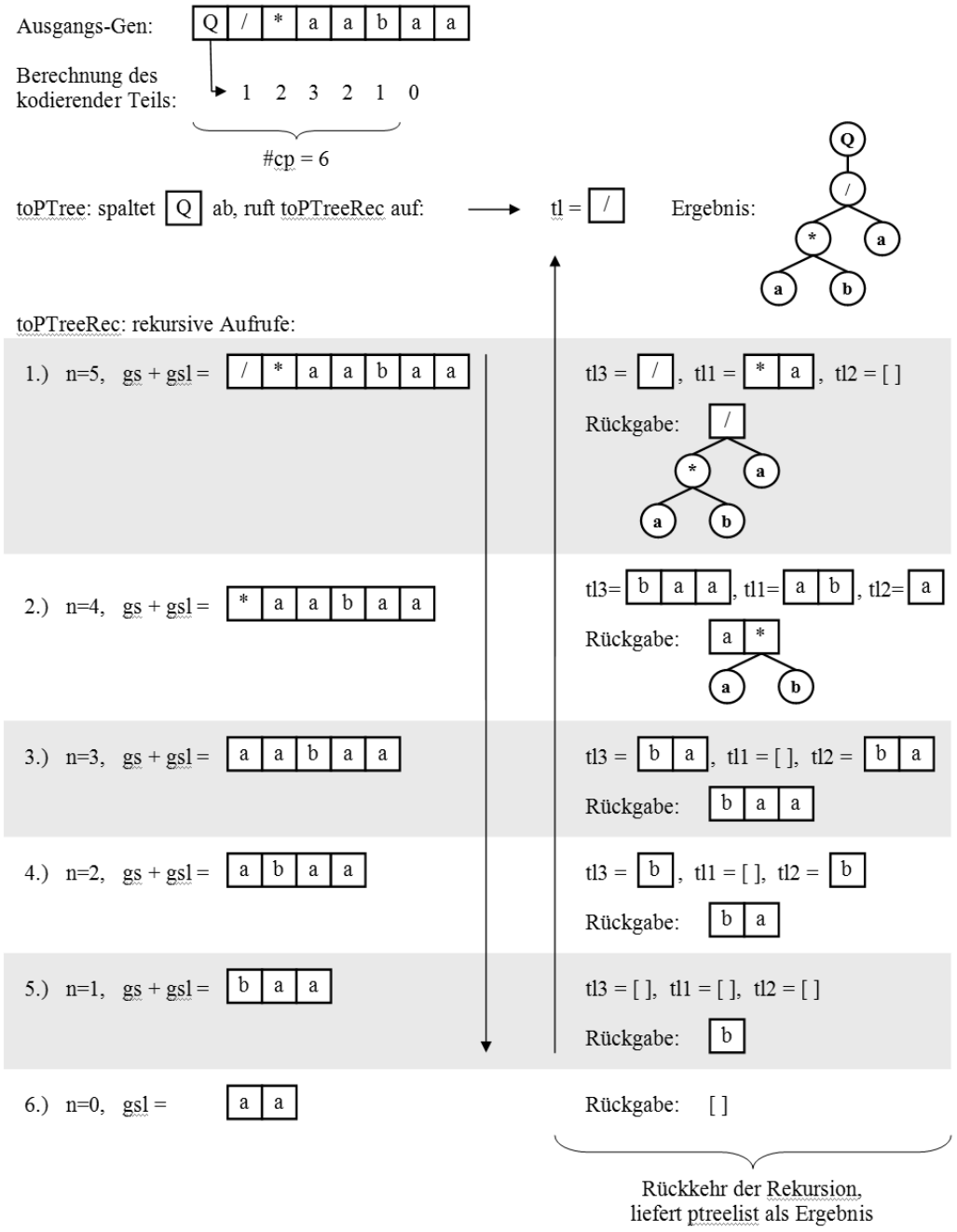


Abbildung 2.2: Beispiel zur Transformation *gene* → *ptree*

---

pheno+geno+transl2 = phenotype + genotype + ptree_gene-translated2

Generelle Beschreibung:
Dies stellt lediglich eine Vereinigung der Spezifikationen *phenotype*, *genotype* und *ptree_gene-translated2* dar, um die beiden Sorten *phenotype* und *genotype* bei ihrer gegenseitigen Transformation in *pheno_geno-translated2* unter Zuhilfenahme der in *ptree_gene-translated2* spezifizierten Funktionen nutzen zu können. Sie dient zudem als Import-Spezifikation des Moduls *pheno_geno-translation2*.

---

pheno_geno-translated2 =
enrich pheno+geno+transl2 **with**
functions

toGenotype	:	phenotype	→	genotype	;
toGenotypeRec	:	ptreelist	→	genelist	;
toPhenotype	:	genotype	→	phenotype	;
toPhenotypeRec	:	genelist	→	ptreelist	;

axioms

ToPheno : toPhenotype(gt) = makePT(toPhenotypeRec(gt.genes));

ToPhenoRec1 : toPhenotypeRec([]) = [];

ToPhenoRec2 : toPhenotypeRec(g + gl) = toPTree(g) + toPhenotypeRec(gl);

ToGeno : toGenotype(pt) = makeGT(toGenotypeRec(pt.ptrees));

ToGenoRec1 : toGenotypeRec([]) = [];

ToGenoRec2 : toGenotypeRec(t + tl) = toGene(t) + toGenotypeRec(tl);

end enrich

Generelle Beschreibung:
Durch diese Spezifikation wird die gegenseitige Transformation von Genotype und Phenotype beschrieben. Sie stellt die Export-Spezifikation des Moduls *pheno_geno-translation2* dar, welches die Transformation implementiert. Die Funktionen *toGenotype* und *toPhenotype* stellen hierbei den Ausgangspunkt dar. Durch sie werden über die Axiome *ToPheno* und *ToGeno* die rekursiven Funktionen zur Transformation aufgerufen.

toPhenotypeRec wandelt eine Liste von Genen (*genelist*) in eine Liste von Bäumen (*ptreelist*) um. Das Axiom *ToPhenoRec1* stellt hierbei das Rekursionsende dar, wenn die Liste leer ist, also komplett abgearbeitet wurde. Über das Axiom *ToPhenoRec2* wird die Rekursion definiert. Der erste Baum wird von der Liste abgetrennt und über die Funktion *toGene* der Spezifikation *ptree_gene-translated2* in ein Gen überführt. Dieses wird verkettet mit der *genelist*, die durch den rekursiven Aufruf für die Restliste erhalten wird.

toGenotypeRec wandelt eine Liste von Bäumen (*ptreelist*) in eine Liste von Genen (*genelist*) um. Das Axiom *ToGenoRec1* stellt hierbei das Rekursionsende dar, wenn die Liste leer ist,

also komplett abgearbeitet wurde. Über das Axiom *ToGenoRec2* wird die Rekursion definiert. Das erste Gen wird von der Liste abgetrennt und über die Funktion *toPTree* der Spezifikation *ptree_gene-translated2* in einen Baum überführt. Dieser wird verkettet mit der *ptreelist*, die durch den rekursiven Aufruf für die Restliste erhalten wird.

2.5 Module

---

ptree_gene-translation2 =
module
export ptree_gene-translated2
refinement
representation of operations

toGene#	implements	toGene;
toGeneRec#	implements	toGeneRec;
toPTree#	implements	toPTree;
toPTreeRec#	implements	toPTreeRec;

implementation
import ptree+gene

procedures dividePTreelistptreelist× nat : ptreelist × ptreelist;

declaration

```
toPTree#(g; var t)
begin
t := empty;
if # g.list > 0 ∧ correctGene(g) then
  var tl = [] in
  begin
    toPTreeRec#(g.list.rest, #cp(g.list) - 1; tl);
    if # tl > 0 then t := node(g.list.first, inv(tl))
  end
end;

toPTreeRec#(gsl, n; var tl)
begin
tl := [];
if # gsl > 0 ∧ n > 0 then
```

```
    var gs = gsl.first, tl3 = [] in
    begin
      toPTreeRec#(gsl.rest, n - 1; tl3);
      if # tl3 ≥ gs.arity then
        var tl1 = [], tl2 = [] in
        begin dividePTreelist(tl3, gs.arity; tl1, tl2); tl := tl2 + node(gs, tl1)end
    end
  end;


  dividePTreelist(tl, n; var tl1, tl2)
  begin
  tl1 := [];
  tl2 := tl;
  if # tl > 0 ∧ n > 1 then
    begin dividePTreelist(tl.rest, n - 1; tl1, tl2); tl1 := tl1 + tl.firstend
  else
    if # tl > 0 ∧ n = 1 then begin tl1 := tl.first '; tl2 := tl.restend
  end;


  toGene#(t; var g)
  begin
  g := makeGene([], 0);
  if is_node(t) ∧ correctPNode(t) then
    var gsl = [] in
    begin toGeneRec#(t.subtrees; gsl); g := makeGene(t.value + gsl, 0)end
  end;


  toGeneRec#(tl; var gsl)
  begin
  gsl := [];
  if tl ≠ [] then
    var t = tl.first in
    if is_node(t) ∧ correctPNode(t) then
      begin toGeneRec#(tl.rest + t.subtrees; gsl); gsl := t.value + gslend
  end
```

uniform restriction declarations:

```
toPTree#(g; var t)
begin
t := empty;
if # g.list > 0 ∧ correctGene(g) then
  var tl = [] in
  begin
    toPTreeRec#(g.list.rest, #cp(g.list) - 1; tl);
    if # tl > 0 then t := node(g.list.first, inv(tl))
```

```
  end
end;


toPTreeRec#(gsl, n; var tl)
begin
tl := [];
if # gsl > 0 ∧ n > 0 then
  var gs = gsl.first, tl3 = [] in
  begin
    toPTreeRec#(gsl.rest, n - 1; tl3);
    if # tl3 ≥ gs.arity then
      var tl1 = [], tl2 = [] in
      begin dividePTreelist(tl3, gs.arity; tl1, tl2); tl := tl2 + node(gs, tl1)end
  end
end;


dividePTreelist(tl, n; var tl1, tl2)
begin
tl1 := [];
tl2 := tl;
if # tl > 0 ∧ n > 1 then
  begin dividePTreelist(tl.rest, n - 1; tl1, tl2); tl1 := tl1 + tl.firstend
else
  if # tl > 0 ∧ n = 1 then begin tl1 := tl.first '; tl2 := tl.restend
end;


toGene#(t; var g)
begin
g := makeGene([], 0);
if is_node(t) ∧ correctPNode(t) then
  var gsl = [] in
  begin toGeneRec#(t.subtrees; gsl); g := makeGene(t.value + gsl, 0)end
end;


toGeneRec#(tl; var gsl)
begin
gsl := [];
if tl ≠ [] then
  var t = tl.first in
  if is_node(t) ∧ correctPNode(t) then
    begin toGeneRec#(tl.rest + t.subtrees; gsl); gsl := t.value + gslend
end
```

Generelle Beschreibung:

Durch dieses Modul werden die in der Export-Spezifikation *ptree_gene-translated2* definierten Funktionen implementiert. Es konnte verifiziert werden, dass alle Funktionen terminieren und die in der Export-Spezifikation festgelegten Axiome erfüllt werden. Daher soll hier nicht weiter darauf eingegangen werden, für weitere Informationen zu den Transformationen wird auf die Spezifikation *ptree_gene-translated2* verwiesen.

In diesem Modul wird die Prozedur *dividePTreelist* definiert, deren Funktionsweise nachfolgend kurz erläutert werden soll. *dividePTreelist* besitzt als Eingabeparameter eine *ptreelist* tl sowie eine Zahl *nat* n. Das Ziel besteht darin, tl in 2 Listen tl1 und tl2 zu zerlegen. Dabei sind zwei Kriterien zu erfüllen, wenn die Länge von tl mindestens n entspricht: 1.) die Länge von tl1 soll n sein, 2.) tl1 soll die invertierte Liste der n ersten Elemente von tl darstellen, also: tl = inv(tl1) + tl2. tl1 und tl2 werden als Ausgabeparameter zurück gegeben. Durch Definition geeigneter Lemmas konnte bewiesen werden, dass *dividePTreelist* terminiert und die o. a. Kriterien erfüllt. Dies stellt einen elementaren Bestandteil zur Verifikation der Korrektheit der Gesamtimplementierung dar, da *dividePTreelist* durch *toPTreeRec#* aufgerufen wird.

Kurz noch ein paar Worte zur Funktionsweise von *dividePTreelist*, welche rekursiv ist. n=1 stellt das Rekursionsende dar, hier finden keine rekursiven Aufrufe mehr statt. Das erste Element von tl wird zur Einerliste tl1, der Rest von tl wird tl2 zugewiesen und tl1 sowie tl2 werden zurück gegeben.
Ist n¿1, so findet ein rekursiver Aufruf von *dividePTreelist* mit der Restliste von tl und n-1 als die Anzahl noch abzutrennender Elemente statt. Bei Rückkehr der Funktion erhält man tl1 und tl2. tl2 wurde bereits korrekt erzeugt, tl1 soll hingegen die invertierte Anfangsliste von tl sein. Hierfür wird an tl1 das erste abgetrennte Element von tl hinten angehangen, wodurch sich über alle rekursiven Aufrufe die Invertierung ergibt. Das Ergebnis wird in tl1 gespeichert und tl1 sowie tl2 werden zurück gegeben.

```
pheno_geno-translation2 =
module
    export pheno_geno-translated2
    refinement
      representation of operations
          toPhenotype#      implements   toPhenotype;
          toPhenotypeRec#   implements   toPhenotypeRec;
          toGenotype#       implements   toGenotype;
          toGenotypeRec#    implements   toGenotypeRec;

      implementation
      import pheno+geno+transl2

      declaration

    toPhenotype#(gt; var pt)
    begin
    var tl = [] in begin toPhenotypeRec#(gt.genes; tl); pt := makePT(tl)end
```

```
    end;

    toPhenotypeRec#(gl; var tl)
    begin
    tl := [];
    if # gl > 0 then
      var tl1 = [] in
      begin toPhenotypeRec#(gl.rest; tl1); tl := toPTree(gl.first) + tl1end
    end;

    toGenotype#(pt; var gt)
    begin
    var gl = [] in begin toGenotypeRec#(pt.ptrees; gl); gt := makeGT(gl)end
    end;

    toGenotypeRec#(tl; var gl)
    begin
    gl := [];
    if # tl > 0 then
      var gl1 = [] in begin toGenotypeRec#(tl.rest; gl1); gl := toGene(tl.first) + gl1end
    end
```

uniform restriction declarations:

```
toPhenotype#(gt; var pt)
begin
var tl = [] in begin toPhenotypeRec#(gt.genes; tl); pt := makePT(tl)end
end;

toPhenotypeRec#(gl; var tl)
begin
tl := [];
if # gl > 0 then
  var tl1 = [] in
  begin toPhenotypeRec#(gl.rest; tl1); tl := toPTree(gl.first) + tl1end
end;

toGenotype#(pt; var gt)
begin
var gl = [] in begin toGenotypeRec#(pt.ptrees; gl); gt := makeGT(gl)end
end;

toGenotypeRec#(tl; var gl)
begin
gl := [];
```

```
if # tl > 0 then
  var gl1 = [] in begin toGenotypeRec#(tl.rest; gl1); gl := toGene(tl.first) + gl1end
end
```

Generelle Beschreibung:
Durch dieses Modul werden die in der Export-Spezifikation *pheno_geno-translated2* definierten Funktionen implementiert. Diese Implementierung stellt eine direkte Abbildung der dort festgelegten Axiome dar, sodass hier nicht weiter darauf eingegangen werden soll. Es ist anzumerken, dass verifiziert werden konnte, dass alle Axiome der Export-Spezifikation durch diese Implementierung erfüllt werden.

2.6 Projekt-Statistik

All theorems

- Proof obligations: 21
- Axioms: 167
- Theorems: 861
- Proof steps: 2707
- Interactions: 859
- Automation: 68.2 %

All modules

- Proof obligations: 21
- Axioms: 9
- Theorems: 8
- Proof steps: 536
- Interactions: 178
- Automation: 66.7 %

All specifications

- Axioms: 158
- Theorems: 853
- Proof steps: 2171
- Interactions: 681
- Automation: 68.6 %

Module ptree_gene-translation2

- Proof obligations: 11
- Axioms: 5
- Theorems: 8
- Proof steps: 322

- Interactions: 104
- Automation: 67.7 %

Module pheno_geno-translation2

- Proof obligations: 10
- Axioms: 4
- Theorems: 0
- Proof steps: 214
- Interactions: 74
- Automation: 65.4 %

Specification genesymbol-basic

- Axioms: 6
- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification genesymbol

- Axioms: 2
- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification genesymbollist-actualized

- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification genesymbollist

- Axioms: 12
- Theorems: 11
- Proof steps: 50
- Interactions: 29
- Automation: 42.0 %

Specification gene-basic

- Axioms: 6
- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification gene

- Axioms: 8
- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification tree-basic

- Axioms: 23
- Theorems: 2
- Proof steps: 3
- Interactions: 0
- Automation: 100.0 %

Specification tree

- Axioms: 26

- Theorems: 47
- Proof steps: 171
- Interactions: 65
- Automation: 61.9 %

Specification ptree-actualized

- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification ptree

- Axioms: 1
- Theorems: 2
- Proof steps: 8
- Interactions: 4
- Automation: 50.0 %

Specification ptree+gene

- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification phenotype

- Axioms: 5
- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification genotype

- Axioms: 5
- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification genelist-actualized

- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification genelist

- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification ptree_gene-translated2

- Axioms: 7
- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Specification pheno+geno+transl2

- Theorems: 0
- Proof steps: 0
- Interactions: 0

- Automation: 0.0 %

Specification pheno_geno-translated2

- Axioms: 6
- Theorems: 0
- Proof steps: 0
- Interactions: 0
- Automation: 0.0 %

Kapitel 3

Fazit

Mit der Bearbeitung dieses Projektes konnte zum einen Erfahrung in der Begriffswelt des Gene Expression Programming gesammelt werden, mit besonderem Fokus auf die Datenstrukturen Genotype und Phenotype sowie die gegenseitige Transformation beider. Zum anderen konnte die Bedienung und der Umgang mit dem Spezifikations- und Verifikationssystem KIV erlernt werden und es wurde ein Einblick in die Programmverifikation und deren Methoden erhalten.

KIV stellt sich als ein sehr mächtiges, aber auch nicht leicht zu erlernendes System dar, welches vor allem mit seinen Heuristiken den Beweisprozess zu unterstützen weiß. Dabei konnten auch Nachteile festgestellt werden, z. B. das Fehlen einer Baumstruktur in den Bibliotheks-Spezifikationen oder die Nicht-Nutzbarkeit der Bibliothek bei der Definition rekursiver Datenstrukturen.

Das gesamte Projekt war nur unter großem Zeitaufwand realisierbar, was nicht zuletzt auf die Spezifikation zahlreicher Hilfsdatenstrukturen zurück zu führen ist. Weiterhin war es schwierig geeignete Algorithmen zur gegenseitigen Transformation von Genotype und Phenotype zu erzeugen, welche vor allem auch mit KIV realisierbar sind. Trotz einiger Umwege denke ich die gestellten Aufgaben erfüllt zu haben, was ich nicht zuletzt mit einer Vielzahl bewiesener Hilfssätze und der erfolgreichen Modul-Verifikation begründe.

Literaturverzeichnis

[1] Candida Ferreira:
Gene Expression Programming
http://www.gene-expression-programming.com
letzter Zugriff: 11.02.07

[2] Candida Ferreira:
Gene Expression Programming in Problem Solving
WSC6 tutorial, 2001
http://www.gene-expression-programming.com/webpapers/GEPtutorial.pdf
letzter Zugriff: 11.02.07

[3] Uwe Petermann:
Gene Expression Programming - An Exercise in Prolog
January 24, 2007