

# Representation of Sets of Trees for Abstract Interpretation

Laurent MAUBORGNE

Thesis defended November 25, 1999 at École Polytechnique

Jury : Ahmed BOUAJJANI                      Patrick COUSOT (director)  
Jean GOUBAULT-LARRECQ                  Nicolas HALBWACHS  
Neil JONES (rapporteur, president)      Andreas PODELSKI  
David SCHMIDT (rapporteur)



## Remerciements

Les remerciements constituent une étape obligée dans une thèse. Ils peuvent être plus ou moins sincères, plus ou moins conventionnels. Pour ma part, je suis très heureux d'utiliser cette opportunité, car il y a vraiment des personnes qui ont beaucoup compté durant cette thèse, et qui ont grandement contribué à son élaboration dans les meilleures conditions.

Valérie a toujours cru en moi et son soutien m'a beaucoup apporté. Je ne sais pas si sans elle cette thèse serait terminée. Je lui dédie cet ouvrage.

Patrick Cousot a été mon directeur de thèse. Il a accepté d'encadrer mon travail en me laissant une très grande liberté d'organisation et de recherche. J'ai beaucoup apprécié les discussions que nous avons eues sur mon travail. Mon seul regret sera de ne pas avoir assez suivi ses conseils au début de ma thèse.

Neil Jones et David Schmidt ont accepté de lire ma thèse et d'en être les rapporteurs. J'ai récemment relu ma thèse, et je sais que leur travail n'a pas été des plus faciles. Je les remercie pour leur gentillesse. David Schmidt, en particulier, m'a beaucoup encouragé. Neil Jones a accepté d'être président du jury, malgré la charge supplémentaire que cela représente. Je suis honoré qu'ils aient accepté de faire un si long voyage pour assister à ma soutenance.

Jean Goubault-Larrecq, Nicolas Halbwachs, Andreas Podelski et Ahmed Bouajjani ont accepté d'être membres de mon jury. J'ai la chance qu'ils aient tous bien voulu venir pour ma soutenance, malgré tous les obstacles d'agenda. Je remercie particulièrement Jean qui, croyant avoir été désigné rapporteur, m'a fait part de commentaires précieux qui ont beaucoup contribué à améliorer cette thèse. Dommage que le quiproquo n'ait pas duré plus longtemps...

Merci aux personnes du laboratoire qui m'ont accueilli et soutenu. Louis Granboulan a relu une des premières versions de ma thèse. Joëlle Isnard m'a bien aidé dans l'organisation de mes déplacements et la réception du jury.

Enfin, je tiens à remercier tous ceux qui m'ont soutenu durant ces années, qui m'ont encouragé sans jamais me critiquer, ou qui m'ont aidé à préparer ma soutenance. J'espère qu'ils se reconnaîtront.



# Résumé

L'interprétation abstraite est un cadre très général permettant l'utilisation sûre d'approximations d'un point de vue sémantique. Cette théorie est utilisée principalement dans l'analyse de programmes et permet la conception de programmes calculant automatiquement les propriétés de programmes. Ces analyseurs s'appuient sur des structures de données permettant de représenter ces propriétés. Ces structures de données sont des objets fondamentaux de l'analyse, car elles déterminent la précision et la complexité de celle-ci. Les structures de données classiques peuvent être peu adaptées à cet usage, car elles ne sont pas conçues pour tirer avantage des possibilités d'approximation. C'est pourquoi différentes représentations ont été développées spécifiquement pour l'interprétation abstraite. Par exemple, dans le cas des ensembles d'entiers, on peut utiliser une représentation basée sur l'utilisation des polyèdres convexes, ou encore sur l'utilisation de congruences.

Les arbres sont des objets fondamentaux en informatique : ils sont présents dès que l'on considère des structures non linéaires. Les structures classiques pour représenter les ensembles d'arbres sont les automates d'arbres. On en utilise parfois des variantes, comme les grammaires d'arbres, mais ces variantes sont souvent moins efficaces à cause de l'introduction de noms de variables. Les automates d'arbres ont une complexité élevée, surtout les automates d'arbres infinis, et les ensembles d'arbres qu'ils permettent de représenter sont limités : ils n'autorisent pas la représentation d'ensembles de la forme  $f(a^n, b^n, c^n)$ ,  $n \in \mathbb{N}$  qui présentent des relations entre les sous-arbres.

Le résultat principal de cette thèse est une nouvelle représentation des ensembles d'arbres plus expressives que les automates d'arbres, adaptée aux techniques d'approximation et mettant en œuvre un principe de partage permettant une meilleure compacité et une meilleure efficacité des algorithmes associés. Cette représentation, les *schémas d'arbres*, permet de décrire des ensembles contenant des arbres infinis et avec une notion de relation entre les sous-arbres. Ce résultat est obtenu par extraction d'une structure, le *squelette* du schémas, qui exprime le partage préfixe maximum de l'ensemble. Cette structure est ensuite enrichie de liens représentés par des relations. Cela permet donc d'extraire l'aspect relationnel des ensembles d'arbres.

Puisque les ensembles représentés peuvent être infinis, et les arbres eux-mêmes

infinis, il a fallu représenter aussi des relations infinies. Ces relations sont très importantes dans les schémas d'arbres : ce sont elles qui déterminent leur pouvoir expressif, et ce sont aussi les facteurs prédominants de leur complexité. La façon la plus efficace connue de représenter des relations finies est l'utilisation de BDDs. Les représentations de relations infinies développées dans cette thèse sont des extensions des BDDs avec un pouvoir expressif comparable aux ensembles de mots infinis de Büchi. Une nouvelle classe de relations infinies permettant une représentation efficace a été définie. Elle permet l'approximation précise d'un grand nombre de relations infinies. Elle permet entre autres l'expression d'ensembles équitables.

La séparation entre l'aspect structurel et l'aspect relationnel des ensembles d'arbres a permis d'envisager l'introduction de nouvelles contraintes dans les schémas d'arbres. Ces contraintes sont basées sur l'utilisation de compteurs. Les compteurs peuvent être utilisés pour déterminer automatiquement quelles boucles grossissent et de quelle façon elles grossissent à l'intérieur d'une séquence de schémas d'arbres. Cette information est utile pour concevoir des outils d'approximation supplémentaires.

Les schémas d'arbres sont donc de nouvelles structures de données disponibles en interprétation abstraite qui devraient permettre des analyses plus fines et plus rapides.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivations . . . . .	1
1.1.1	Abstract Interpretation . . . . .	1
1.1.2	Representation of Sets of Trees . . . . .	1
1.2	Goal of the Thesis . . . . .	2
1.3	Guiding Ideas . . . . .	2
1.3.1	High Expressive Power for the Representation . . . . .	2
1.3.2	Uniqueness . . . . .	3
1.3.3	Formal Definitions . . . . .	3
1.4	Overview of the Thesis . . . . .	3
<b>2</b>	<b>Preliminary Definitions</b>	<b>5</b>
2.1	Basic Notations . . . . .	5
2.1.1	Functions . . . . .	5
2.1.2	Orderings . . . . .	5
2.1.3	Equivalences . . . . .	6
2.2	Words . . . . .	6
2.2.1	Orderings on Words . . . . .	7
2.2.2	Infinite Words . . . . .	7
2.2.3	Automata on Words . . . . .	7
2.3	Trees . . . . .	8
2.3.1	Arity . . . . .	8
2.3.2	Subtrees . . . . .	8
2.3.3	Infinite Trees . . . . .	9
2.3.4	Substitutions in Trees . . . . .	9
2.3.5	Recursive Functions on Trees . . . . .	10
2.3.6	Representation of a Tree . . . . .	10
2.4	Graphs . . . . .	11
2.4.1	Multi-edge Graphs . . . . .	12
2.4.2	Walking Through a Graph . . . . .	12
2.5	Relations . . . . .	12
2.5.1	Projection . . . . .	13
2.5.2	Support Set of a Relation . . . . .	13

2.5.3	Boolean Operations on Relations . . . . .	13
2.5.4	Projection According to a Value . . . . .	14
2.5.5	Independent Decomposition . . . . .	14
2.6	Abstract Interpretation . . . . .	14
2.6.1	Soundness and Galois Connections . . . . .	14
2.6.2	Fixpoints and Iteration Sequences . . . . .	15
2.6.3	Dynamic Approximation of Fixpoints . . . . .	16
<b>3</b>	<b>Sharing Regular Trees</b>	<b>17</b>
3.1	Classic Sharing . . . . .	17
3.1.1	General Framework . . . . .	17
3.1.2	Structures for the Trees . . . . .	18
3.1.3	Finite Trees . . . . .	18
3.2	Sharing Strongly Connected Graphs . . . . .	19
3.2.1	Maximal Sharing Strongly Connected Graphs . . . . .	19
3.2.2	Keys of Strongly Connected Graphs . . . . .	22
3.3	Regular Trees . . . . .	24
3.3.1	Informal Presentation . . . . .	24
3.3.2	Algorithm . . . . .	24
3.3.3	More Remarks to Understand the Algorithm . . . . .	27
3.3.4	Proof of the Algorithm . . . . .	30
3.3.5	Example . . . . .	32
3.4	Algorithms on Shared Regular Trees . . . . .	35
3.4.1	Easy Algorithms . . . . .	35
3.4.2	Generic Memoizing Algorithms . . . . .	36
3.4.3	Recursive Construction . . . . .	39
3.4.4	Complexity Issues . . . . .	40
<b>4</b>	<b>Representation of Relations</b>	<b>43</b>
4.1	Definitions and Common Structures . . . . .	44
4.1.1	Equivalent Entries . . . . .	44
4.1.2	Isomorphic Relations . . . . .	46
4.1.3	Regular Relations . . . . .	47
4.2	Decision Diagrams . . . . .	50
4.2.1	MDD for Finite Relations . . . . .	51
4.2.2	Regular Infinite Relations . . . . .	54
4.2.3	$\omega$ -deterministic Relations . . . . .	60
4.2.4	MDD with Large Sets of Choice . . . . .	63
4.3	Algorithms on $\omega$ -deterministic Relations . . . . .	64
4.3.1	Canonicity of the representation . . . . .	64
4.3.2	Entry Names . . . . .	73
4.3.3	Basic Algorithms . . . . .	75
4.3.4	Intersection . . . . .	77



4.3.5	Approximation . . . . .	79
4.4	Other Representations . . . . .	83
4.4.1	A Set of Vectors is a Set of Trees . . . . .	83
4.4.2	Exact Relations . . . . .	83
4.5	Conclusion . . . . .	84
<b>5</b>	<b>Tree Schemata</b>	<b>85</b>
5.1	Different Approaches . . . . .	85
5.1.1	Representations Based on Automata . . . . .	85
5.1.2	Representations Based on Expressions . . . . .	87
5.1.3	Tree Schemata . . . . .	88
5.2	The Skeleton . . . . .	88
5.2.1	A Set of Trees is a Tree . . . . .	88
5.2.2	Definition of the Set of Tree Schema Skeletons . . . . .	91
5.2.3	The Choice Nodes . . . . .	92
5.3	The links . . . . .	93
5.3.1	The Choice Space . . . . .	93
5.3.2	Links as Relations . . . . .	94
5.3.3	Links are Local . . . . .	94
5.3.4	The Names of the Links . . . . .	96
5.3.5	Sharing . . . . .	96
5.3.6	Set Represented by a Tree Schema . . . . .	98
5.3.7	Restrictions on the Links . . . . .	101
5.4	The Set of Tree Schemata, $\mathcal{TS}(F)$ . . . . .	106
5.4.1	Definition . . . . .	106
5.4.2	Description of the Sets of Trees Representable by Tree Schemata . . . . .	107
5.5	Discussion . . . . .	109
5.5.1	Expressive Power . . . . .	109
5.5.2	Strong Determinism . . . . .	110
5.5.3	Sources of Non-Uniqueness . . . . .	110
<b>6</b>	<b>Algorithms on Tree Schemata</b>	<b>113</b>
6.1	Auxiliary Informations on the Links . . . . .	113
6.2	Inclusion . . . . .	114
6.2.1	Inclusion of Skeletons . . . . .	114
6.2.2	Inclusion of Tree Schemata . . . . .	116
6.3	Intersection . . . . .	119
6.3.1	Intersection of Skeletons . . . . .	119
6.3.2	Intersection of Tree Schemata . . . . .	121
6.4	Union . . . . .	125
6.4.1	Union of Skeletons . . . . .	125
6.4.2	Union of Tree Schemata . . . . .	130

6.5	Projection . . . . .	130
6.5.1	Definition . . . . .	130
6.5.2	Projection of Skeletons . . . . .	131
6.5.3	Projection of Tree Schemata . . . . .	131
6.6	Meta Expressions . . . . .	133
6.7	Discussion . . . . .	134
<b>7</b>	<b>Tree Schemata with Counters</b>	<b>135</b>
7.1	New Entries for the Links . . . . .	135
7.1.1	Tree Schemata on Labels and Variables . . . . .	136
7.1.2	Many Variables in a Link . . . . .	136
7.1.3	Variables on Infinite Domains . . . . .	137
7.2	Interaction between Counters and Skeletons . . . . .	137
7.2.1	Choice Node Counters . . . . .	137
7.2.2	Labels as Abstract Operators . . . . .	140
7.2.3	Counters as Labels . . . . .	140
7.3	Finding Cycles . . . . .	141
7.3.1	Deciding When to Look for a Cycle . . . . .	141
7.3.2	Finding or Building a New Cycle . . . . .	142
7.3.3	Maximal Folding . . . . .	143
7.4	Modified Algorithms . . . . .	143
7.4.1	Global Information and Forgetting . . . . .	144
7.4.2	Inclusion and Loops . . . . .	144
7.4.3	Intersection and Counter Functions Merging . . . . .	145
7.5	Conclusion . . . . .	147
<b>8</b>	<b>Tree Based Abstract Interpretations</b>	<b>149</b>
8.1	Model of Logic Programs . . . . .	149
8.1.1	Syntax and Collecting Semantics . . . . .	149
8.1.2	Approximation of the Model . . . . .	150
8.1.3	Comparison with Other Approaches . . . . .	153
8.2	Sets of Traces . . . . .	153
8.2.1	Flowchart Semantics . . . . .	154
8.2.2	Big-Step Semantics . . . . .	155
8.2.3	Parallel Programs: Fairness . . . . .	157
8.3	Other Applications . . . . .	159
8.3.1	Model Checking . . . . .	159
8.3.2	Closure Analysis . . . . .	160
<b>9</b>	<b>Conclusion</b>	<b>161</b>
9.1	Main Results . . . . .	161
9.2	Future Work . . . . .	162
9.2.1	Practical Usefulness . . . . .	162

9.2.2	Negation . . . . .	162
9.2.3	Extensions . . . . .	162
<b>A</b>	<b>List of Symbols and Conventions</b>	<b>163</b>
A.1	Widely Used Mathematical Symbols . . . . .	163
A.2	Conventions for Other Mathematical Symbols . . . . .	163
A.3	Sets and Operators Defined in the Thesis . . . . .	164
A.3.1	Relations . . . . .	164
A.3.2	Tree Schemata . . . . .	165
<b>B</b>	<b>Algorithms</b>	<b>167</b>
B.1	Notations . . . . .	167
B.2	List of Algorithms . . . . .	168



# Chapter 1

## Introduction

### 1.1 Motivations

#### 1.1.1 Abstract Interpretation

Abstract interpretation [CC77, Cou78, CC92b] is a formalized framework designed to deal with approximations, specially useful for the static analysis of programs. This framework is very general and can be applied to a great variety of problems. In fact, a lot of works have been achieved, either based on abstract interpretation or trying to improve different parts of the framework. Because abstract interpretation is so general, there have been theoretical advances and researches to improve the possible applications of abstract interpretation.

When we try to apply abstract interpretation techniques, we must consider the objects manipulated during the process. This question is crucial to the implementation because it determines the accuracy and the complexity of the computation. But in many cases, the objects that are available from classical computer science may be inadequate, because they don't take advantage of the possibility of approximation. Thus some specific objects and their manipulation have been defined to be used in abstract interpretation. Usually, they cannot represent any possible object but they come with approximation mechanisms. If we consider for example sets of numbers or tuples of numbers, Patrick Cousot and Nicolas Halbwachs presented in [CH78] a representation based on convex polyhedra. Philippe Granger proposed the use of congruences in [Gra89].

#### 1.1.2 Representation of Sets of Trees

Finite and infinite trees are present in a way or another in most parts of computer science. In [Knu97], Donald E. Knuth describes the trees as “the most important nonlinear structures that arise in computer algorithms”. They can be seen as the underlying structure of many different kind of objects such as words, graphs—provided we can distinguish a node [Eng94, EV94]—, and of course non linear

data structures. In program semantics, they appear as control flow graphs or traces of execution. Even a set of trees can be seen as a tree.

Alas, representing sets of trees is difficult. In the case of finite trees, automata techniques can be implemented but they lack some natural approximation method. When it comes to infinite trees—a necessary step to represent graphs, for example—we can only find theoretical description of automata, but no realistic implementation is available. When someone needs sets of trees in an abstract interpretation, he will have to restrain his analysis to linear structures, such as in [Deu92], or to very crude approximations losing any relation between different parts of the trees as in [HJ90, Hei92].

## 1.2 Goal of the Thesis

The goal of this thesis is to provide a set of tools to manipulate sets of possibly infinite trees in abstract interpretation. The user of this set of tools should only care about basic operations over sets of trees, such as intersection, projection, etc. The set of tools would then take care of computing those operations using an efficient internal representation of sets of trees.

As representing sets of possibly infinite trees is a difficult task, some operations on sets of trees will require the use of automatic approximation. In order to make things easier for the user, operations of explicit approximation will be provided, as well as algorithms to decide whether there is an approximation that seems to suit the current iteration on the set of trees. In order to give the possibility to trade between accuracy and complexity options to force more automatic approximations should be available.

## 1.3 Guiding Ideas

In order to fulfill our objectives, the representation of sets of trees is to combine two main qualities: efficiency and expressive power.

### 1.3.1 High Expressive Power for the Representation

All experiments in abstract interpretation indicate that the later we approximate the more accurate the final result. In order to approximate as late as possible, automatic approximation should be reduced to a minimum. It means that for every representation used in an abstract interpretation, we must try to be able to represent the largest set of object possible. In this thesis, we try in every area of the representation to go as far as possible in the expressive power without, hopefully, jeopardizing the efficiency of the representation.

### 1.3.2 Uniqueness

The efficiency of a representation lies in its compactness and in the complexity of the algorithms using the objects. This is a matter of balance, so we cannot use algorithms from data compression for example, because, although the compactness of compressed objects is very good, their manipulation would be very difficult. The notion of uniqueness, on the other hand, seems to deal with every aspect of the efficiency of the representation.

The representation of an object is said to be unique if no other representation represents this object in the same frame. A typical example of non-uniqueness is the use of variable names: they produce at least as many possible representation for a given object as the number of possible variable names. Striving after the uniqueness of a representation is moving towards a better compactness of this representation. Actually, if it is not unique, then the representation contains some redundancy.

The problem with uniqueness is that it may require a lot of computation to find a kind of normal form (the minimal state representation when possible). Our argument is that, in most case, the gain in most algorithms makes up for this overhead. It is particularly true if we need equality testing or emptiness testing.

### 1.3.3 Formal Definitions

Because it deals with the essence of approximation from a very theoretical point of view, abstract interpretation is a mathematical model. It is very well formalized, and this formal presentation of the model allows a very high confidence in its correctness and a genericity of the results. The drawback—or the advantage—is that any extension or application of abstract interpretation should be very formal too.

In our case, a formal definition is an advantage, because we have to deal with infinite structures and infinite behavior. Experience shows that in this case, informal presentation and intuitive proofs can lead to wrong results.

## 1.4 Overview of the Thesis

After this presentation, the thesis starts with a recall of basic mathematical background and the associated notations. The following fields are presented: words, trees, graphs, relations and abstract interpretation. If the reader is already familiar with one of these fields, he can skip it and use the summary of the notations in appendix A when necessary.

Chapter 3 gives the basic algorithms on regular trees that are extended later to insure good sharing properties.

Chapter 4 studies the efficient representation of relations. It gives new classes of infinite relations that can be represented by decision diagrams. The expressive

power of these relations is essential in the representation of sets of trees.

Chapter 5 describes the representation at the heart of the thesis, tree schemata. Tree schemata and their use of relations are formally described. In the next chapter, some algorithms on tree schemata are presented as a guideline to use them.

Chapter 7 presents some extensions to basic tree schemata. These extensions take advantage of the way relations are used in tree schemata. The most useful extension is the incorporation of loop counters which allow more precise approximation by numerical analysis.

Finally, chapter 8 presents briefly some possible applications of tree schemata and the last chapter concludes.



# Chapter 2

## Preliminary Definitions

In this chapter, we introduce some basic definitions over mathematical objects that will be used in the thesis. These presentations are quite succinct because most of them are very well-known in computer science. They are mainly given to set the notations over these objects. Most of these notations are summarized in appendix A.

### 2.1 Basic Notations

The set of natural numbers will be denoted  $\mathbb{N}$ . If  $n \in \mathbb{N}$ , then

$$[n] \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid 0 \leq i < n\}$$

If, moreover,  $m \in \mathbb{N}$ ,

$$[n, m[ \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid n \leq i < m\}$$

The cartesian product of two sets  $E$  and  $F$  is denoted  $E \times F$ . The cartesian product of a set family  $(E_i)_{i \in I}$  is denoted  $\bigotimes_{i \in I} E_i$ .

#### 2.1.1 Functions

Let  $E$  and  $F$  be two sets, and  $f \subset E \times F$ . The set  $f$  is said to be a function, and we write  $f : E \rightarrow F$ , if and only if  $\forall a \in E$ , there is at most one  $b$  in  $F$  such that  $\langle a, b \rangle \in f$ . We write  $f(a) = b$ . We say that  $f$  is a function from  $E$  to  $F$ . The domain of  $f$  is denoted  $\text{dom}(f) \stackrel{\text{def}}{=} \{a \in E \mid \exists b \in F, f(a) = b\}$ . The image of  $f$  is denoted  $\text{im}(f) \stackrel{\text{def}}{=} \{b \in F \mid \exists a \in E, f(a) = b\}$ .

If  $f : E \rightarrow E$ , the element  $x \in E$  is called a *fixpoint* of  $f$  if  $f(x) = x$ .

#### 2.1.2 Orderings

Let  $E$  be a set. A *partial ordering*  $\leq$  on  $E$  is a subset of  $E \times E$  such that  $\forall a, b, c \in E$ ,  $\langle a, a \rangle \in \leq$  (reflexivity),  $(\langle a, b \rangle \in \leq \text{ and } \langle b, a \rangle \in \leq) \Rightarrow a = b$

(antisymmetry), and  $(\langle a, b \rangle \in \leq \text{ and } \langle b, c \rangle \in \leq) \Rightarrow \langle a, c \rangle \in \leq$  (transitivity). When using partial orderings, we use the infix notation:  $a \leq b$  for  $\langle a, b \rangle \in \leq$ .

Every partial ordering  $\leq$  corresponds to a strict partial ordering denoted  $<$  defined as  $< \stackrel{\text{def}}{=} \{ \langle a, b \rangle \in \leq \mid a \neq b \}$ . A *total ordering*  $\leq$  on  $E$  is a partial ordering on  $E$  such that  $\forall a, b \in E, a \leq b$  or  $b \leq a$ .

Let  $E$  be a set partially ordered by  $\leq$ . Let  $F \subset E$ . The element  $a \in E$  is an *upper bound* of  $F$  if  $\forall b \in F, b \leq a$ . It is said to be the *least upper bound* of  $F$ , denoted  $\text{lub}_{\leq}(F)$ , if it is an upper bound of  $F$ , and for all  $b$  that is also an upper bound of  $F, a \leq b$ . Note that  $\text{lub}_{\leq}(F)$  is not always defined, but when it exists, it is unique. We define dually the greatest lower bound of  $F$  ( $\text{glb}_{\leq}(F)$ ).

A *complete lattice* is a set  $E$  and a partial ordering  $\leq$  on  $E$  such that for all  $F \subset E, \text{lub}_{\leq}(F)$  and  $\text{glb}_{\leq}(F)$  exist.

### 2.1.3 Equivalences

Let  $E$  be a set. An *equivalence*  $\equiv$  on  $E$  is a subset of  $E \times E$  such that  $\forall a, b, c \in E, \langle a, a \rangle \in \equiv$  (reflexivity),  $\langle a, b \rangle \in \equiv \Leftrightarrow \langle b, a \rangle \in \equiv$  (symmetry), and  $(\langle a, b \rangle \in \equiv \text{ and } \langle b, c \rangle \in \equiv) \Rightarrow \langle a, c \rangle \in \equiv$  (transitivity). As for partial orderings, we use the infix notation:  $\langle a, b \rangle \in \equiv$  is denoted  $a \equiv b$ .

Equivalences define equivalence classes. Let  $\equiv$  be an equivalence on  $E$ . The equivalence classes of  $\equiv$  are the sets in  $\{ \{ b \mid a \equiv b \} \mid a \in E \}$ . Note that equivalence classes cannot be empty because of the reflexivity property.

## 2.2 Words

Let  $A$  be a set of letters. Then  $A^*$  is the *set of finite words* made of letters from  $A$ , and  $A^+$  the set of non-empty words over  $A$ . The empty word will be written  $\varepsilon$ . The word consisting of a single letter  $a$  of  $A$  is written  $a$ . If  $u$  and  $v$  are words over  $A$ , then their concatenation will be denoted  $u.v$  or  $uv$ .  $|u|$  is the length of  $u$ .  $|\varepsilon| = 0$ .

If  $(u_i)_{i \in I}$  is a family of words with total ordering  $<_I$  on  $I$ , their concatenation following the order of their indexes is written  $\bigodot_{i \in I}^{<_I} u_i$ . If  $I = \emptyset$ , then this is  $\varepsilon$ . This operation can be extended to families of sets of words in the following way:  $L_1 L_2 \stackrel{\text{def}}{=} \{ u_1 u_2 \mid u_1 \in L_1, u_2 \in L_2 \}$

If  $L \subset A^*$ , we can define the set of all finite concatenations of elements of  $L$ , in any orders. This set is written  $L^*$ . Formally,

$$L^* \stackrel{\text{def}}{=} \bigcup_{i \in \mathbb{N}} \left\{ \bigodot_{j < i}^{<} u_j \mid \forall j < i, u_j \in L \right\}$$

### 2.2.1 Orderings on Words

There is a canonical ordering on  $A^*$ , the *prefix* relation  $\prec$  defined as:

$$\forall u, v \in A^*, u \prec v \stackrel{\text{def}}{\Leftrightarrow} \exists u_1 \in A^+, v = uu_1$$

A subset  $M$  of  $A^*$  is said to be *prefix closed* if and only if  $\forall u \in M, \forall v \in A^*, v \prec u \Rightarrow v \in M$ .

An ordering  $<_A$  on  $A$  can be extended to  $A^*$  by means of the *alphabetic ordering*,  $\prec_A^*$  defined as:

$$\forall u, v \in A^*, u \prec_A^* v \Leftrightarrow u \prec v \vee \exists w \in A^*, a, b \in A, wa \preceq u \wedge wb \preceq v \wedge a <_A b$$

If  $<_A$  is total, then  $\prec_A^*$  is total.

### 2.2.2 Infinite Words

The set of infinite words made of letters from  $A$  is written  $A^\omega$ . An infinite word can be seen as a function from  $\mathbb{N}$  to  $A$ , so we write  $u(i)$  for the  $i^{\text{th}}$  letter of  $u$ .  $A^\infty \stackrel{\text{def}}{=} A^* \cup A^\omega$ . A subset of  $A^\infty$  is also called a language. The prefix ordering can be extended to  $A^\infty$  by  $\forall u \in A^*, \forall v \in A^\infty, u \prec v \Leftrightarrow \exists u_1 \in A^\infty, v = uu_1$ . Using this ordering, we can define the infinite closure of sets of finite words (the same as the adherence of [BN80]). The infinite closure of the set of words  $L \subset A^*$ , written  $\vec{L}$  is defined by:

$$\vec{L} \stackrel{\text{def}}{=} \{x \in A^\omega \mid \{u \in L \mid u \prec x\} \text{ is infinite}\}$$

If  $L \subset A^*$ , we define  $L^\omega$  as  $\vec{L}^*$ . An infinite word  $w$  is said to be *ultimately periodic* if there is a finite word  $u$  and a finite word  $v$  such that  $w = u.v^\omega$ .

### 2.2.3 Automata on Words

Let  $A$  be a set of letters. We can define many different automata to recognize sets of finite or infinite words of  $A$ . All these automata have a common structure, which we call an automaton in general, that is a set of states,  $Q$  and a transition set  $E \subset Q \times A \times Q$ . An automaton is said to be finite if  $E$  is finite. Two transitions  $(p, a, q)$  and  $(p', a', q')$  are said to be consecutive if  $q = p'$ . A path is a sequence of consecutive transitions. The label of a path is the sequence of the letters of its transitions. The first state of the first transition of a path is called its origin. If the path is finite, the last state of its last transition is called its end. If the path  $p$  is infinite, we write  $In(p)$  the set of states that appear in an infinite number of transitions of  $p$ . This set is called the set of infinitely repeated states of  $p$ .

A Büchi automaton [Büc60] is a finite automaton plus a set of initial states  $I$  and a set of final states  $F$ . We write  $\mathcal{A} = (Q, E, I, F)$ .  $\mathcal{A}$  recognizes any infinite

word that is the label of a path of  $\mathcal{A}$  with origin in  $I$  and such that there is a state of  $F$  in the set of infinitely repeated states of the path. We write  $L^\omega(\mathcal{A})$  this set of infinite words. A Büchi automaton can be used to recognize a set of finite words:  $L^*(\mathcal{A})$  is the set of words labeling path of  $\mathcal{A}$  with origin in  $I$  and end in  $F$ .

## 2.3 Trees

Let  $F$  be a set of labels. A tree  $t$  labeled by  $F$  is a function of  $pos(t) \rightarrow F$  such that  $pos(t)$  is a prefix closed non-empty subset of  $\mathbb{N}^*$ . The *domain* of  $t$ ,  $pos(t)$ , defines the shape of the tree. The elements of  $pos(t)$  can also be called the *paths* of the tree.  $t(\varepsilon)$  is called the *root* of  $t$ .

### 2.3.1 Arity

The set of labels  $F$  is said *ranked* if it is associated with an arity function  $Ar_F : F \rightarrow \mathbb{N}$ . A tree  $t$  labeled by  $F$  is said to *respect the arity* of  $F$  whenever the following property holds:

$$\forall p \in pos(t), \forall n \in \mathbb{N}, p.n \in pos(t) \Leftrightarrow n < Ar_F(t(p))$$

In this thesis, every tree labeled by a ranked set will be assumed to respect its arity. The set of trees labeled by the ranked set  $F$  (and respecting its arity) will be denoted  $\mathcal{H}(F)$ .

### 2.3.2 Subtrees

Let  $p \in pos(t)$ . The *subtree* of  $t$  rooted at  $p$ , denoted  $t_{[p]}$ , is defined by:

$$pos(t_{[p]}) \stackrel{\text{def}}{=} \{q \in \mathbb{N}^* \mid pq \in pos(t)\}$$

$$\forall q \in pos(t_{[p]}), t_{[p]}(q) \stackrel{\text{def}}{=} t(pq)$$

A subtree is a *child* of  $t$  if there is an  $i \in \mathbb{N}$  and this subtree is  $t_{[i]}$ . The tree  $t$  is a *father* of this tree.

The notion of subtrees defines an ordering on  $\mathcal{H}(F)$ , denoted  $\triangleleft$ . The formal definition of this ordering is:

$$\forall u, t \in \mathcal{H}(F), u \triangleleft t \stackrel{\text{def}}{\Leftrightarrow} \exists p \in pos(t), u = t_{[p]}$$

If a  $u \triangleleft t$  or  $t \triangleleft u$  we say that  $u$  and  $t$  are *parent*.

The set of subtrees of a tree  $t$  is denoted  $Subtrees(t)$ .

### 2.3.3 Infinite Trees

A tree  $t$  is said to be infinite as soon as  $\text{pos}(t)$  is infinite.

A tree  $t$  is *regular* [Tho90] if and only if the number of its distinct subtrees is finite (or  $\text{Subtrees}(t)$  is finite). If  $t$  respects an arity function, then it is the case if and only if:

$$\exists N \in \mathbb{N}, \forall p \in \text{pos}(t), |p| > N \Rightarrow \exists q, q_1 \in \text{pos}(t), q \prec q_1 \preceq p \quad \text{and} \quad t_{[q]} = t_{[q_1]}$$

**Proof:** Suppose  $t$  is regular. We just have to take  $N$  the number of distinct subtrees of  $t$ . Conversely, if  $t$  satisfies the formula, we show that whatever  $t_{[p]}$ ,  $\exists q \in \text{pos}(t)$  such that  $|q| \leq N$  and  $t_{[q]} = t_{[p]}$ . Thus, the number of distinct subtrees of  $t$  is bounded by  $n^N$ , where  $n$  is the maximum arity of the labels.  $\square$

### 2.3.4 Substitutions in Trees

We present different notions of substitution in trees. Let  $t$  and  $u$  be trees and  $p$  a path of  $\text{pos}(t)$ . The substitution of  $t$  by  $u$  at position  $p$ , denoted  $t_{\{p\}\backslash u}$  is the tree  $v$  defined by:

$$\text{pos}(v) \stackrel{\text{def}}{=} \{q \in \text{pos}(t) \mid p \not\prec q\} \cup \{pq \mid q \in \text{pos}(u)\}$$

$$v(q) \stackrel{\text{def}}{=} \begin{cases} \text{if } \exists r \in \text{pos}(u), q = pr \text{ then } v(r) \\ \text{else } t(q) \end{cases}$$

If now  $P$  is a set of paths, the substitution of  $t$  by  $u$  at positions  $P$ , denoted  $t_{P\backslash u}$  is the tree  $v$  defined by:

$$\text{pos}(v) \stackrel{\text{def}}{=} \{q \in \text{pos}(t) \mid \forall p \in P, p \not\prec q\} \cup \bigcup_{p \in P} \{pq \mid q \in \text{pos}(u)\}$$

$$v(q) \stackrel{\text{def}}{=} \begin{cases} \text{if } \exists r \in \text{pos}(u), \exists p \in P, q = pr \text{ then } v(r) \\ \text{else } t(q) \end{cases}$$

If  $s$  is another tree, the substitution of  $s$  by  $u$  in  $t$ , denoted  $t[s\backslash u]$  is defined by:

$$t[s\backslash u] \stackrel{\text{def}}{=} t_{\{p \in \text{pos}(t) \mid t_{[p]} = s\}\backslash u}$$

We can also define substitutions with respect to a label  $f$ , which replaces every node labeled by  $f$  by  $u$ .

### 2.3.5 Recursive Functions on Trees

Basically, trees are recursive, so most functions defined on trees will be recursive. The recursive definition can be written as a fixpoint equation of the form:

$$f(t) = \mathcal{F} \left( t(\varepsilon), \bigotimes_{i < \text{Ar}_F(t(\varepsilon))} f(t_{[i]}) \right)$$

Depending on  $\mathcal{F}$  and its domain, this equation can have many solutions or even no solution at all. We know from [Tar55] that if  $\mathcal{F}$  is monotone on a complete lattice then the set of fixpoints of  $\mathcal{F}$  is non-empty and is a complete lattice. In that case, the usual convention is to consider the least fixpoint of  $\mathcal{F}$  as the semantics of the equation.

In the case of an infinite tree though, this convention will often lead to uninteresting results, that is the image of any infinite trees by the least fixpoint of  $\mathcal{F}$  is the bottom of the lattice. Thus the convention we follow in this thesis is to consider the greatest fixpoint as the semantics of the equation, including for finite trees.

### 2.3.6 Representation of a Tree

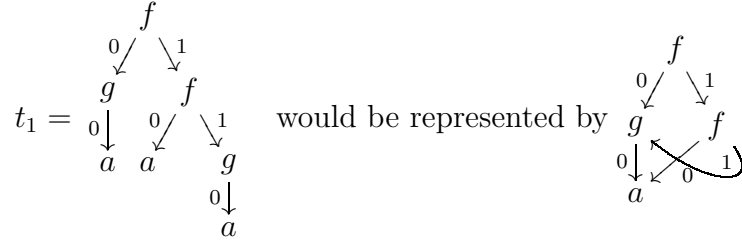
Finite trees have been very commonly used in computer science for years. It is not surprising that a very efficient way of representing trees is known and widely used (see for example [Knu97], Sect. 2.3). This method merely consists in representing a tree as a set of paths associated with their labels and then share the prefix parts of the paths. Thus, a tree is represented by its root and by arrows associated with each possible suffix of  $\varepsilon$ , starting from the root and pointing towards representations of the subtrees corresponding to the suffix. When it comes to representing objects, an illustrated example is often easier to understand than an exhaustive explanation:

$$t_1 = \begin{cases} \varepsilon & \rightarrow f \\ 0 & \rightarrow g \\ 00 & \rightarrow a \\ 1 & \rightarrow f \\ 10 & \rightarrow a \\ 11 & \rightarrow g \\ 110 & \rightarrow a \end{cases} \quad \text{will be represented by}$$

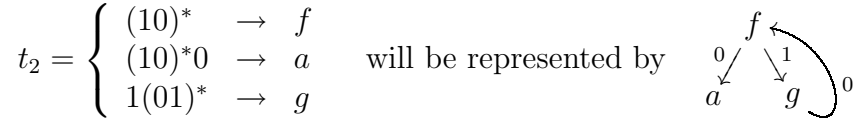
Usually, the values associated with the arrows are omitted. The convention is that the arrows are ordered from left to right according to their values. Even the arrows can be replaced by mere lines, with the convention that the tree is read top-down .

The qualities of this representation lie in its simplicity and in its ease of use. Most algorithms just go through the representation using local informations only.

This representation is often easily improved by sharing the end of the paths (see [AHU83] for example). In fact, a part of the path can be shared if its labels can be shared too. This amounts to share the common subtrees. At the expense of the maintenance of a global information (the set of already encountered subtrees), this technique allows an appreciable gain of space and time. In our example,



Moreover, this optimization allows for a very natural representation of infinite regular trees [Col82]. Thus:



The problem with this representation is that classical techniques for sharing in finite trees do not apply to infinite trees. In this thesis, a new technique has been devised that allows this representation to be canonical. It is described in length in chapter 3.

Other optimizations are possible, but the added complexity seems too heavy for a generic use of the trees.

In the sequel, a generic tree from  $\mathcal{H}(F)$  will be denoted  $\begin{matrix} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{matrix}$ , where  $f \in F$  is the root of the tree,  $n$  is the arity of  $f$  and  $(t_i)_{i < n}$  are the children of the root.

## 2.4 Graphs

A *directed graph*  $G$  is a set of nodes  $G^N$  and a set of edges  $G^E \subset G^N \times G^N$ . The empty graph (i.e.  $G^N = \emptyset$ ) is denoted  $\emptyset^G$ . A finite Graph is a graph with a finite set of nodes. A path from the node  $N$  to the node  $M$  of  $G^N$  is a word over  $G^E$  such that the first letter of the word is of the form  $(N, N')$ , the last letter of the word,  $(M', M)$ , and each consecutive letter of the word,  $(N_1, N_2)(N_2, N_3)$ . A directed graph is said to be *strongly connected* if and only if, for all nodes  $N$  and  $M$  of the graph, there is a path of the graph from  $N$  to  $M$ . If  $p$  is a path from  $N$  to  $M$ , we write  $N.p \stackrel{\text{def}}{=} M$ .

A symmetric graph<sup>1</sup> is a graph whose set of edges is symmetric. That is  $G$  is symmetric if and only if  $\forall(N, M) \in G^E, (M, N) \in G^E$ . The underlying symmetric graph of a directed graph is the graph with the same set of nodes and with a set of edges defined by: if  $(N, M)$  is an edge of  $G^E$ , then  $(N, M)$  and  $(M, N)$  are edges of the underlying symmetric graph of  $G$ . A graph is said to be *connected* if and only if its underlying symmetric graph is strongly connected.

A graph is said to be labeled if there is a function from its set of nodes to a set of labels.

A *subgraph*  $H$  of a graph  $G$  is a graph such that  $H^N \subset G^N$  and  $H^E = \{(N, M) \in G^E \mid N \text{ and } M \in H^N\}$ .

### 2.4.1 Multi-edge Graphs

A *multi-edge* graph  $G$  is a set of nodes  $G^N$ , a set of edges names  $G^\nu$  and a set of edges  $G^E \subset G^N \times G^N \times G^\nu$ . The underlying directed graph of a multi-edge graph is the directed graph defined by  $G^N$  and  $G^E$  where we forget the third component of the edges. A multi-edge graph is strongly connected (respectively connected) if and only if its underlying directed graph is strongly connected (respectively connected). A path of a multi-edge graph is a word over its set of edges such that when forgetting the third component of its letters, we obtain a path of the underlying directed graph. The word over the set of edges names obtained from a path of the graph is called the label of the path. If  $N$  is a node of a multi-edge graph,  $w$  a word over the set of edges names, we write  $N.w$  for the set of nodes  $M$  such that  $w$  is the label of a path from  $N$  to  $M$ .

### 2.4.2 Walking Through a Graph

Many algorithms on graphs proceed by “walking through” the graph, that is they go from nodes to nodes of the graphs, following the edges. The most popular way of going through a graph is the depth first search [Tar72]. Whatever the way of going through a graph, some common notions arise from the notion of visiting nodes sequentially. The *root* of a subgraph of the graph analyzed by the algorithm is the first node of the subgraph visited by the algorithm. The *return edges* of the graph are those edges that go from a node  $N$  visited by the algorithm to a node that is already visited when it reaches  $N$ .

## 2.5 Relations

Let  $I$  be a set of indexes, and  $(E_i)_{i \in I}$  a set family. A *relation*  $R$  on this family is a subset of  $\bigotimes_{i \in I} E_i$ , the cartesian product of the family. The elements of  $I$  are also called the *entries* of the relation.

<sup>1</sup>Sometimes, a symmetric graph is said to be simply a graph, as opposed to directed graphs.



### 2.5.1 Projection

Let  $e \in \bigotimes_{i \in I} E_i$ . If  $I = \emptyset$ ,  $e$  is denoted  $\langle \rangle$ . If  $I \neq \emptyset$ , then let  $j \in I$ . The *projection* of  $e$  according to its  $j^{\text{th}}$  component is denoted  $e_{(j)}$ . In the same way, if  $R$  is a subset of  $\bigotimes_{i \in I} E_i$ , its projection according to its  $j^{\text{th}}$  component is denoted  $R_{(j)}$ :

$$R_{(j)} \stackrel{\text{def}}{=} \{x \in E_j \mid \exists e \in R, x = e_{(j)}\}$$

Let  $J \subset I$ . The projection of  $e$  according to  $J$  is denoted likewise by  $e_{(J)}$ . In the same way,

$$R_{(J)} \stackrel{\text{def}}{=} \left\{ x \in \bigotimes_{j \in J} E_j \mid \exists e \in R, x = e_{(J)} \right\}$$

Dually, we will write  $e_{(\mathbb{C}J)}$  for  $e_{(I \setminus J)}$ , as  $I$  is known from  $e$ .

### 2.5.2 Support Set of a Relation

Let  $R$  be a relation on  $(E_i)_{i \in I}$ . The set family  $(F_i)_{i \in I}$  is called the *support set* of  $R$  if and only if  $R \subset \bigotimes_{i \in I} F_i$  and  $\forall i \in I, \forall x \in F_i, \exists e \in R, e_{(i)} = x$ . The support set of a relation  $R$  is denoted  $Support(R)$ . It is unique, as

$$Support(R) = \bigotimes_{i \in I} \{e_{(i)} \mid e \in R\}$$

A relation is said to be *square* if it is equal to its support set.

### 2.5.3 Boolean Operations on Relations

Let  $R$  be a relation and  $D \supset Support(R)$ . The negation of  $R$  with respect to  $D$  is defined by:

$$\neg_D R \stackrel{\text{def}}{=} \{e \in D \mid e \notin R\}$$

When  $D$  is clear from the context, it can be omitted.

Let  $R$  and  $S$  be two relations. Let  $R$  be a relation on  $\bigotimes_{i \in I} E_i$  and  $S$  on  $\bigotimes_{i \in J} F_j$ . We define  $extend(R, S) \stackrel{\text{def}}{=} \bigotimes_{i \in I \cup J} E_i \cup F_i$  with  $E_i = \emptyset$  if  $i \notin I$  and  $F_i = \emptyset$  if  $i \notin J$ . Then:

$$R_1 \wedge R_2 \stackrel{\text{def}}{=} \{e \in extend(R, S) \mid e_{(I)} \in R \text{ and } e_{(J)} \in S\}$$

$$R_1 \vee R_2 \stackrel{\text{def}}{=} \{e \in extend(R, S) \mid e_{(I)} \in R \text{ or } e_{(J)} \in S\}$$

### 2.5.4 Projection According to a Value

Let  $J \subset I$ . Let  $e \in \bigotimes_{i \in J} E_i$ . Let  $R$  be a relation on  $(E_i)_{i \in I}$ . The *projection*<sup>2</sup> of  $R$  according to  $e$  is the relation  $R_{:J=e}$  defined by:

$$R_{:J=e} \stackrel{\text{def}}{=} \left\{ x \in \bigotimes_{i \in I \setminus J} E_i \mid \exists y \in R, y_{(\mathbb{C}J)} = x \wedge y_{(J)} = e \right\}$$

In the case where  $J = \{j\}$ , we can write  $R_{:j=e} \stackrel{\text{def}}{=} R_{:\{j\}=e}$ . If  $J$  is clear from the context, we write  $R(e)$  for  $R_{:J=e}$ .

If there is a family  $(F_j)_{j \in K}$  isomorphic to  $(E_i)_{i \in J}$ , and the isomorphism  $\sigma$  is clear from the context, we will write  $R_{:J=e}$  for  $R_{:J=\sigma(e)}$ .

The set of all possible projections according to a value of  $R$  is defined by  $\text{AllProj}(R) \stackrel{\text{def}}{=} \{ R_{:J=e} \mid J \subset I \text{ and } e \in \bigotimes_{i \in J} E_i \}$ .

### 2.5.5 Independent Decomposition

Let  $I = I_1 \cup I_2$  and  $I_1 \cap I_2 = \emptyset$ . A relation  $R$  can be independently decomposed in  $R_{(I_1)}$  and  $R_{(I_2)}$ , written  $R = R_{(I_1)} \bar{\wedge} R_{(I_2)}$ , if and only if:

$$\forall e \in \bigotimes_{i \in I} E_i, \quad e \in R \Leftrightarrow (e_{(I_1)} \in R_{(I_1)} \wedge e_{(I_2)} \in R_{(I_2)})$$

## 2.6 Abstract Interpretation

The way abstract interpretation deals with semantic approximations ([Cou78, CC92b, Cou96]) is through the use of semantic domains and the relationship between them. Given two semantics, one is called the *concrete semantics* and the other one the *abstract semantics* which is an approximation of the concrete one. In the case of program analysis, the most concrete semantics is the *standard semantics* which describes all the possible behaviors of programs. A first approximation is the *collecting semantics* which contains the behaviors of interest. Thus, when comparing those two semantics, the standard semantics is the concrete one, and the collecting semantics the abstract one. Other more abstract semantics may be introduced, compared to which the collecting semantics will be the concrete semantics.

### 2.6.1 Soundness and Galois Connections

The properties of the different semantics are taken from sets called *semantic domains*. So we have a concrete semantic domain  $\mathcal{P}^b$  and an abstract semantic domain  $\mathcal{P}^\sharp$ . The most basic way of describing the relationship between the

<sup>2</sup>This is also called *cofactor* in [BHMSV84].

concrete and the abstract semantics is the *soundness relation*,  $S$ , a relation on  $\mathcal{P}^b \times \mathcal{P}^\sharp$ .  $\langle c, a \rangle \in S$  means that  $a$  is a sound approximation of  $c$ .

Because we can have many sound approximation for a given property, we need to know more about these approximations, about their accuracy. This information is given by a partial ordering on the properties, the *approximation ordering*, denoted  $\leq^b$  for the concrete domain, and  $\leq^\sharp$  for the abstract one. The approximation ordering must be compatible with the soundness relation: if  $\langle c, a \rangle \in S$ , then,  $a \leq^\sharp a' \Rightarrow \langle c, a' \rangle \in S$  and  $c' \leq^b c \Rightarrow \langle c', a \rangle \in S$ . If there is a best abstraction for all concrete properties, then we can define an *abstraction function*,  $\alpha$  which maps every concrete property to its best abstraction. Dually, if there is a greatest concrete property that is soundly approximated by a given  $a$  for any  $a$ , then there is a concretisation function  $\gamma$ .

If we have an abstraction function and a concretisation function, the relationship between the semantics can be described by a Galois connection:

$$(\mathcal{P}^b, \leq^b) \xleftrightarrow[\alpha]{\gamma} (\mathcal{P}^\sharp, \leq^\sharp)$$

It means that  $(\mathcal{P}^b, \leq^b)$  and  $(\mathcal{P}^\sharp, \leq^\sharp)$  are partially ordered sets and that  $\alpha : \mathcal{P}^b \rightarrow \mathcal{P}^\sharp$  and  $\gamma : \mathcal{P}^\sharp \rightarrow \mathcal{P}^b$  are such that:  $\forall a \in \mathcal{P}^\sharp, \forall c \in \mathcal{P}^b, \alpha(c) \leq^\sharp a \Leftrightarrow c \leq^b \gamma(a)$ . When  $\alpha$  is surjective, we say that we have a *Galois insertion*.

## 2.6.2 Fixpoints and Iteration Sequences

The semantics of a program will often be described in terms of fixpoint of a semantic function,  $F$ . In order to insure the existence of such a fixpoint and to characterize it constructively ([CC79]), we suppose that the semantic domains are complete lattices for some partial orderings, the computation orderings, denoted  $\sqsubseteq^b$  and  $\sqsubseteq^\sharp$ , which can be different from the approximation orderings. In this case, the least fixpoint of  $F^b : \mathcal{P}^b \rightarrow \mathcal{P}^b$ , a monotonic function for  $\sqsubseteq^b$ , is the limit of the concrete iteration sequence:

$$\begin{aligned} F^{b\beta+1} &\stackrel{\text{def}}{=} F^b \left( F^{b\beta} \right) \text{ for all } \beta \text{ ordinal} \\ F^{b\lambda} &\stackrel{\text{def}}{=} \bigsqcup_{\beta < \lambda} F^{b\beta} \text{ where } \lambda \text{ is a limit ordinal}^3 \end{aligned}$$

What we try to compute, now, is a sound approximation of the least fixpoint of  $F^b$ . It can be performed by the abstract iteration sequence over the abstract semantic function  $F^\sharp$  defined as:  $F^\sharp(a) \stackrel{\text{def}}{=} \alpha(F^b(\gamma(a)))$ . Note that the least fixpoint of this function may be less precise than the abstraction of the least fixpoint of the concrete semantic function.

---

<sup>3</sup>Note that 0 is a limit ordinal. The least upper bound of the empty set is called the infimum of the lattice.

### 2.6.3 Dynamic Approximation of Fixpoints

The approximation of the concrete fixpoint by the abstract iteration sequence is a static approximation: we choose the abstract domain for a whole class of programs and it determines *a priori* the approximation. But the more precise these approximations, the more precise the result. In fact, experiments show that we get better results by choosing a more precise abstraction, which can lead to unpractical abstract iteration sequences, and then dynamically approximate the abstract iteration sequence. In abstract interpretation, this is performed by means of a *widening* operator.

A widening operator  $\nabla$  is a partial function:  $\wp(\mathcal{P}^\sharp) \rightarrow \mathcal{P}^\sharp$  such that  $\forall E^\sharp \subset \mathcal{P}^\sharp$ , if  $\nabla(E^\sharp)$  exists,  $\forall c \in \mathcal{P}^b$ , if  $\exists a \in E^\sharp$  such that  $\langle c, a \rangle \in S$ , then  $\langle c, \nabla E^\sharp \rangle \in S$ .

The abstract iteration sequence with widening is:

$$F^\sharp \uparrow^{\beta+1} \stackrel{\text{def}}{=} \nabla \left\{ F^\sharp \uparrow^\beta, F^\sharp \left( F^\sharp \uparrow^\beta \right) \right\} \text{ for all } \beta \text{ ordinal}$$

$$F^\sharp \uparrow^\lambda \stackrel{\text{def}}{=} \nabla \left\{ F^\sharp \uparrow^\beta \mid \beta < \lambda \right\} \text{ where } \lambda \text{ is a limit ordinal}$$

The limit of the abstract iteration sequence with widening is a post fixpoint of  $F^\sharp$ . Thus it is a sound approximation of the least fixpoint of  $F^b$ .

# Chapter 3

## Sharing Regular Trees

Sharing common substructures is one of the leading ideas of efficient representations. It allows more compact representations and reuse of intermediate results. It is always involved in a way or another when representing infinite structures.

In general, sharing mechanisms use a dictionary mechanism to know whether a given substructure has already been encountered. These mechanisms only work for acyclic structures, as they need to start on a basic substructure that does not depend on some other substructure. So, these classical mechanisms would not apply to infinite regular trees. In this chapter, we present a new sharing mechanism that can be applied to any structure, even cyclic or connected structures.

We present first the classical sharing mechanism, that is used for finite trees. Then we present the new mechanism on strongly connected graphs, which are the essence of the difficulty in the sharing of infinite regular trees. This mechanism is then used together with the classical one to build the shared representation of any infinite regular tree. We end this chapter with the description of some algorithms on this representation.

### 3.1 Classic Sharing

#### 3.1.1 General Framework

The main problem of sharing is to store intermediate substructures when building a structure in such a way that it is easy to know whether this substructure has been encountered already, and where it is stored. In order to do this, we use keys extracted from the structures, which are linked —through a dictionary mechanism— to the places where these structures are stored. For this process to be valid, the keys must properly discriminate between the structures. We suppose we have a structure (and substructure) universe,  $\mathcal{U}$ . We describe the possible semantic equality between structures by an equivalence class  $\equiv$  on  $\mathcal{U}$ . A

key  $k$  is a *valid key* with respect to  $\equiv$  if:

$$\forall u, v \in \mathcal{U}, u \equiv v \Leftrightarrow k(u) = k(v)$$

In this way, if two structures are equivalent, they are stored in the same place: we keep just one representative of the equivalence class.

### 3.1.2 Structures for the Trees

In our paradigm, each regular tree  $t$  labeled on  $F$  is represented by a node of a labeled multi-edge graph  $G_t$ , with set of labels  $F$  and set of edges names  $\mathbb{N}$ . In the computer representation, the idea is that each node of those graphs corresponds to a memory location. Thus, the structure universe for the trees will be the nodes of such graphs. We define  $\equiv_t$  the semantic equivalence on the nodes:  $N \equiv_t M$  if and only if  $N$  and  $M$  represent the same tree.

Let  $N$  be a node of  $G_t$  with label  $f$  of arity  $Ar_F(f) = n$ . As defined in chapter 2, there are  $n$  edges from  $N$  to nodes  $N_i$  whose name is  $i$  in  $G_t^E$ . As

a shortcut, we will write  $N$  is labeled by  $\underset{N_0 \dots N_{n-1}}{\overset{f}{\swarrow \searrow}}$ . As the outgoing edges of a

particular node are uniquely determined by their names, there is at most one path labeled by a given word of  $\mathbb{N}^*$  starting from a given node. Thus, if  $p$  is a word of  $\mathbb{N}^*$  and  $N$  a node,  $N.p$  is either empty or a singleton. When it is a singleton, we will identify the singleton and its element.

Each node of the graph  $G_t$  represents a subtree of  $t$ . The graph is said to be *of maximal sharing* if the number of its nodes is the number of different subtrees of  $t$ . In fact, the graph  $G_t$  is part of a wider graph defined by a dictionary  $D$ , which links any subtree of  $t$ , and possibly of some other trees, to their nodes.

### 3.1.3 Finite Trees

In the case of finite trees, sharing is easy and widely used. The dictionary mechanism used to link the keys to the nodes is usually a hash table<sup>1</sup> ([Knu73], Sect. 6.4). The keys are built as follows: first, sharing the leaves is easy, as the label of the node can be its key. Then, if the keys of  $(t_i)_{i < n}$  are known, they

are linked to the nodes  $(N_i)_{i < n}$ . The key for  $\underset{t_0 \dots t_{n-1}}{\overset{f}{\swarrow \searrow}}$  is  $(f, (N_i)_{i < n})$ . Each time

a key is not present in the dictionary, it is associated with a new unique node, which is linked to the nodes  $(N_i)_{i < n}$  by edges labeled by the corresponding  $i$ . The key defined by this mechanism is valid for  $\equiv_t$  by induction on the height of the trees represented by the nodes. The graphs resulting of this building mechanism are of maximal sharing due to the validity of the keys.

<sup>1</sup>An interesting alternative to hashing is proposed in [CP95]. It is used in [KR96].

## 3.2 Sharing Strongly Connected Graphs

The main difference between the representations of finite and infinite (regular) trees, is that the graphs representing infinite trees contain strongly connected subgraphs. Finding the best representative and a key for such subgraphs is difficult, as we can't use any recursion mechanism. We are thus forced to use non-local informations.

### 3.2.1 Maximal Sharing Strongly Connected Graphs

#### Equivalence of Graph Nodes

In order to achieve maximal sharing, we need to know constructively when two nodes represent the same tree. The following simple algorithm decides the equivalence of two nodes, which can be in the same graph or not. Let  $N$  and  $M$  be two graph nodes. The two nodes represent the same tree if and only if  $\text{eqTree}(N, M)$  holds **true**. A similar algorithm was already sketched in [Knu97] for “List structures” representing infinite trees (exercise 2.3.5-12).

$\text{eqTree}(N, M)$ :

$S \leftarrow \emptyset$

**return**( $\text{eqTreeRec}(N, M)$ )

$\text{eqTreeRec}(N, M)$ :

**if**  $\{N, M\} \in S$  **or**  $N = M$  **then return**(**true**)

$N$  is labeled by  $\begin{matrix} f \\ \swarrow \searrow \\ N_0 \dots N_{n-1} \end{matrix}$  and  $M$  is labeled by  $\begin{matrix} g \\ \swarrow \searrow \\ M_0 \dots M_{m-1} \end{matrix}$

**if**  $f = g$  **then**

$S \leftarrow S \cup \{\{N, M\}\}$

**return**( $\bigwedge_{i < n} \text{eqTreeRec}(N_i, M_i)$ )

**else return**(**false**)

**Proof:**  $N$  represents the tree  $u$  and  $M$  the tree  $v$ . If  $\text{eqTree}(N, M)$  is **false**, then there is a path  $p$  such that  $u(p) \neq v(p)$  and so  $u \neq v$ .

If  $u \neq v$ , then there is a path  $p$  such that  $u(p) \neq v(p)$ . We try to prove that there is such a path that is reached by  $\text{eqTree}(N, M)$ . If it is the case, then  $\text{eqTree}(N, M) = \text{false}$ .

First we restrict the set of eligible paths to  $\text{Acyclic}(M, N) \stackrel{\text{def}}{=} \{p \mid \forall q \prec q' \preceq p, \{M.q, N.q\} \neq \{M.q', N.q'\}\}$ . Any path that is reached by  $\text{eqTree}(N, M)$  is in  $\text{Acyclic}(M, N)$ . Note that  $\text{Acyclic}(M, N)$  is a finite set, and so orders on this set are well founded. Let  $p$  be a path such that  $u(p) \neq v(p)$  of minimal length. If  $p$  is not in  $\text{Acyclic}(M, N)$  then there is  $q \prec q' \preceq p$  such that  $\{M.q, N.q\} = \{M.q', N.q'\}$ . Let  $p = q'p_1$ .  $\{u_{[q]}, v_{[q]}\} = \{u_{[q']}, v_{[q']}\}$ , and  $u_{[q']}(p_1) \neq v_{[q']}(p_1)$ , so

$u_{[q]}(p_1) \neq v_{[q]}(p_1)$  which means that  $u(qp_1) \neq v(qp_1)$ . But  $|qp_1| < |p|$ . Thus  $p$  is in  $Acyclic(M, N)$ , and so  $Acyclic(M, N)$  contains at least one path such that  $u$  and  $v$  differ on that path.

Let  $p_f$  be the least path of  $Acyclic(M, N)$  for  $\prec_{\mathbb{N}}^*$  such that  $u(p_f) \neq v(p_f)$ . If  $p_f$  is not reached by  $\text{eqTree}(N, M)$ , then there is a  $q \prec p_f, q \in Acyclic(M, N)$  such that, either  $N.q = M.q$ , or there is a  $q_f \prec_{\mathbb{N}}^* q, q_f \in Acyclic(M, N)$  such that  $\{N.q, M.q\} = \{N.q_f, M.q_f\}$ . Let  $p_1$  be such that  $p_f = qp_1$ . In the first case,  $u_{[q]} = v_{[q]}$ , and so  $u_{[q]}(p_1) = v_{[q]}(p_1)$ , which means that  $u(p) = v(p)$ . In the second case,  $\{u_{[q]}, v_{[q]}\} = \{u_{[q_f]}, v_{[q_f]}\}$ . But  $q_f \prec_{\mathbb{N}}^* q$ , and  $q_f \not\prec q$  because  $q \in Acyclic(M, N)$ , so  $q_f p_1 \prec_{\mathbb{N}}^* q$ , and so  $q_f p_1 \prec_{\mathbb{N}}^* qp_1$ . By minimality of  $p_f$ , this means that  $q_f p_1 \notin Acyclic(M, N)$ . So, there are  $r \prec r' \preceq q_f p_1$  such that  $\{M.r, N.r\} = \{M.r', N.r'\}$ . We take for  $r$  the least such path for  $\prec$ . Because  $q_f \in Acyclic(M, N)$ , we cannot have both  $r$  and  $r'$  prefix of  $q_f$ . Because  $p_f \in Acyclic(M, N)$ , we cannot have both  $r$  and  $r'$  suffix of  $q_f$ . Thus,  $r \preceq q_f \prec r'$ . Because  $p_f \in Acyclic(M, N)$ ,  $r \not\prec p_f$ , so  $r \not\prec q$ . Let  $q_f p_1 = r' p_2$ . We have  $\{u_{[rp_2]}, v_{[rp_2]}\} = \{u_{[p_f]}, v_{[p_f]}\}$ , but this time  $rp_2 \in Acyclic(M, N)$ . As  $r \prec q_f$ ,  $r \not\prec q$  and  $q_f \prec_{\mathbb{N}}^* q$ ,  $r \prec_{\mathbb{N}}^* q$ , and so  $rp_2 \prec_{\mathbb{N}}^* p_f$ . This contradicts the minimality of  $p_f$ . It proves that  $p_f$  is reached by  $\text{eqTree}(N, M)$  and so that  $\text{eqTree}(N, M)$  holds false.  $\square$

### Finding the Graph of Maximal Sharing

A graph is of maximal sharing if and only if there is no couple of nodes,  $N_1$  and  $N_2$ , such that  $N_1 \neq N_2$  and  $N_1$  and  $N_2$  represent the same subtree. One way of finding the graph with maximal sharing equivalent to a given graph is to test every pair of nodes with  $\text{eqTree}$ , and each time the result of the algorithm is **true**, merge the two nodes. The  $\text{eqTree}$  algorithm is quite efficient in practice, despite its quadratic worst case complexity. But testing every pair of nodes leads to a quadratic complexity in any case. It is possible to be more efficient by using a partitioning algorithm based on the same principle as Hopcroft's automata minimization algorithm [Hop71]. As the graph we partition represents a tree, it is not the most general problem, and so we present a simpler algorithm than [CC82].

The following algorithm modifies a graph into an equivalent graph (i.e. representing the same tree) with maximal sharing. The node  $G$  is a node of the graph to be modified. It is also a node of the resulting graph.

**share**( $G$ ):

  Blocks is a function which associates with every integer the empty set

  BCount  $\leftarrow 0$

$D \leftarrow \emptyset$

  initiateBlocks( $G$ )



MaxAriety is the maximal arity of the labels in the graph

```

for  $i \leftarrow 0$  to MaxAriety - 1 do
   $a \leftarrow$  the block with the greatest number of incoming edges labeled by  $i$ 
  ToTest( $i$ )  $\leftarrow$  [BCount] \ { $a$ }
while  $\exists i$  such that ToTest( $i$ )  $\neq \emptyset$  do
  take a  $a$  out of ToTest( $i$ )
   $k \leftarrow$  BCount - 1
  for  $b \leftarrow 0$  to  $k$  do
    Blocks(BCount)  $\leftarrow$  {  $N \in$  Blocks( $b$ ) |  $\exists M \in$  Blocks( $a$ ),  $N \rightarrow^i M$  }
    if Blocks(BCount)  $\neq \emptyset$  and Blocks(BCount)  $\neq$  Blocks( $b$ ) then
      for  $j \leftarrow 0$  to MaxAriety - 1 do
        if the number of nodes leading (labeled by  $j$ ) to the block  $b$  is
          greater than the number of nodes leading to the block BCount
          and  $b \notin$  ToTest( $j$ ) then add  $b$  to ToTest( $j$ )
        else add BCount to ToTest( $j$ )
      BCount  $\leftarrow$  BCount + 1
  for  $a \leftarrow 0$  to BCount - 1 do
    if |Blocks( $a$ )| > 1 then
      if  $G \in$  Blocks( $a$ ) then  $N \leftarrow G$ 
      else  $N$  is any node in Blocks( $a$ )
       $\forall M \in$  Blocks( $a$ ),  $M \neq N$  do
        every edge  $O \rightarrow^i M$  is replaced by  $O \rightarrow^i N$ 
        remove  $M$  from  $G^N$ 

```

initiateBlocks( $N$ ):

```

if  $N \notin D$  then
  add  $N$  to  $D$ 
   $N$  is labeled by  $\begin{matrix} f \\ \swarrow \searrow \\ N_0 \dots N_{n-1} \end{matrix}$ 
  if  $\exists i <$  BCount such that Blocks( $i$ ) contains a node labeled by  $f$  then
    add  $N$  to Blocks( $i$ )
  else
    Blocks(BCount)  $\leftarrow$  { $N$ }
    BCount  $\leftarrow$  BCount + 1
  for  $i \leftarrow 0$  to  $n - 1$  do initiateBlocks( $N_i$ )

```

The algorithm first creates a partition (represented by Blocks) according to the labels of the nodes, and then progressively refines it until we get the coarsest congruence which refines the initial partition. With an adequate representation of the different sets involved, this algorithm yields an  $n \log n$  worst case complexity. To compute efficiently the best partitions, we can compute the inverse of the graph (the sets of nodes which are linked to a given node) during the initial traversal of the graph. In practice, we also mark the blocks containing only

one element, which cannot be refined further, and when a block contains two elements, we use the `eqTree` algorithm to refine it or merge the two elements (and possibly many other during the process). It seems to give faster results in practice.

**Example:**

$$\text{share} \left( \begin{array}{c} \left( \begin{array}{c} k_1 \xrightarrow{1} k_2 \\ \left( \begin{array}{c} 0 \\ 0 \end{array} \right) \left( \begin{array}{c} 1 \\ 1 \end{array} \right) \\ \left( \begin{array}{c} k_1 \xrightarrow{1} k_2 \\ \left( \begin{array}{c} 0 \\ 0 \end{array} \right) \left( \begin{array}{c} 1 \\ 1 \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) = 0 \begin{array}{c} \curvearrowright \\ \curvearrowleft \end{array} k_1 \begin{array}{c} \xrightarrow{1} \\ \xleftarrow{0} \end{array} k_2 \begin{array}{c} \curvearrowright \\ \curvearrowleft \end{array} 1$$

◇

### 3.2.2 Keys of Strongly Connected Graphs

When using the graphs, it is efficient to use as a key of a node its label and the nodes of its neighbors, as for finite trees. The problem with strongly connected graphs is that some nodes must be chosen arbitrarily, because the nodes of a strongly connected graph all depend on each other. So the simple key that is the label of the node and the neighbor nodes is not enough to know if a given strongly connected graph have already been encountered elsewhere. So we give another key which is valid for strongly connected graphs with maximal sharing.

As we use the graphs to represent trees, we always distinguish one node of the graph, which is the root of the tree. This node may be any node of the graph, so we must compute a key for each node of a given graph, each of them linked to that node. The pointed graph is first transformed into a finite tree which then gives a unique key as defined above. The tree associated with a given node is labeled by the labels of  $G^N$  and elements of  $\mathbb{N}^*$ . The arity of the elements of  $\mathbb{N}^*$  is 0. The elements of  $\mathbb{N}^*$  are the smallest (for  $\prec$ ) paths without cycle that lead from the root to a given node. In a sense, they represent these nodes when we loop in the graph. The tree is computed through the following algorithm:

`treeKey(N):`

$S$  is a partial function from  $G^N$  to  $\mathbb{N}^*$  and  $\text{dom}(S) = \emptyset$

**return**(`treeKeyRec`( $\varepsilon, N$ ))

`treeKeyRec(p, N):`

**if**  $N \in \text{dom}(S)$  **then return**( $S(N)$ )

add  $N \rightarrow p$  to  $S$

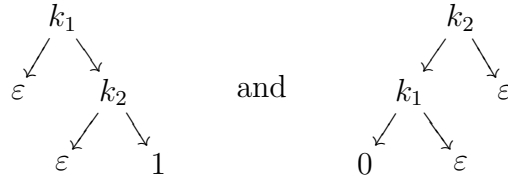
$N$  is labeled by  $\begin{array}{c} f \\ \swarrow \searrow \\ N_0 \dots N_{n-1} \end{array}$

```

for  $i \leftarrow 0$  to  $n - 1$  do  $t_i \leftarrow \text{treeKeyRec}(p.i, N_i)$ 
return  $\begin{pmatrix} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{pmatrix}$ 

```

**Example:** The keys of the two nodes of  $\begin{matrix} & & 1 & & \\ & \curvearrowright & k_1 & \xrightarrow{\quad} & k_2 & \curvearrowleft \\ & 0 & & & 0 & \\ & & & & & 1 \end{matrix}$  are respectively



◇

As with any key, `treeKey` defines an equivalence. This equivalence relates isomorphic graphs, that is, graphs with a one to one node equivalence. This equivalence is included in  $\equiv_t$  but is not equal to  $\equiv_t$ . Thus, `treeKey` is not a valid key for  $\equiv_t$  on any graph. But the point is that the two equivalences are indeed equal when restricted on graphs with maximal sharing:

**Property 3.1** *Whatever  $M$  and  $N$ , nodes of graphs with maximal sharing,  $\text{treeKey}(M) = \text{treeKey}(N) \Leftrightarrow M \equiv_t N$ .*

**Proof:** The difficult point is  $M \equiv_t N \Rightarrow \text{treeKey}(M) = \text{treeKey}(N)$ . Suppose there are  $M$  and  $N$  nodes of graphs with maximal sharing such that  $M \equiv_t N$  and  $\text{treeKey}(M) \neq \text{treeKey}(N)$ . Let  $t_M = \text{treeKey}(M)$  and  $t_N = \text{treeKey}(N)$ . Because  $t_M \neq t_N$ , there is a path  $p$  such that  $t_M(p) \neq t_N(p)$ . But if  $t_M(p)$  is a label of the graph,  $t_M(p)$  is the label of  $M.p$ , and the same holds for  $N$ . Because  $M \equiv_t N$ ,  $M.p$  and  $N.p$  have the same label, so at least one of  $t_M(p)$  or  $t_N(p)$  is not a label of the graphs (and so is in  $\mathbb{N}^*$ ), say  $t_M(p)$ . It means there is a  $q \prec p$  such that  $M.q \equiv_t M.p$ . So  $N.q \equiv_t N.p$ , but by maximal sharing of  $N$ ,  $N.q$  and  $N.p$  must be the same node, and so  $t_N(p) = q = t_M(p)$ . □

Because we can find an equivalent graph with maximal sharing for strongly connected graphs, we have a valid key mechanism for any strongly connected graph: we first apply `share`, then `treeKey`.

## 3.3 Regular Trees

### 3.3.1 Informal Presentation

We present the sharing of regular trees in algorithmic form. The idea of this algorithm is to use the classical sharing algorithm combined with the algorithm for strongly connected graphs. As any regular tree is not either finite or represented by a strongly connected graph, we must follow a kind of decomposition algorithm. This decomposition algorithm allows the procedure to be incremental.

First, we apply the classical algorithm as much as we can on finite subtrees of the tree we want to represent. Each finite subtree being associated with a node by the classical algorithm, they can be considered as a leaf of a new tree.

In this new tree, we eliminate the leaves by incorporating them into the labels of their fathers. This is coded by what we call partial keys, which are labels associated with the nodes of their children that are leaves of this tree. A partial key is considered as a new label of lesser arity than the label it is based on. This encoding in partial key gives another new tree with no finite subtree.

A tree with no finite subtree contains a subtree represented by a strongly connected graph. We associate a node with this subtree through a specific algorithm based on the algorithm devised in the preceding section. This algorithm is quite involved to efficiently reduce cycles. Then we can iterate on a new tree where this subtree is considered as a leaf labeled by its node. This new tree contains at least one finite subtree, so we can iterate our algorithm.

This process is iterated until we can build the key of the root. It is a finite process because at each iteration, we associate a node with a new subtree, and the number of distinct subtrees of a regular tree is finite by definition.

### 3.3.2 Algorithm

$D$  is the dictionary linking keys to nodes of a graph representing a tree. A partial key  $k$  is a name  $\mathbf{name}(k) \in F$  and a partial function from  $[Ar_F(\mathbf{name}(k))]$  to nodes.  $D_G$  is the dictionary linking keys obtained by `treeKey` to nodes of the graph representing the strongly connected graph.

We suppose that we can sometimes know if two subtrees are equals, but we may allow that sometimes two isomorphic subtrees won't be recognized as equals. In order for the algorithm to terminate, this should only happen a finite number of times. For example, the tree is represented with no maximal sharing, and there is a finite number of representatives for a given subtree. Because we allow a previous representation of trees that does not share, we must use the `share` algorithm to reduce the strongly connected graphs. This tolerance is useful in some algorithms manipulating shared regular trees.

The algorithm is illustrated on a full example section 3.3.5, page 32.

representation( $t$ ):

$(N, S, G) \leftarrow \text{represent}(t, \{t\})$   
**return**( $N$ )

$\text{represent}\left(\begin{array}{c} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}, T\right)$ :

$t \leftarrow \begin{array}{c} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}$

$k$  is the partial key such that  $\text{name}(k) = f$  and  $\text{dom}(k) = \emptyset$

$S$  is a partial function from trees to sets of nodes and numbers and  $\text{dom}(S) = \emptyset$

$N$  is a new node

**for**  $i \leftarrow 0$  **to**  $n - 1$  **do**  
  **if**  $t_i \in T$  **then** add  $t_i \rightarrow (N, i)$  to  $S$   
  **else**  $(N_i, S_i, G_i) \leftarrow \text{represent}(t_i, T \cup \{t_i\})$   
  **if**  $\text{dom}(S_i) = \emptyset$  **then** add  $i \rightarrow N_i$  to  $k$   
  **else** add  $S_i$  to  $S$   
10 **if**  $\text{dom}(S) = \emptyset$  **then**  
  **if**  $k \in \text{dom}(D)$  **then** **return**( $D(k), S, \emptyset^G$ )  
  **else**  
  Let  $N$  be labeled by  $f$  and  $N.i \leftarrow N_i$   
  add  $k \rightarrow N$  to  $D$   
  **return**( $N, S, \emptyset^G$ )  
**else**  
 $G^N \leftarrow \bigcup_{i < n} G_i^N$   
 $G^E \leftarrow \bigcup_{i < n} G_i^E$   
  add  $N$  labeled by  $k$  to  $G^N$   
20 add  $\{(N, N_i, i) \mid \text{dom}(S_i) \neq \emptyset\}$  to  $G^E$   
  **if**  $t \in \text{dom}(S)$  **then**  
  **while**  $S(t) \neq \emptyset$  **do**  
  take  $(M, i)$  out of  $S(t)$   
  add  $(M, N, i)$  to  $G^E$   
  delete  $t$  from  $\text{dom}(S)$   
  **if**  $\text{dom}(S) = \emptyset$  **then** **return**( $\text{representCycle}(N), S, \emptyset^G$ )  
  **else** **return**( $N, S, G$ )

representCycle( $N$ ):

$l_1$  is the key obtained from  $\text{treeKey}(N)$

**if**  $l_1 \in \text{dom}(D_G)$  **then** **return**( $D_G(l_1)$ )

$D_S$  and  $D_D$  are two empty node dictionary

$b \leftarrow \text{shareWithDone}(N)$

**if**  $b$  **then**

add  $l_1 \rightarrow D_S(N)$  to  $D_G$

```

    return( $D_S(N)$ )
  else
    share( $N$ )
10   $l_2$  is the key obtained from treeKey( $N$ )
    if  $l_2 \in \text{dom}(D_G)$  then
      | add  $l_1 \rightarrow D_G(l_2)$  to  $D_G$ 
      | return( $D_G(l_2)$ )
    else
      | add  $l_1 \rightarrow N$  to  $D_G$ 
      |  $\forall M \in G^N$  do
      |    $M$  is labeled by  $m$ 
      |    $m'$  is  $m$  completed by the children of  $M$  in  $G$ .
      |   add  $m' \rightarrow M$  to  $D$ 
20  |  $l'$  is the key obtained from treeKey( $M$ )
      | add  $l' \rightarrow M$  to  $D_G$ 
      | return( $N$ )

shareWithDone( $N$ )
  if  $N \in \text{dom}(D_S)$  then return(false)
  add  $N$  to  $D_S$ 
   $N$  is labeled by  $\begin{matrix} k \\ \swarrow \searrow \\ N_0 \dots N_{n-1} \end{matrix}$ 
   $\forall i \in \text{dom}(k)$  do if tryShare( $N, k(i), i, k(i)$ ) then return(true)
  for  $i \leftarrow 0$  to  $n - 1$  do if shareWithDone( $N_i$ ) then return(true)
  return(false)

tryShare( $N, M, i, O$ )
  if  $M \in \text{dom}(D_D)$  then return(false)
  add  $M$  to  $D_D$ 
  if  $N$  and  $M$  have the same label and  $M.i = O$  then
     $D_C$  is an empty node dictionary
    if compareWithDone( $N, M$ ) then
      replace  $D_S$  by  $D_C$ 
      return(true)
   $M$  is labeled by  $\begin{matrix} f \\ \swarrow \searrow \\ M_0 \dots M_{n-1} \end{matrix}$ 
  return( $\bigvee_{j < n}$  tryShare( $N, M_j, i, O$ ))

compareWithDone( $N, M$ )
  if  $N \in \text{dom}(D_C)$  then
    if  $D_C(N) = M$  then return(true) else return(false)

```

$N$  is labeled by  $\begin{matrix} f \\ \swarrow \searrow \\ N_0 \dots N_{n-1} \end{matrix}$  by extending the partial key  $k$  labeling  $N$

$M$  is labeled by  $\begin{matrix} g \\ \swarrow \searrow \\ M_0 \dots M_{m-1} \end{matrix}$

```

if  $f = g$  then
  add  $N \rightarrow M$  to  $D_C$ 
  for  $i \leftarrow 0$  to  $n - 1$  do
    if  $i \in \text{dom}(k)$  then if  $N_i \neq M_i$  then return(false)
    else if  $\text{compareWithDone}(N_i, M_i) = \text{false}$  then return(false)
  return(true)
else return(false)

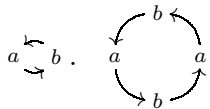
```

### 3.3.3 More Remarks to Understand the Algorithm

#### Global Algorithm

The `represent` function is evaluated on a tree  $t$  and a set of trees  $T$ . Its result is a node  $N$ , a partial function  $S$  and a graph  $G$ .  $t$  is the subtree that we examine. In particular we have to decide whether or not to associate it with a new node.  $T$  is the set of trees that contains  $t$  and the subtrees that have been encountered so far. It is used to detect cycles in the tree. Each cycle that we detect and that is not resolved by the use of the algorithm on strongly connected graphs is stored in  $S$ . If there are unresolved cycles, that means that we are in a strongly connected part of the tree, and we must build the graph representing this part. It is the role of the graph  $G$ .  $N$  is the node that is associated with the subtree  $t$ . If there is no unresolved cycle, this node is the final one.

When we identify an independent subgraph in the representation of the tree, we call `representCycle`. The first task of this algorithm is to verify that the graph has not been encountered yet. This is performed using the `treeKey`, `shareWithDone` and `share` procedures. Finding out whether a given graph has already been encountered is quite a difficult task. It is performed in four steps. First, we verify, using the `treeKey` algorithm, whether the graph in its form has already been encountered. If it is the case, it is stored in  $D_G$ , and we can retrieve it from this dictionary. If not, we must then fold the cycles in the graph in order to have a normal form. There are two sources of cycle unfolding: a cycle can be unfolded by cycle growth, or by root unfolding. For example, consider the cycle

 is an example of cycle growth, and  $a \rightarrow b \rightarrow a$  is an example of

root unfolding. The second step determines whether there is root unfolding. It is performed using the `shareWithDone` algorithm. If this step was not successful, we must go to the third step. This third step uses the `share` procedure to reduce cycle growth, and then, the graph being in normal form, we use the `treeKey`

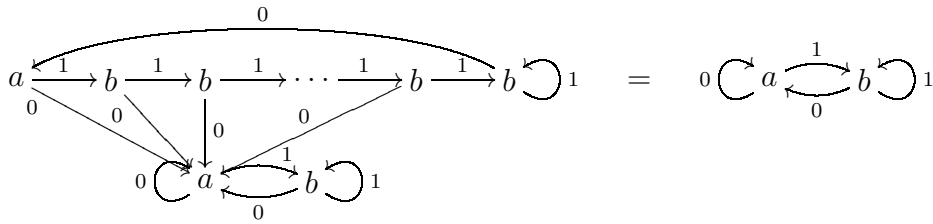
algorithm again.

$\mathbf{representation}(t)$  is the node representing  $t$  in the graph defined by  $D$ . It is easy to see that this graph is of maximal sharing, assuming there is no duplication of strongly connected graph representation. If there are  $p_1$  and  $p_2$  such that  $t_{[p_1]} = t_{[p_2]}$ , if  $p_1 \prec p_2$  then when  $t_{[p_2]}$  is reached, it is already present in  $T$ , and so no new node is created. If the paths don't compare for  $\prec$ , then for example  $p_1 \prec_{\mathbb{N}}^* p_2$  and the nodes for  $t_{[p_1]}$  and all its subtrees are already indexed in  $D$ , thus no new node is added in  $D$  when the key for  $t_{[p_2]}$  is computed.

### Algorithm to Avoid Duplication of Strongly Connected Subgraphs

As we described in the global algorithm remarks, avoiding duplication of strongly connected subgraph is performed in four steps. The first step<sup>2</sup> using  $\mathbf{treeKey}$  is performed to avoid unnecessary work before any transformation of the strongly connected subgraph. The unnecessary work is avoided only if the same node of the same strongly connected subgraph have already been encountered. The second use of this algorithm<sup>3</sup>, performed on a normal form of the graph, decides whether the graph has already been encountered. This is because the first time a graph is constructed, every key of every node of the graph in normal form is computed and associated via  $D_G$  with its unique representation. We could not store every key for any node in any form of the graph (and so the first use of  $\mathbf{treeKey}$  could not decide whether the graph had already been encountered in an other form) because the number of such forms is infinite. It is obvious when considering cycle growth. But even when considering root unfolding, the number of tree keys can be infinite too, because the root itself can be a cycle.

**Example:** Root unfolding with a strongly connected root that can be as big as we want, even after cycle growth reduction.



◇

The order in which we fold the subgraph is important. It is a consequence of the  $\mathbf{represent}$  algorithm. In the case of root unfolding, the cycle of this unfolding has already been treated by the algorithm, and so it is uniquely represented. All

<sup>2</sup>first lines of  $\mathbf{representCycle}$ .

<sup>3</sup>lines 10 to 13 of  $\mathbf{representCycle}$ .



we have to do, then, is to return the node in this cycle that is equivalent to the node we are considering. There is no need to check for cycle growth. If there is no root unfolding, then we can reduce cycle growth with the `share` algorithm, and have a graph in normal form. If later on we come to a node of an equivalent graph with no root unfolding, we will find the same tree key, because the partial values associated with the nodes will be the same due to the uniqueness of the representation.

The algorithm determining whether there is root unfolding is based on some properties of the graph to reduce the complexity of the computation. The basic idea is to compare every node of the graph we consider with every node that is reachable, and already computed, from this graph. These nodes are accessed through the partial keys labeling the graph. As in the algorithm, we call  $G$  the graph we are considering.  $H$  is the graph already computed and stored in  $D$  that is reachable from the partial keys of  $G$ . `shareWithDone( $N$ )` determines whether a node  $N$  of  $G$  is equivalent to a node of  $H$ . If it is the case, then there is root unfolding. If not, there is no root unfolding. It is enough to verify this property for one node to treat the entire graph  $G$  because  $G$  is strongly connected. Suppose  $N$  is equivalent to  $M$  in  $H$ . Then, whatever the legible path  $p$ ,  $N.p$  is equivalent to  $M.p$ . Because  $H$  has been treated already, any  $M.p$  is in  $H$ , and because  $G$  is strongly connected, any node of  $G$  is a  $N.p$ .

There is a kind of reciprocal property that is exploited too: for some subsets of  $H^N$ , if no node of the subset is equivalent to a particular node of  $G$ , then they are not equivalent to any node of  $G$ . A subset of  $H^N$  is said to be closed if and only if, for every legible path  $p$ , for every node  $N$  in the subset,  $N.p$  is in the subset.

**Property 3.2**  $\forall F \subset H^N$  such that  $F$  is closed, if  $\exists N \in G^N$  such that  $\forall M \in F$ ,  $N$  and  $M$  represent different trees, then it is the case whatever  $N \in G^N$ .

**Proof:** Let  $F$  be such a subset and  $N$  a node of  $G$ . If  $N$  is not equivalent to any node in  $F$ , then, suppose there is a  $M \in G^N$  and a  $O \in F$  such that  $M$  is equivalent to  $O$ . As  $G$  is strongly connected, there is a  $p$  such that  $M.p = N$ . So,  $N$  would be equivalent to  $O.p$ , which is in  $F$ . This proves that no element of  $G^N$  is equivalent to any element of  $F$ .  $\square$

Because of these properties, we can compare only the nodes of  $G$  with the nodes that are reachable from their partial keys and not already encountered. That is what `shareWithDone` does, calling `tryShare` for every node in  $G$ . The nodes in  $G$  that have already been visited are stored in  $D_S$ . `tryShare` takes the node  $N$  of  $G$  to be compared with the node  $M$  of  $H$ , and the label  $i$  and  $N.i$ . These last two arguments serve to avoid too much calls to `compareWithDone`. We exploit the property that, as  $N.i$  is in  $H^N$  and the representations in  $H$  are

unique, if  $N$  is equivalent to  $M$ , then  $N.i = M.i$ . The nodes of  $H$  that have already been visited are stored in  $D_D$ . We know by the property proved above that no element of  $D_D$  is equivalent to a node of  $G$ .

When the equality is possible, `tryShare` uses `compareWithDone` to compare the two nodes. The first node is in  $G$  and the second one in  $H$ . Because the representation in  $H$  is unique, a given node of  $G$  can be equivalent to at most one element of  $H^N$ . Thus we can use a dictionary ( $D_C$ ) to store the pairs of already compared nodes:  $\{N, M\}$  with  $N \in G^N$  and  $M \in H^N$  has already been compared if and only if  $D_C(N) = M$ . To perform the comparison, we have to extend the partial key labeling  $N$ , but the nodes which come from the partial key are in  $H^N$ , and so they can be directly compared to  $M$ , because of the uniqueness of the representation in  $H$ . If `compareWithDone` is `true`, then every node of  $G$  is mapped in  $D_C$  to a node of  $H$  to which it is equivalent.  $D_S$  takes the value of  $D_C$ , so that the node which is equivalent to the initial node with which `shareWithDone` was called can be easily found.

This algorithm leads to a quadratic worst case complexity. An alternative would be to use the classical partitioning Hopcroft algorithm on the whole graph, including the already shared part. This algorithm gives an  $n \log(n)$  worst case complexity. But as for `eqTree`, the worst case is rare, and in practice, the algorithm presented here seems to work faster. A possible explanation lies in the fact that the partitioning algorithm is badly suited to take advantage of the parts of the graph that are already shared. The algorithm presented here is more incremental.

### 3.3.4 Proof of the Algorithm

The algorithm returns the node of a graph. We must prove that this graph represents the good tree, and that it is a graph of maximal sharing.

#### Equivalence and Partial Keys

During its execution, the algorithm manipulates graphs labeled by partial keys. In order to show the correctness of the result, we must give the exact meaning of these graphs, and in particular define the corresponding equivalence. This equivalence, denoted  $\equiv_{pk}$  shows when two graphs represent the same tree.

The idea of partial keys is that their name corresponds to the label of the tree, and the partial function to subtrees which have already been treated by the algorithm. So we have a kind of a progression ordering on keys:  $k_1 \leq_{pk} k_2 \Leftrightarrow \text{name}(k_1) = \text{name}(k_2)$  and  $\forall i \in \text{dom}(k_1), k_2(i) = k_1(i)$ .

The partial keys can be forgotten so that we give an equivalent graph labeled on  $F$ , and with no notion of progression. Considering a graph  $G$  labeled by a set of partial keys  $K_G$ , the new graph  $G'$  is defined as:  $G'^N = G^N \cup$  the nodes of the graphs in  $K_G$ , and  $G'^E = G'^E \cup \{(N, i, M) \mid N \in G^N \text{ labeled by } k \text{ and}$

$k(i) = M \cup$  the edges of the graphs in  $K_G$ . The equivalence with partial keys is defined as the transitive and symmetric closure of  $\equiv_t$  augmented by the  $G \equiv_{pk} G'$ .

### Correctness

The property proving the correctness is the following: if **represent**( $t, T \cup \{t\}$ ) returns  $(N, \emptyset, \emptyset^G)$ , then  $N$  represents  $t$ . Considering that we already showed earlier that the algorithm terminates, we will prove it by induction on the recursive calls to **represent**. The three first return points in **represent** (lines 11, 15 and 26) fall into the property, whereas for the fourth one (line 27), the function  $S$  is not empty. So we need only prove the property on the three first return points.

Lines 11 and 15, the partial function  $S$  is empty, so the test line 6 have always been false, and each call to **represent** returned a value of the form  $(N_i, \emptyset, \emptyset^G)$ , so by induction, each  $N_i$  represent correctly  $t_i$ . The line 11 corresponds to a case which have already been encountered. Line 15, the node  $N$  is linked to the different  $N_i$ , so it is a correct representation of  $t$ .

Line 26 depends on **representCycle**, which has four return point. The property of **representCycle**( $N$ ) is that it returns a node  $M$  of a graph with no partial keys and such that  $N \equiv_{pk} M$ . The first return point, line 2, supposes that there is a  $M$  such that  $M \equiv_{pk} N$  which has already been treated.  $M \equiv_{pk} N$  implies  $M \equiv_t N$ , so it is correct to return the same result as for  $M$ . The second return point, line 7, supposes that **shareWithDone** has detected a node  $M = D_S(N)$  which represents the same tree as  $N$  and which is already a unique representation of the tree. So returning  $M$  is correct. The third return point, line 13, comes after a call to **share**, so  $N$  is now the node of another graph which is equivalent (for  $\equiv_t$ ) to the initial one. The third return point supposes that there have already been a node  $M$  such that  $M \equiv_{pk} N$  which has already been encountered, so it is correct to return the same result as for  $M$ . The last return point returns  $N$ , which is of course correct.

The last point to make the link between **representCycle** and **represent** is to prove that line 26 of **represent**, the node  $N$  represents  $t$ . When we call **representCycle**, the partial function  $S$  is empty. It means that every bit of information stored line 6 have been used and translated in the graph  $G$  (lines 22–25). The nodes in the partial keys come from calls to **represent** with return value  $(M, \emptyset, \emptyset^G)$  and so are correct by hypothesis. The rest of the graph is directly derived from the tree.

### Dictionaries and Maximal Sharing

In order to ensure the uniqueness of the representation, we use two dictionaries:  $D$  and  $D_G$ . For every new node created which definitely represents a tree, an entry in  $D$  is created with for a key the label of the node and the nodes of its children (line 14 of **represent**, and line 19 of **representCycle**). The problem is

that this kind of key is not valid for  $\equiv_t$  with infinite trees. So the dictionary  $D_G$  stores the key obtained by `treeKey` for every node which definitely represents a tree, and which is in a loop of the graph (line 21 of `representCycle`).

When we use the results stored in the dictionaries, we don't create any new node, so we cannot duplicate anything. We just have to see that the new nodes effectively created do not duplicate any other one stored in  $D$ . We go through the different opportunities for duplication, and suppose it is the first time a duplication occurs. If no such first time is possible, then no duplication ever occurs.

Line 14 of `represent`, we know that the key  $k$  is not in  $D$ . Moreover, each one of the  $N_i$  composing the key is unique by hypothesis. So if a tree  $\begin{matrix} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{matrix}$  had already been encountered, the key  $(f, (N_i)_{i < n})$  would already have been encountered.

Line 22 of `representCycle`, we know that the key `treeKey(share(N))` has never been encountered before. Because such a key is valid for strongly connected graphs, it means that no other node  $M$  such that  $M \equiv_t N$  have been encountered before. But the problem is that we have a partial key semantics on these graphs, and  $\equiv_t \subset \equiv_{pk}$ , so we could have  $M \not\equiv_t N$  but  $M \equiv_{pk} N$  in effect representing the same tree. Because  $M \not\equiv_t N$ , there is a path  $p$  such that  $M.p$  and  $N.p$  don't have the same label,  $k_M$  and  $k_N$ . But as  $N$  and  $M$  represent the same tree,  $k_M$  and  $k_N$  must have the same name, so their only possible difference is in the partial function. It mean there is an  $i$  such that one of the keys is defined on  $i$  and not the other key (if both of them were defined on  $i$ , their value would be the same on  $i$ , as the nodes in partial keys are unique representations). By construction, the nodes  $M$  and  $N$  are in strongly connected graphs. So if one of the keys is not defined on  $i$ , there is a  $q$  such that  $M.piq = M$  or  $N.piq = N$ . If  $t$  is the tree represented by both nodes, it means that  $t_{[piq]} = t$ . Suppose  $k_M$  is defined on  $i$ , then there is a node reachable from  $k_M(i)$  which represents the same tree as  $M$ , and as such it would have been found by `shareWithDone`. So the graph defined by  $M$  would never have gone beyond the line 7 of `representCycle`. It means that another representative is stored for the cycle (we go on like this until we find one which is equivalent to  $N$ , which means that the test line 11 could not have been false). If  $k_N$  is defined on  $i$ , by the same argument, we could not have been beyond the line 7, and so no new node is created.

### 3.3.5 Example

We present the algorithm to represent regular trees on an example. Let us consider the graph of figure 3.1, where each node is assigned a number. We will write  $t_i$  for the tree represented by the node  $\square_i$ . We will describe each call to `represent` on each  $t_i$ . The return result of this algorithm,  $(N, S, G)$  are described

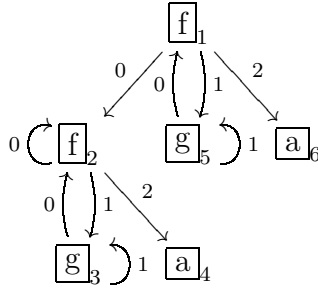
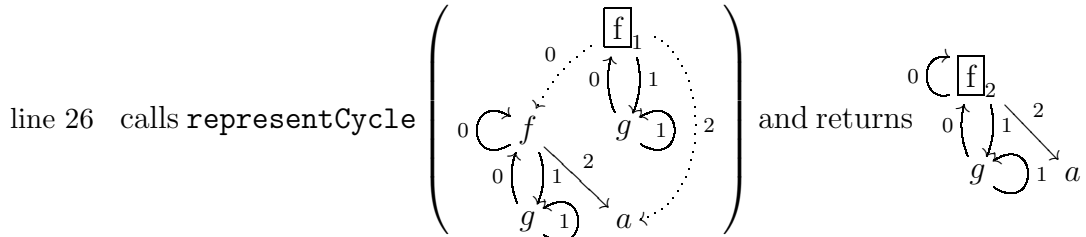
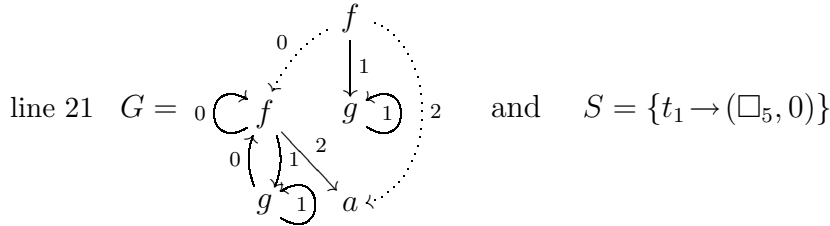


Figure 3.1: Example

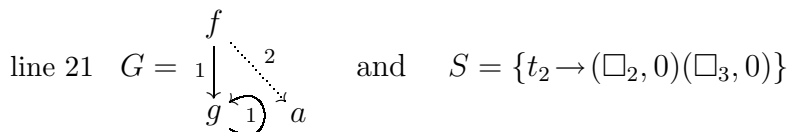
by a framed node  $N$  in a graph and possibly a finite function. If there is no such function, it means that the domain of  $S$  is empty and the graph  $G$  is empty. If there is such a function, then  $G$  is the graph containing  $N$  and  $S$  is the function.

In some graphs, we will draw partial keys. In these cases, the actual edges of the graph are represented by plain lines, and the nodes which are linked by the partial keys are associated with those keys by dotted lines.

`represent( $t_1, \{t_1\}$ ):`



`represent( $t_2, \{t_1, t_2\}$ ):`



line 26 calls `representCycle`  $\left( \begin{array}{c} \textcircled{0} \text{ } \boxed{f}_2 \\ \uparrow \quad \searrow^2 \\ 0 \quad 1 \quad a \\ \downarrow \quad \downarrow \\ g \quad 1 \end{array} \right)$  and returns  $\begin{array}{c} \textcircled{0} \text{ } \boxed{f}_2 \\ \uparrow \quad \searrow^2 \\ 0 \quad 1 \quad a \\ \downarrow \quad \downarrow \\ g \quad 1 \end{array}$

`represent`( $t_3, \{t_1, t_2, t_3\}$ ):

line 21  $G = g$  and  $S = \{t_2 \rightarrow (\square_3, 0), t_3 \rightarrow (\square_3, 1)\}$

line 27 returns  $\boxed{g}_3 \curvearrowright_1$  and  $\{t_2 \rightarrow (\square_3, 0)\}$

`represent`( $t_4, \{t_1, t_2, t_4\}$ ):

line 14  $D$  becomes  $\{a \rightarrow \square_4\}$

line 15 returns  $\boxed{a}_4$

`represent`( $t_5, \{t_1, t_5\}$ ):

line 21  $G = g$  and  $S = \{t_1 \rightarrow (\square_5, 0), t_5 \rightarrow (\square_5, 1)\}$

line 27 returns  $\boxed{g}_5 \curvearrowright_1$  and  $\{t_1 \rightarrow (\square_5, 0)\}$

`represent`( $t_6, \{t_1, t_6\}$ ):

line 11 returns  $\boxed{a}_4$

`representCycle`  $\left( \begin{array}{c} \textcircled{0} \text{ } \boxed{f}_2 \\ \uparrow \quad \searrow^2 \\ 0 \quad 1 \quad a \\ \downarrow \quad \downarrow \\ g \quad 1 \end{array} \right)$ :

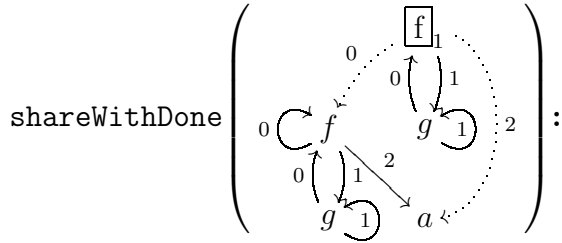
line 4 `shareWithDone` returns false, because the only node already in a partial key is  $\boxed{a}_4$ , and it is not equivalent to any node in the graph.

line 9 `share` does not modify the graph which is already of maximal sharing.

line 22 before returning,  $D_G$  and  $D$  have been modified.

$$D_G \text{ is now } \left\{ \begin{array}{l} \begin{array}{c} f \\ \swarrow \downarrow \searrow \\ \varepsilon \quad g \quad a \\ \downarrow \downarrow \\ \varepsilon \quad 1 \end{array} \rightarrow \square_2, \quad \begin{array}{c} g \\ \swarrow \downarrow \searrow \\ f \quad \varepsilon \\ \downarrow \downarrow \downarrow \\ 0 \quad \varepsilon \quad a \end{array} \rightarrow \square_3 \end{array} \right\}$$

$$\text{and } D \text{ is } \left\{ \begin{array}{l} \begin{array}{c} f \\ \swarrow \downarrow \searrow \\ \square_2 \quad \square_3 \quad \square_4 \end{array} \rightarrow \square_2, \quad \begin{array}{c} g \\ \swarrow \downarrow \searrow \\ \square_2 \quad \square_3 \end{array} \rightarrow \square_3, \quad a \rightarrow \square_4 \end{array} \right\}$$



line 4 calls `tryShare(□1, □2, 0, □2)` and returns `true` with  $D_S = \{\square_1 \rightarrow \square_2, \square_5 \rightarrow \square_3\}$  (line 6 of `tryShare`).

### 3.4 Algorithms on Shared Regular Trees

Trees are structures which are designed for algorithms, and usually, algorithms on trees are quite easy to describe. The main problem with shared regular trees, is that when an algorithm transforms a tree, its result must be shared. The second problem, of course, is the possibility for a regular trees to be infinite.

The basic property of shared regular trees, its main quality, lies in the equality testing. Two trees are equal if and only if they are represented by the same node.

#### 3.4.1 Easy Algorithms

##### Subtrees

Considering a tree  $t$  and a path  $p$  of the tree, finding the representation of the subtree  $t_{[p]}$  is nearly immediate. We just have to follow the graph representing  $t$  according to  $p$ , and the resulting node is the representation of  $t_{[p]}$  with maximal sharing. This is because the property of being of maximal sharing is a global property of the graph and does not depend on the particular node that represents either  $t$  or  $t_{[p]}$ .

Determining whether a tree  $t_1$  is a subtree of  $t_2$  is easy too. We just have to compare the node representing  $t_1$  with every node in  $G_{t_2}$ .  $t_1$  is a subtree of  $t_2$  if and only if it is represented by a node of  $G_{t_2}$ .

##### Root Construction

Considering a label  $f$  and  $Ar_F(f)$  trees  $(t_i)_{i < n}$  represented by the nodes  $(N_i)_{i < n}$ , we want to represent the tree  $\begin{matrix} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{matrix}$ . The dictionary is called  $D$ . If the key

$f, (N_i)_{i < n}$  is in  $D$ , then the corresponding node represents  $\begin{matrix} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{matrix}$  and is sharing

(because, as we said above, any node of the dictionary represents a tree with maximal sharing). If the key is not in the dictionary, then there is no node representing this tree, and so we must use a new node and associate the key

with the new node in the dictionary. Again, this node represents the tree with maximal sharing, without any additional work.

### 3.4.2 Generic Memoizing Algorithms

An algorithm is memoizing [Mic68] if it keeps a private dictionary mapping previous arguments to the results of the computation on them. Because in the case of shared regular trees equality testing is so easy, memoizing is well adapted, and it is the best technique to exploit this structure.

#### Tree Map

We present a schematic generic algorithm using memoizing on a simple tree. This algorithm, `treeMap`, applies a function  $\mathcal{F}$  to a shared regular tree. We require that  $\mathcal{F}$  be a function that takes a label  $f$  and  $Ar_F(f)$  values and gives a value. Moreover, there should be a tractable fixpoint computation strategy of  $\mathcal{F}$ . We define a fixpoint equation  $Eq$  as being of the form:

$$Eq = \begin{cases} X_1 & = \mathcal{F}(f_1, U_1^0, \dots, U_1^{n_1-1}) \\ & \vdots \\ X_m & = \mathcal{F}(f_m, U_m^0, \dots, U_m^{n_m-1}) \end{cases}$$

where  $X_i$  are variables and  $U_i^j$  are either a value or one of the variables. A fixpoint computation strategy of  $\mathcal{F}$  is a function  $fixpoint_{\mathcal{F}}$  from fixpoint equations to vectors of values such that  $fixpoint_{\mathcal{F}}(Eq)$  is a solution of  $Eq$ . Usually, the complexity of the fixpoint algorithm is more than linear. The tree map algorithm decompose the regular tree so that the fixpoint algorithm is applied to smaller parts of the trees, and as few times as possible.

The algorithm builds fixpoint equations through the use of partial values. A partial value is either a value, `InCycle` or `InEvaluation`. The current state of the computation is stored in the dictionary  $D_{mem}$  which links some nodes to a partial value. The current state of the computation defines the fixpoint equation with equations  $X_N = \mathcal{F}(f_N, U_N^0, \dots, U_N^{n_N-1})$ ,  $N$  being nodes of the graph labeled

by  $\begin{matrix} & f_N & \\ \swarrow & & \searrow \\ N_0 & \dots & N_{n_N-1} \end{matrix}$  and  $U_N^i$  is  $D_{mem}(N_i)$  if it is a value, and  $X_{N_i}$  otherwise. This

fixpoint equation defines a relation between nodes of the graph:  $M \leftarrow_{D_{mem}} N$  if  $X_N$  depends on  $X_M$ . The fixpoint computation strategy applied to a node  $N$  is considered as restricted to those variables  $X_M$  that are reachable from  $X_N$  in the fixpoint equation. We will write  $fixpoint_{\mathcal{F}}(N)$  for the family of values indexed over the set of nodes  $M$  such that  $M \leftarrow_{D_{mem}} N$  solution of this fixpoint equation.

`treeMap`( $N, \mathcal{F}$ ):

$N$  is labeled by  $\begin{matrix} & f & \\ \swarrow & & \searrow \\ N_0 & \dots & N_{n-1} \end{matrix}$



```

add  $N \rightarrow \text{InEvaluation}$  to  $D_{mem}$ 
for  $i \leftarrow 0$  to  $n - 1$  do
  if  $N_i \in \text{dom}(D_{mem})$  then
     $V_i \leftarrow D_{mem}(N_i)$ 
    if  $V_i = \text{InEvaluation}$  then  $\text{Cycle}_i \leftarrow \{N_i\}$  else  $\text{Cycle}_i \leftarrow \emptyset$ 
    else  $(V_i, \text{Cycle}_i) \leftarrow \text{treeMap}(N_i, \mathcal{F})$ 
 $\text{Cycle} \leftarrow \bigcup_{i < n} \text{Cycle}_i$ 
if every  $V_i$  is a value then
   $V \leftarrow \mathcal{F}(f, V_0, \dots, V_{n-1})$ 
  replace the value of  $N$  by  $V$  in  $D_{mem}$ 
  return $(V, \emptyset)$ 
else if  $\text{Cycle} = \{N\}$  then
   $(V_M)_{M \leftarrow_{D_{mem}} N} \leftarrow \text{fixpoint}_{\mathcal{F}}(N)$ 
  replace the value of every  $M$  such that  $M \leftarrow_{D_{mem}} N$  by  $V_M$  in  $D_{mem}$ 
  return $(V_N, \emptyset)$ 
else
  replace the value of  $N$  by  $\text{InCycle}$  in  $D_{mem}$ 
  return $(\text{InCycle}, \text{Cycle} \setminus \{N\})$ 

```

### Tree Substitution

Tree substitution consists in replacing any subtree labeled by a given label  $x$  by a tree  $t$  given by the substitution. It is a typical example of memoizing algorithm, and it can be expressed using `treeMap` and the function  $\mathcal{F}$  defined as:

$$\mathcal{F}(x, N_0, \dots, N_{n-1}) = N, \text{ the node representing } t$$

$$\mathcal{F}(f, N_0, \dots, N_{n-1}) = \text{the node labeled by } \underset{N_0 \dots N_{n-1}}{\underset{\vee}{\vee}}^f \text{ if } f \neq x$$

The fixpoint equations of  $\mathcal{F}$  are solved using the algorithm presented in `represent`.

Tree substitution is one of the basic operations on trees. It is the reason why we give the description of its algorithm. As the algorithm is not very different from the `representation` algorithm, we first give the simple modifications to give to this algorithm to obtain the tree substitution.

The only necessary modification is a test at the beginning of the algorithm to see whether the label of the current tree is  $x$ . If it is the case, we can return the substitute tree with an empty graph. Because the `representation` algorithm does not require that we always know when two subtrees are equal, no more modification is needed to compute the tree substitution.

In addition, we can use another optimization: because the tree is already shared, if a subtree does not contain  $x$ , then it does not need to be changed. It is an information that is easy to propagate while exploring the tree to perform

the substitution. We add a new result to **represent**, a boolean value  $b$ . If we come to an  $x$ ,  $b$  is **true**. Else,  $b$  is **true** if and only if the result of **represent** on one of its children is **true**. Children of  $t$  that are in  $T$  don't participate to the computation of this value. As soon as  $b$  and  $S$  are computed<sup>4</sup>, we can exploit them: if  $b$  is **false** and  $\text{dom}(S) \subset \{t\}$ , then there is no need to modify the subtree  $t$ . In this way, we avoid the possible computation of the strongly connected graph containing  $t$  and the painful algorithm to avoid duplication of the graph.

**treeSubst**( $t_1, x, t_2$ ):

```
(N, S, G, b) ← treeSubstRec( $t_1, \{t_1\}, x, t_2$ )
return(N)
```

**treeSubstRec** $\left(\begin{array}{c} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}, T, x, u\right)$ :

```
if  $f = x$  then return( $u, \emptyset, \emptyset^G, \mathbf{true}$ )
```

```
 $t \leftarrow \begin{array}{c} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}$ 
```

$k$  is the partial key such that  $\text{name}(k) = f$  and  $\text{dom}(k) = \emptyset$

$S$  is a partial function from trees to sets of nodes and numbers and  $\text{dom}(S) = \emptyset$

$N$  is a new node

```
 $b \leftarrow \mathbf{false}$ 
```

```
for  $i \leftarrow 0$  to  $n - 1$  do
```

```
  if  $t_i \in T$  then add  $t_i \rightarrow (N, i)$  to  $S$ 
```

```
  else  $(N_i, S_i, G_i, b_i) \leftarrow \mathbf{treeSubstRec}(t_i, T \cup \{t_i\}, x, u)$ 
```

```
     $b \leftarrow b \vee b_i$ 
```

```
    if  $\text{dom}(S_i) = \emptyset$  then add  $i \rightarrow N_i$  to  $k$ 
```

```
    else add  $S_i$  to  $S$ 
```

```
if  $b = \mathbf{false}$  and  $\text{dom}(S) \subset \{t\}$  then return( $t, \emptyset, \emptyset^G, \mathbf{false}$ )
```

```
if  $\text{dom}(S) = \emptyset$  then
```

```
  if  $k \in \text{dom}(D)$  then return( $D(k), S, \emptyset^G, \mathbf{true}$ )
```

```
  else
```

```
    Let  $N$  be labeled by  $f$  and  $N.i \leftarrow N_i$ 
```

```
    add  $k \rightarrow N$  to  $D$ 
```

```
    return( $N, S, \emptyset^G, \mathbf{true}$ )
```

```
else
```

```
   $G^N \leftarrow \bigcup_{i < n} G_i^N$ 
```

```
   $G^E \leftarrow \bigcup_{i < n} G_i^E$ 
```

```
  add  $N$  labeled by  $k$  to  $G^N$ 
```

```
  add  $\{(N, N_i, i) \mid \text{dom}(S_i) \neq \emptyset\}$  to  $G^E$ 
```

```
  if  $t \in \text{dom}(S)$  then
```

```
    while  $S(t) \neq \emptyset$  do
```

---

<sup>4</sup>just after line 9 of **represent**.

```

    take  $(M, i)$  out of  $S(t)$ 
    add  $(M, N, i)$  to  $G^E$ 
    delete  $t$  from  $dom(S)$ 
if  $dom(S) = \emptyset$  then return(representCycle( $N$ ),  $S, \emptyset^G$ , true)
else return( $N, S, G, b$ )

```

### 3.4.3 Recursive Construction

Recursive construction is the basic operation to build infinite looping trees. The idea is to solve the equation  $x = t$  where  $x$  is a label of arity zero which may appear in  $t$ , but  $t(\varepsilon) \neq x$ . Formally, let  $u = \text{recCons}(t, x)$ . the domain of  $u$  is defined by:

$$pos(u) = \left\{ p \mid \exists (p_i)_{i \leq n} \in pos(t), p = \bigcirc_{i \leq n}^{<} p_i \text{ and } \forall i < n, t(p_i) = x \right\}$$

The values of  $u$  on its domain are defined by a simple recursion:

$$u(p) = \text{if } (p = qp_1) \text{ and } (t(q) = x) \text{ then } u(p_1) \text{ else } t(p)$$

As recursive construction is a basic algorithm, it is based on the **representation** algorithm. As for tree substitution, the algorithm does not change subtrees that do not contain  $x$ .

In **representation**, we call **represent** with the set  $\{t, x\}$  instead of  $\{t\}$ .

The test to decide whether the current subtree can be left unchanged is performed just after the line 9 of **represent**. The subtree can be left unchanged if  $dom(S) \subset \{t\}$ . It corresponds exactly to the test of line 26. So there is no need to call **representCycle** in **represent**. We need only one call to **representCycle**, in **representation**, after modifying the resulting graph of **represent** according to  $S$ , adding the edges corresponding to the different occurrences of  $x$  in  $t$ .

Using root construction and recursive construction, one can build any regular tree.

```

recCons( $t, x$ ):
  ( $N, S, G$ )  $\leftarrow$  recConsRec( $t, \{t, x\}$ )
  if  $dom(S) = \emptyset$  then return( $N$ )
  while  $S(x) \neq \emptyset$  do
    take  $(M, i)$  out of  $S(x)$ 
    add  $(M, N, i)$  to  $G^E$ 
  return(representCycle( $N$ ))

```

```

recConsRec  $\left( \begin{array}{c} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}, T \right)$ :

```

```

 $t \leftarrow \begin{array}{c} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}$ 

```

$k$  is the partial key such that  $\mathbf{name}(k) = f$  and  $\mathbf{dom}(k) = \emptyset$   
 $S$  is a partial function from trees to sets of nodes and numbers and  $\mathbf{dom}(S) = \emptyset$   
 $N$  is a new node  
**for**  $i \leftarrow 0$  **to**  $n - 1$  **do**  
    **if**  $t_i \in T$  **then** add  $t_i \rightarrow (N, i)$  to  $S$   
    **else**  $(N_i, S_i, G_i) \leftarrow \mathbf{recConsRec}(t_i, T \cup \{t_i\})$   
        **if**  $\mathbf{dom}(S_i) = \emptyset$  **then** add  $i \rightarrow N_i$  to  $k$   
        **else** add  $S_i$  to  $S$   
**if**  $\mathbf{dom}(S) \subset \{t\}$  **then return**  $(t, \emptyset, \emptyset^G)$   
 $G^N \leftarrow \bigcup_{i < n} G_i^N$   
 $G^E \leftarrow \bigcup_{i < n} G_i^E$   
add  $N$  labeled by  $k$  to  $G^N$   
add  $\{(N, N_i, i) \mid \mathbf{dom}(S_i) \neq \emptyset\}$  to  $G^E$   
**if**  $t \in \mathbf{dom}(S)$  **then**  
    **while**  $S(t) \neq \emptyset$  **do**  
        take  $(M, i)$  out of  $S(t)$   
        add  $(M, N, i)$  to  $G^E$   
        delete  $t$  from  $\mathbf{dom}(S)$   
**return**  $(N, S, G)$

### 3.4.4 Complexity Issues

Algorithms on shared regular trees are more difficult than standard algorithms on regular trees. Comparing complexities of algorithms on the two representations is difficult, though. The complexity is measured with respect to the size of the inputs of the algorithms, which can be reduced to the number of distinct subtrees of the inputs in our case. In the case of shared regular trees, the number of distinct subtrees is exactly the number of different subtrees of the tree, but when the tree is not shared, the number of subtrees that are distinguished by the representation can be of any value greater than the number of different subtrees. In the sequel, we denote by  $n$  this number, but we must keep in mind that this  $n$  can be much bigger in the case of non-shared trees.

#### Summary of worst case time complexities

	sharing representation	naive representation
testing $t_1 = t_2$	$\mathcal{O}(1)$	$\mathcal{O}((n_1 + n_2) \log(n_1 + n_2))$
testing $t_1 < t_2$	$\mathcal{O}(n_2)$	$\mathcal{O}((n_1 + n_2) \log(n_1 + n_2))$
building $t_{[p]}$	$\mathcal{O}( p )$	$\mathcal{O}( p )$
root construction	$\mathcal{O}(1)$	$\mathcal{O}(1)$
recursive construction	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
tree substitution: $(t_1, x, t_2)$	$\mathcal{O}(n_1 n_2)$	$\mathcal{O}(n_1)$

In the case of equality testing with a non sharing representation, we must perform the `eqTree` algorithm and in the worst case compare any pair of distinguished subtrees<sup>5</sup>. This would lead to a quadratic worst case complexity. However, we can always use the partitioning algorithm of Hopcroft [Hop71] to get an  $n \log(n)$  complexity in the worst case. Although this algorithm seems less efficient in practice for equality testing, we assume the best possible bound for non-sharing representations.

For recursive construction, the prevailing operation seems to be the final `share` algorithm. In the worst case, though, the quadratic complexity of `shareWithDone` will take precedence.

For tree substitution, it is the possibly numerous calls to `shareWithDone` which can be the most costly. Remember that the worst case complexity seems rare in practice for this algorithm.

This summary suggests that if we are to perform equality testing, it can be beneficial to perform sharing during the calculus. What we show here are worst case complexity, though, and the difficult cases are quite pathological, and thanks to some simple optimizations, they are quite rare. The situation is quite similar to the complexity of operations on BDDs [Bry86] compared to the operations on boolean formulas. The size of the formula representing a given boolean function is unbounded, but the basic operations, like conjunctions are linear in the size of one of the formulas but the product of the size of the BDDs.

## Conclusion

The series of basic algorithms we presented in this chapter allows the construction and the manipulation of regular trees while preserving property of maximal sharing for the representation of the trees. To maintain this property, constructing algorithms are more complex than traditional algorithms that cannot share any subtree because of their inability to deal with sharing of cyclic structures. The tradeoff for this increased complexity is a bounded compact representation of a regular tree, whereas in the classical case there is no bound on the size of the representation (due to cycle growth or root unfolding). Moreover, in the presence of memoizing, and everywhere where equality testing is required, these techniques allow in practice a substantial speed-up, especially when the number of operations performed on the trees is high.

---

<sup>5</sup>We cannot use the linear algorithm proposed in [AHU74] because we deal with possibly infinite trees, for which there is no notion of “level” for the nodes.



# Chapter 4

## Representation of Relations

Relations and sets of trees are intimately bound: a set of trees puts in relation subtrees of common prefix path except for the last letter of the path.

For example the set of trees  $\left\{ \begin{array}{c} f \\ \swarrow \searrow \\ a \quad b \end{array}, \begin{array}{c} f \\ \swarrow \searrow \\ a \quad c \end{array}, \begin{array}{c} f \\ \swarrow \searrow \\ b \quad c \end{array} \right\}$  can be seen as the relation  $\{ \langle ab \rangle, \langle ac \rangle, \langle bc \rangle \}$ . The connection between sets of trees and relations can also be emphasized by the study of the prefix regular relations of [AH84]. So relations are used in a way or another in the representation of sets of trees.

The problem is that we want to represent some possibly complex infinite sets of infinite trees, which means that we may need to represent infinite relations. These representations should even reflect the infinite behavior of the relations, distinguishing for example between different kind of fairness. To our knowledge, there is no such representation already available that would be tractable. The second problem attacked in this chapter concerns the possibility of approximating the relations whenever their representations are too big or impossible.

Because different kinds of relations and different kinds of approximation strategies can be adapted to different kinds of problems, we present in this chapter different ways of representing relations that can be useful in the representation of sets of trees. Some of these representations can be more or less efficient depending on the number of entries of the relation, the size of the sets associated with the entries, or the particular shape of the relation.

Thus, the first part of this chapter subsumes definitions and structures that are common to these different possible representations. The second part presents representations of relations based on decision diagrams. This section is focused on the representation of infinite relations and their infinite behavior and is the main contribution of this chapter. The third section presents algorithms to manipulate these possibly infinite relations based on decision graphs. It presents a first way to deal with approximated relations. The last section develops other representations of relations that allow different kinds of approximation. This last section is very short. It is there to show how we can build on the first section to represent different kinds of relations.

## 4.1 Definitions and Common Structures

The relations we want to represent are relations over sets of the form  $[n]$ . Moreover, as the relations bind nodes of a tree, their sets of indexes are subsets of  $\mathbb{N}^*$ . Formally, they are elements of

$$\mathcal{Rel} \stackrel{\text{def}}{=} \left\{ R \mid \exists (n_i)_{i \in I} \in (\mathbb{N} \setminus \{0\})^I, I \subset \mathbb{N}^*, R \subset \bigotimes_{i \in I} [n_i] \right\}$$

If  $I$  is finite, it is identified to  $[|I|]$ , preserving the order  $\prec_{\mathbb{N}}^*$  on  $I$ . In the same way, if  $I$  is infinite, it can be identified to the ordinal<sup>1</sup>  $|I|$ . Thus, elements of the relations can be identified to words of  $\mathbb{N}^*$  or infinite words. This relation between words and vectors is defined inductively by:  $word(\langle \rangle) \stackrel{\text{def}}{=} \varepsilon$ , and if  $e \in \bigotimes_{i < k+1} [n_i]$ ,  $word(e) \stackrel{\text{def}}{=} word(e_{([k])}) e_{(k)}$ . If  $e \in \bigotimes_{i \in \lambda} [n_i]$ , with  $\lambda$  a limit ordinal,  $word(e)$  is the limit of the  $word(e_{(k \in \lambda)})$ . The reverse of this operation is  $\langle . \rangle$ , that is  $\langle word(e) \rangle = e$ .

Each entry of the relation is associated with a name. This name is intended to be used as a way to share some entries of the relations. As a consequence, two entries that are not equivalent cannot be associated with the same name. The name of the entry  $i$  of the relation  $R$  is written  $name_R(i)$ . The set of entry names of  $R$ ,  $\bigcup_{i \in I} name_R(i)$ , is denoted  $ename(R)$ . The projection of  $R$  according to an entry name is defined by  $R_{(name_R(i))} \stackrel{\text{def}}{=} R_{(i)}$ . As we will see, this definition does not depend on the particular choice of  $i$  for a given name.

### 4.1.1 Equivalent Entries

#### Basic Case

Let  $e \in \bigotimes_{i \in I} E_i$ ,  $j, k \in I$  such that  $E_j = E_k$ . The element of  $\bigotimes_{i \in I} E_i$  obtained from  $e$  by exchanging its  $k^{\text{th}}$  and its  $j^{\text{th}}$  components is denoted  $e_{j \leftrightarrow k}$ . Formally,

$$\begin{aligned} \forall i \in I, e_{j \leftrightarrow k(i)} &\stackrel{\text{def}}{=} e_{(i)} \text{ if } i \neq j \wedge i \neq k \\ &\stackrel{\text{def}}{=} e_{(j)} \text{ if } i = k \\ &\stackrel{\text{def}}{=} e_{(k)} \text{ if } i = j \end{aligned}$$

Let  $R$  be a relation. Two entries  $i$  and  $j$  of  $R$  are said to be *equivalent* if and only if:

$$\forall e \in R, e_{i \leftrightarrow j} \in R$$

---

<sup>1</sup>The size of  $\mathbb{N}^*$  is strictly greater than the size of  $\mathbb{N}$ . It means that, to be exact, we must consider transfinite relations. To simplify the presentation, we will restrict transfinite relations to relations that can be described by a mere infinite relation in the following way: an element of the transfinite relation is in the relation if and only if its first  $\omega$  elements are in the infinite relation, and so on,  $\omega$  elements by  $\omega$  elements. So, we will mostly speak of infinite relations in the sequel of the chapter.



**Equivalent Vectors of Entries**

We consider the case where, instead of comparing the behavior of a relation on two entries, we compare its behavior on two disjoint sets of entries of the form  $[j, j + n[$  and  $[k, k + n[$ . If  $\forall i < n, E_{j+i} = E_{k+i}$ , we define the exchange of the two vectors of entries on  $e$  by:

$$\begin{aligned} \forall i \in I, e_{[j,j+n[ \leftrightarrow [k,k+n[ (i)} &\stackrel{\text{def}}{=} e_{(i)} \text{ if } i \notin [j, j + n[ \text{ and } i \notin [k, k + n[ \\ &\stackrel{\text{def}}{=} e_{(k+l)} \text{ if } i = j + l \text{ with } l < n \\ &\stackrel{\text{def}}{=} e_{(j+l)} \text{ if } i = k + l \text{ with } l < n \end{aligned}$$

The two vectors of entries are equivalent for  $R$  if  $\forall e \in R, e_{[j,j+n[ \leftrightarrow [k,k+n[} \in R$ . The definition is exactly the same as for single entries, and in fact reasoning about equivalent vectors of entries is the same as reasoning about equivalent entries. Thus, it is possible to give the same name to the corresponding entries of the vectors, if in the vector every entry has a distinct name. As these vectors of entries behave the same as a single entry, though, we will only consider single entries in this chapter, to improve the clarity of the presentation.

**Infinite Sets of Equivalent Entries**

In order to represent infinite relations, we must consider the possibility for infinite sets of entries to share the same name. But in this case, it must make sense even for the infinite behavior of the relation, which is not the case if we just require that any two given entries of the set are equivalent.

An infinite set of equivalent entries can share the same name if for any permutation of the entries<sup>2</sup>, the relation is the same.

**Example:** Consider the relation containing any infinite vector of 0's and 1's such that there is an infinite number of 0's followed by a 0. This relation contains  $(001)^\omega$  but not  $(01)^\omega$ . If we considered finite permutations only, that is permutations generated from finite exchange of entries, then every entry of this relation is equivalent. But there is an infinite permutation that transforms  $(001)^\omega$  to  $(01)^\omega$ :

$$\begin{array}{l} \sigma(3 \times k) = 2 \times k \\ \sigma(3 \times k + 1) = 4 \times k + 1 \\ \sigma(3 \times k + 2) = 4 \times k + 3 \end{array} \quad \begin{array}{cccccccccccccccc} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \dots \\ \left| \begin{array}{c} \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \dots \end{array} \right. & \begin{array}{c} \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \diagdown \\ \diagup \\ \dots \end{array} & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{array}$$

---

<sup>2</sup>A permutation of the entries is a bijection from the set of entries to itself.

The idea is that we should have the possibility to present the values corresponding to entries with the same name in any order. But the meaning of this substitution is that, if a relation with all entries equivalent contains  $(001)^\omega$  then there is a way of giving the values of  $(01)^\omega$  such that it is accepted by the relation too. So, concerning our relation, it is consistent to forbid the equivalence of every entry.  $\diamond$

As an immediate consequence, we have the following properties for every relation  $R$  where all entries are equivalent:

**Property 4.1** *Let  $v$  be a word, and  $b$  a letter in  $v$ . Then  $\langle v^\omega \rangle \in R$  if and only if  $\langle (vb)^\omega \rangle \in R$ .*

**Property 4.2** *Let  $e$  be a word where the letter  $b$  appears infinitely often. Then  $\langle e \rangle \in R$  if and only if  $\langle be \rangle \in R$ .*

**Proof:** In both cases, we have an infinite permutation of the entries that transforms the first vector in the other one. In the second case, we just have to shift the  $b$ 's of  $be$  to the right, each  $b$  going at the entry of the next one. In the first case, we keep shifting by one more  $b$  for each  $vb$ .  $\square$

If the set of indexes of an infinite relation  $R$  is totally ordered, the sequence of the names of its entries forms an infinite word. We say that the entry names of the relation are *ultimately periodic* if the word they form is ultimately periodic. Then the *periodicity* of the entry names of the relation is the least  $k$  such that there is an  $N \in \mathbb{N}$ ,  $\forall i > N$ ,  $\text{name}_R(i) = \text{name}_R(i + k)$ . The periodicity of a relation is the least periodicity of all the possible entry names for that relation.

### 4.1.2 Isomorphic Relations

Let  $R$  and  $S$  be two relations with named entries.  $R$  is a relation on  $\bigotimes_{i \in I} E_i$  and  $S$  a relation on  $\bigotimes_{j \in J} F_j$ . The two relations are said to be *isomorphic*, and we write  $R \simeq S$ , if there is a bijection  $\sigma : I \rightarrow J$  such that:

$$\begin{aligned} \forall i \in I, E_i &= F_{\sigma(i)} \\ \forall i \in I, \text{name}_R(i) &= \text{name}_S(\sigma(i)) \\ \forall e \in \bigotimes_{i \in I} E_i, e \in R &\Leftrightarrow \sigma^*(e) \in S \end{aligned}$$

The extension of  $\sigma$  to  $\bigotimes_{i \in I} E_i \rightarrow \bigotimes_{j \in J} F_j$ ,  $\sigma^*$ , is defined, when  $\forall i \in I$ ,  $E_i = F_{\sigma(i)}$ , by:

$$\forall e \in \bigotimes_{i \in I} E_i, \forall j \in J, \sigma^*(e)_{(\sigma(j))} \stackrel{\text{def}}{=} e_{(j)}$$

In the case where  $I$  and  $J$  are ordered, we require moreover that  $\sigma$  respects the ordering, that is if  $i <_I i'$ ,  $\sigma(i) <_J \sigma(i')$ .

### 4.1.3 Regular Relations

We introduce the notion of *regular relation*, which is used to characterize a class of infinite relations that can be finitely represented, in the same way as regular trees. Remember, though, that we are interested in relations with infinite number of entries, not to be mistaken for infinite binary relations of [GN84] for example. In our paradigm, an infinite relation can be seen as a set of infinite words, so it should be possible to use the notion of  $\omega$ -regularity of Büchi [Büc60, Büc73]. We must be more restrictive, though, as we must deal with entry names in the representation too.

#### Prefix Regular Relations

A relation  $R$  of domain  $\bigotimes_{i \in I} E_i$ , with  $I$  an ordinal, is said to be *prefix regular* if:

$$\exists N \in \mathbb{N}, \forall k > N, \forall e \in \bigotimes_{j < k} E_j, \exists l < l_1 \leq k, R(e_{([l])}) \simeq R(e_{([l_1])})$$

The idea is that the finite part of the relation can be finitely represented. An equivalent way of characterizing a prefix regular relation is the finiteness of the set  $\left\{ R(e) \mid e \in \bigotimes_{j < k} E_j, k \in \mathbb{N} \right\}$  modulo  $\simeq$ .

#### Iterative Relations

Prefix regularity only constrains the finite behavior of the relation. We describe the infinite behavior of the relation using what we call iterative relations. These relations are defined as a special interpretation of open relations.

An *open relation*  $R$  of domain  $\bigotimes_{i \in I} E_i$  is characterized by:

$$\forall e \in R, \exists k \in \mathbb{N}, R(e_{([k])}) = \bigotimes_{i \in I \setminus [k]} E_i$$

Open relations are used to describe the relation between finite parts of an infinite vector.

A relation  $R$  is said to be *iterative* if and only if it is described by a prefix regular open relation  $S$  in the following way:  $e \in R$  if and only if  $e \in S$  and  $\exists (k_i)_{i \in \mathbb{N}}$  such that:

$$k_0 = \min \left\{ k \mid \square(S, [k], e_{([k])}) \text{ and } \text{nameiter}(S, k) \right\}$$

$$\forall i \in \mathbb{N}, \exists k > k_i, \square(S, [k - k_i], e_{([k_i, k])})$$

$$k_{i+1} = \min \{k \mid \square(S, [k - k_i], e_{([k_i, k])}) \text{ and } \text{nameiter}(S, k - k_i)\}$$

Where  $\square(S, J, w)$  is a shortcut for  $S_{:J=w} = \bigotimes_{i \in I \setminus J} E_i$  and  $\text{nameiter}(S, k)$  stands for  $\forall j \in \mathbb{N}, \text{name}_S(j + k) = \text{name}_S(j)$ .

We write  $R = \Omega(S)$ .

Note that the empty relation is open and iterative.

### Examples:

- The relation consisting of the set of every infinite vector beginning with a 0 is an open relation, and the iterative relation it represents contains only the infinite vector of 0's (written  $0^\omega$ ).
- The relation containing any vector with at least one 0 is an open relation, and the iterative relation it represents is the set of vectors with an infinite number of 0's.
- The relation containing any vector with at least one 0 and one 1 is an open relation, and the iterative relation it represents is the set of vectors with an infinite number of 0's and 1's.

◇

**Property 4.3** *An iterative relation is prefix regular.*

**Proof:** Let  $R = \Omega(S)$ , with  $S$  prefix regular. If  $S(u) \simeq S(v)$ , then  $R(u) \simeq R(v)$ . So the set of distinct  $R(u)$  modulo  $\simeq$  is smaller than the set of distinct  $S(u)$  modulo  $\simeq$ , which is finite because  $S$  is prefix regular. Thus,  $R$  is prefix regular. □

### Definition of Regular Relations

A relation  $R$  is  $\omega$ -regular if and only if it is prefix regular and there is a finite set of non-empty iterative relations  $\text{iter}(R)$  such that  $\forall e \in R, \exists k, \exists S \in \text{iter}(R), e_{(\mathbb{C}[k])}$  is in  $S$  and for all  $f$  in  $S, \langle \text{word}(e_{([k])}) \text{word}(f) \rangle$  is in  $R$ .

A relation is *regular* if it is finite or  $\omega$ -regular.

### Properties of $\omega$ -regular Relations

This notion of  $\omega$ -regularity is really closely related to the notion of  $\omega$ -regularity of sets of words, although we don't use exactly the  $\omega$ -regularity of Büchi in regular relations. The link between the two notions can illustrate the expressive power of regular relations. This link is expressed by the following property:

**Property 4.4** *A relation  $R$  is  $\omega$ -regular if and only if its entry names are ultimately periodic and the set  $\{e \mid \langle e \rangle \in R\}$  is  $\omega$ -regular in the sense of Büchi.*

The idea of the presentation of  $\omega$ -regular relations is the use of the characterization of  $\omega$ -regular sets of words as finite unions of sets of the form  $U.V^\omega$  with  $U$  and  $V$  regular sets of finite words [Büc60]. In the case of relations with named entries, the iterative relations can be seen as sets of the form  $V^\omega$ .

**Proof:** Let  $R$  be an  $\omega$ -regular relation. Because it is prefix regular, its entry names are ultimately periodic. Let  $iter(R)$  be  $(R_i)_{i \in \mathbb{C}}$ . Every iterative relation defines an  $\omega$ -regular language: the Büchi automaton that accepts the language defined by an iterative relation represented by the open relation  $S$  is  $(Q, E, \{S\}, \{S\})$  with:

$$\begin{aligned} Q &= \{S(\langle p \rangle) \mid p \in \mathbb{N}^*\} \text{ modulo } \simeq \\ E &= \{(S(\langle p \rangle), j, S_{pj}) \mid j \in [n_{|p|}]\} \\ &\quad \text{where } S_q = \text{if } \square(S, [|q|], \langle q \rangle) \text{ then } S \text{ else } S(\langle q \rangle) \end{aligned}$$

The transition set  $E$  is finite because of the prefix regularity of  $S$ .

$R = \bigcup_{p \in P} \bigcup_{i \in \{j \in \mathbb{C} \mid \forall \langle e \rangle \in R_i, \langle pe \rangle \in R\}} \{\langle pe \rangle \mid \langle e \rangle \in R_i\}$  by definition of the  $\omega$ -regularity of  $R$ . Each set  $\{pe \mid \langle e \rangle \in R_i\}$  is  $\omega$ -regular because  $\omega$ -regularity of words is closed under concatenation, and the number of such sets is finite by prefix regularity of  $R$ . Being a finite union of  $\omega$ -regular languages,  $\{e \mid \langle e \rangle \in R\}$  is  $\omega$ -regular.

Now let  $(Q, E, I, F)$  be a Büchi automaton such that there is a relation  $R$  with  $\{e \mid \langle e \rangle \in R\} = L^\omega(Q, E, I, F)$  and the entry names of  $R$  are ultimately periodic.  $R$  is a relation on  $\bigotimes_{i \in \mathbb{N}} [n_i]$ . Let  $R_q = \{\langle e \rangle \mid e \in L^\omega(Q, E, \{q\}, \{q\})\}$ . Each  $R_q$  is an iterative relation represented by the open relation:

$$S_q = \left\{ e \in \bigotimes_{i \in \mathbb{N}} [n_i] \mid \exists k, \text{word}(e_{([k])}) \in L^*(Q, E, \{q\}, \{q\}) \right\}$$

These relations are prefix regular because the automaton is finite and the entry names are ultimately periodic.  $\forall e \in L^\omega(Q, E, I, F)$ ,  $e$  is the label of an infinite path such that there is a  $q$  in  $F$  and  $q$  is in the set of infinitely repeated states of the path. Thus, there is a  $k$  such that  $\langle e \rangle_{([k])}$  is in  $R_q$ , and for all  $f \in R_q$ ,  $\text{word}(\langle e \rangle_{([k])}) \text{word}(f)$  is in  $L^\omega(Q, E, I, F)$ . So  $R$  is an  $\omega$ -regular relation and  $iter(R) = (R_q)_{q \in F}$ .  $\square$

The notion of  $\omega$ -regularity of relations is a strict restriction of the  $\omega$ -regularity of languages:

**Example:** The set  $\{a, bb\}^\omega$  is an  $\omega$ -regular language, but not an  $\omega$ -regular relation. Suppose two entries  $i < j$  are equivalent. Then  $a^{j-1}bba^\omega$  is in the relation, so  $a^{i-1}ba^{j-i}ba^\omega$  is in the relation too, because of the equivalence of entries  $i$  and  $j$ .  $\diamond$

This strict restriction is necessary, because in the case of relations we have a notion of entry which is of greater importance than the notion of rank in the case of words of a language. We need to have a notion of regularity for the entries. Moreover, this restriction will allow for an important optimization commonly used for finite relations, namely path compression in decision diagrams.

The ultimate periodicity of entry names is preserved by union, intersection and negation of relations. So, we can extend some closure properties of  $\omega$ -regular languages [Büc60]:

**Property 4.5**  *$\omega$ -regular relations are closed under union, intersection and negation.*

## 4.2 Decision Diagrams

There are two ways of representing relations: either as sets of vectors, or using a divide and conquer strategy based on the following observation: if  $R \subset \bigotimes_{i \in I} E_i$ , then for all  $k$  of  $I$ ,

$$e \in R \Leftrightarrow e_{\mathcal{C}\{k\}} \in R(e_{(k)})$$

If, moreover,  $I$  is totally ordered, then it is easy to see that this property can be the basis of a decision procedure.

In this section, we study the representation of relations as decision diagrams. The most general diagram is the multiple decision diagrams (denoted MDD) which is a simple extension of binary decision diagrams [Bry86, Bry92]. This extension, which was already introduced in [SKMB90, CMR92] for finite relations, consists in replacing binary choices at each variable, by choices in  $[n]$  depending on the variables. We start from this decision diagrams because it is well known [BCM<sup>+</sup>90] that it is a very efficient way of representing binary relations that have brought considerable breakthrough to many areas in computer science. Moreover, as this representation have been studied from some time now, some efficient implementations are available [BRB90].

In the case of finite relations, the decision process, based on a decision tree, is quite standard. In the case of infinite relations, though, we must introduce new concepts, that will lead to an extension of BDDs for finite relation. Some work has already been done to represent infinite sets of finite relations [GF93], but this work is not adapted to the representation of infinite relations, in particular it does not describe no infinite behavior. So this extension, which relies heavily

on the possibility to represent regular trees with maximal sharing, is the main contribution of this section.

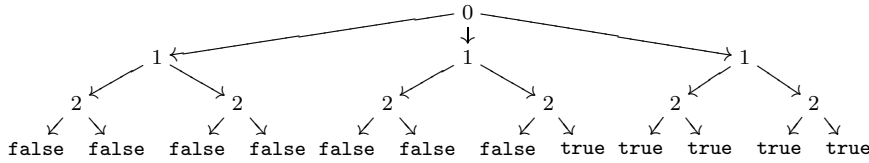
### 4.2.1 MDD for Finite Relations

Let  $R$  be a finite relation of  $\mathcal{R}el$ .  $R$  is a relation on  $\bigotimes_{i < k} [n_i]$ . The *decision tree* for  $R$ ,  $dt$ , is defined by:

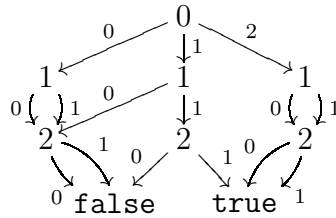
$$pos(dt) \stackrel{\text{def}}{=} \left\{ p \mid \exists e \in \bigotimes_{i < k} [n_i], p \preceq word(e) \right\}$$

$$\begin{aligned} \forall p \in pos(dt), dt(p) &\stackrel{\text{def}}{=} |p| \text{ if } |p| < k \\ &\stackrel{\text{def}}{=} \text{true if } \langle p \rangle \in R \\ &\stackrel{\text{def}}{=} \text{false if } \langle p \rangle \notin R \end{aligned}$$

**Example:** Let  $R$  be the relation  $\{ \langle 111 \rangle, \langle 200 \rangle, \langle 201 \rangle, \langle 210 \rangle, \langle 211 \rangle \}$  on  $[3] \times [2] \times [2]$ . If we call  $e_i$  the  $i^{\text{th}}$  entry of the relation, another way of describing it is the logical formula: “ $(e_0 = 1 \text{ and } e_1 = 1 \text{ and } e_2 = 1) \text{ or } e_0 = 2$ ”. The decision tree for this relation is:



or if we represent the tree with maximal sharing,



◇

As for binary relations, this tree corresponds to a decision procedure to determine whether a given element of  $\bigotimes_{i < k} [n_i]$  is in  $R$ . This decision procedure is slightly different from the standard one, due to the possible use of common names for equivalent entries. To illustrate this, we will describe the elements of  $\bigotimes_{i < k} [n_i]$  as functions from names to sets of integers:  $e(x) \stackrel{\text{def}}{=} \{ e_{(i)} \mid \text{name}_R(i) = x \}$ . We define a function to read and delete an element of  $e$ ,  $\text{read}(e, x) = (e', m)$ . This

function is defined on elements such that  $e(x) \neq \emptyset$ . Its result has the following property:  $m \in e(x)$  and  $e' = e_{(\mathbf{c}\{j\})}$ , where  $j$  is such that  $\text{name}_R(j) = x$  and  $e_{(j)} = m$ . Then the decision procedure is:

$$\begin{aligned} \text{decision}(e, \text{true}) &\stackrel{\text{def}}{=} \text{true} \\ \text{decision}(e, \text{false}) &\stackrel{\text{def}}{=} \text{false} \\ \text{decision}\left(e, \begin{array}{c} k \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}\right) &\stackrel{\text{def}}{=} \text{decision}(e', t_i) \text{ where } (e', i) = \text{read}(e, \text{name}_R(k)) \end{aligned}$$

**Example:** We keep our example of the relation “( $e_0 = 1$  and  $e_1 = 1$  and  $e_2 = 1$ ) or  $e_0 = 2$ ”. In this relation, the entries 1 and 2 are equivalent. So, we can choose  $\text{name}_R(0) = x$ , and  $\text{name}_R(1) = \text{name}_R(2) = y$ . Suppose we want to decide whether the vector  $\langle 010 \rangle$  is in the relation. This vector can be described by the function  $e : x \rightarrow \{0\}, y \rightarrow \{1, 0\}$ . Note that the vector  $\langle 001 \rangle$  would be described by the same function. The decision process starts with

$$\text{decision}\left(e, \begin{array}{c} 0 \\ \swarrow \downarrow \searrow \\ 1 \quad 1 \quad 1 \\ \swarrow \downarrow \searrow \\ 2 \quad 2 \quad 2 \\ \swarrow \downarrow \searrow \\ \text{false} \quad \text{true} \end{array}\right).$$

As  $\text{name}_R(0)$  is  $x$ , we first call  $\text{read}(e, x)$ , which

can only yield  $(y \rightarrow \{1, 0\}, 0)$ . So, the next step is  $\text{decision}\left(y \rightarrow \{1, 0\}, \begin{array}{c} 1 \\ \swarrow \downarrow \\ 2 \\ \swarrow \downarrow \\ \text{false} \end{array}\right)$ .

This time,  $\text{read}$  can give either  $(y \rightarrow \{1\}, 0)$  or  $(y \rightarrow \{0\}, 1)$ . In the first case, we call  $\text{decision}\left(y \rightarrow \{1\}, \begin{array}{c} 2 \\ \swarrow \downarrow \\ \text{false} \end{array}\right)$ , then  $\text{decision}(\emptyset, \text{false})$ , so the result of the decision process is **false**. Of course, the second case would have led to the same result.  $\diamond$

The decision tree can be simplified by elimination of nodes of the form  $\begin{array}{c} x \\ \swarrow \downarrow \\ t \dots t \end{array}$  which are called *redundant*, because whatever the set  $e(x)$ , the result of the decision procedure on this node will be the same. It is true even if  $x$  is the name of equivalent entries because if one way of reading its values holds a result, any way holds the same result. In fact, there is an interesting property about such nodes:

**Property 4.6** *Whatever the redundant node labeled by  $x$ , in the subtree of this node, any node labeled by  $x$  is redundant.*

**Proof:** Let  $p$  be a path such that  $\text{dt}(p) = x$  and  $\forall i < n_{|p|}, \text{dt}_{[pi]} = \text{dt}_{[p0]}$ , and  $q$  such that  $\text{dt}(p0q) = x$ . Then as the entries  $|p|$  and  $|p0q|$  are equivalent,



$\forall i, j \in [n_{|p|}], dt_{[piqj]} = dt_{[pjqi]}$ . So  $dt_{[p0qi]} = dt_{[piq0]} = dt_{[p0q0]}$ . □

This property justifies the use of names of entries instead of entries as labels of the decision tree. We use it to simplify the presentation of relations.

The tree obtained from the decision tree after the elimination of every redundant nodes and represented in the way defined in chapter 2 is the *multiple decision diagram*. In a direct way, the MDD for the finite relation  $R$  is defined by the following algorithm:

```

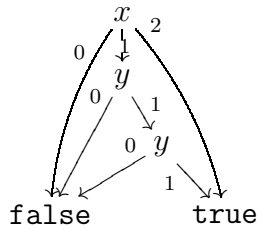
mdd(R):
  return(mddbbuild(R, ε))

mddbbuild(R, e):
  if |e| = k then
    if <e> ∈ R then return(true)
    else return(false)
  else
    for i ← 0 to n|e| - 1 do
      ti ← mddbbuild(R, ei)
    if all the ti are equals then return(t0)
    else x ← nameR(|e|)
       return(
          $\begin{matrix} & x & \\ \swarrow & & \searrow \\ t_0 & \dots & t_{n_{|e|}-1} \end{matrix}$ 

```

In the case of binary relations such that all entry names are different, this gives the BDD of the relation.

**Example:** For the relation which was already illustrated above, described by: “ $(e_0 = 1 \text{ and } e_1 = 1 \text{ and } e_2 = 1) \text{ or } e_0 = 2$ ”, with entry name  $x$  for  $e_0$  and  $y$  for the equivalent entries  $e_1$  and  $e_2$ , the MDD is:



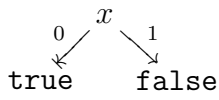
◇

### 4.2.2 Regular Infinite Relations

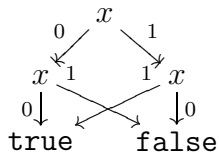
If we start from the decision tree to define the MDD of an infinite relation, then we will have the same decision tree for every relation, up to a renaming of the names of the entries. This decision tree would not in any way represent the relation. This is because trees are defined as functions from *finite* paths to labels, so we cannot give directly the value of the relation if it depends on an infinite vector.

The first step towards a finite representation is the use of entry names to share some entries in the graph representing the decision procedure. The second step is to be able to represent the infinite behaviors of the relations.

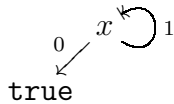
**Examples:** We start with a few examples to illustrate the rest of the section. The exact meaning of the diagrams and how they are built will be specified later. Intuitively, they are read like a decision diagram for a finite relation, except we can loop and some arrows are of the form  $\begin{smallmatrix} x \\ \downarrow \\ t \end{smallmatrix}$ , which means that we can start the description of an infinite behavior.



represents the relation containing every vector starting with a 0. This relation is open. The finite behavior is that we must start with a 0. The infinite behavior is described by **true** and means that we can end with anything.



represents the relation containing the two vectors  $0^\omega$  and  $1^\omega$ . The finite behavior is that starting by a 0 or a 1, we get a different behavior. We have two possible infinite behaviors,  $0^\omega$ , represented by  $\begin{smallmatrix} 0 \\ \swarrow \\ \text{true} \end{smallmatrix} \begin{smallmatrix} x \\ \downarrow \\ \text{false} \end{smallmatrix} \begin{smallmatrix} 1 \\ \swarrow \\ \text{false} \end{smallmatrix}$ , and  $1^\omega$ . The idea when deciding whether a vector is in the relation is that we must start again the latest infinite behavior when we reach a **true**.



represents the relation  $R$  containing any vector ending with  $0^\omega$ . The finite behavior of the relation does not describe the relation, because whatever the finite vectors  $u$  and  $v$ ,  $R(u) \simeq R(v)$ . The infinite behavior is  $0^\omega$ , and can be started after any 1. Intuitively, the decision process is a success if we reach a **true** infinitely often, but we must reset our count each time we start a new infinite behavior.

Note that in the last two examples, all entries are equivalent (i.e. if a vector is in the relations, whatever the permutations of the entries, the permuted vector is in the relation too).  $\diamond$

### Open Relations

In the case of open relations, the only infinite behaviors are square relations. So, if we start the decision tree with such parts already shared and the empty sets shared too, the decision tree will represent the relation.

Let  $R \in \mathcal{R}el$  be an open relation.  $R$  is a relation on  $\bigotimes_{i \in \mathbb{N}} [n_i]$ .

$$\begin{aligned} dt_{op}(R)(p) &\stackrel{\text{def}}{=} \text{true if } \square(R, [|p|], \langle p \rangle) \text{ and } nameiter(R, |p|) \\ &\stackrel{\text{def}}{=} \text{false if } R(\langle p \rangle) = \emptyset \\ &\stackrel{\text{def}}{=} name_R(|p|) \text{ otherwise} \end{aligned}$$

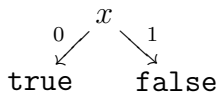
With the convention that  $\square(S, J, w)$  is  $S_{J=w} = \bigotimes_{i \in I \setminus J} E_i$  and  $nameiter(S, k)$  is  $\forall j \in \mathbb{N}, name_S(j+k) = name_S(j)$ .

For a given naming of the entries, this leads to a canonical representation of an open relation, assuming there is a canonical representation of the tree (chapter 3).

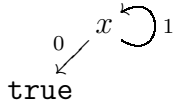
### Iterative Relations

An iterative relation is represented by an open relation. Given the decision tree of this open relation, the decision procedure uses a stacking mechanism: an infinite vector is considered as an infinite word and the decision tree is used as a determinist automaton. Each time the process of reading the vector reaches a **true**, we stack a **true**, and we read the rest of the vector, starting from the beginning of the decision tree. The process stops if we reach a **false**. The process is successful if it does not stop and we stack an infinite number of **true**.

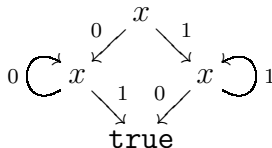
**Examples:** Remember that iterative relations are used to represent the infinite behavior of the relations. For these examples to be more meaningful, we describe the type of infinite property they represent from the point of view of temporal logic [Lam77, MP88].



represents the relation containing just the vector  $0^\omega$ . It is an example of infinite behavior representing a **safety** property.

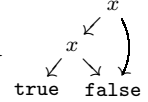


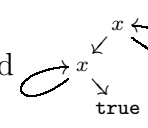
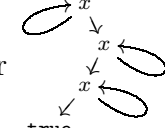
represents the relation that accepts any vector not ending by  $1^\omega$ , that is, with an infinite number of 0's. It is an example of infinite behavior representing a **liveness** property.



represents the relation with any vector with an infinite number of 0's and 1's. It is an example of infinite behavior representing a **fairness** property.

The problem with representing an iterative relation by an open relation, is that there are many open relations that can represent a given iterative relation.

In the first example ( $\{0^\omega\}$ ), we could have used . This looks like a special instance of cycle growth, but we can have more complicated cases, as for the last example (every vector with an infinite number of 0's and 1's), where

we could have used  or . It is an important result of this

chapter that there is indeed a canonical open relation for every iterative relation:

**Property 4.7** *Let  $R$  be an iterative relation. The relation  $S = \{e \mid \exists u \prec \text{word}(e), \langle u^\omega \rangle \in R \text{ and } R(\langle u \rangle) \simeq R\}$  is the greatest (for set inclusion) open relation such that  $R = \Omega(S)$ .*

**Proof:** In order to simplify the proof, we will consider that all entries of  $R$  are equivalent. It is easy to reduce the problem to this case by considering the relation on the product of every different entry name on the period of the relation. We will also ignore the distinction between vector and words, forgetting the casts for a better clarity.

$S$  is an open relation, because whatever the element  $e$  of  $S$ , there is a  $u \prec e$  such that  $\forall f, u.f \in S$ .  $R$  is iterative, so there is an open relation  $S'$  such that  $R = \Omega(S')$ .  $\forall e \in S'$ , there is a  $k$  such that  $\Box(S', [k], e_{([k])})$ . Let  $k_0$  be the least such  $k$ . Because  $R = \Omega(S')$ ,  $e_{([k_0])}^\omega$  is in  $R$ , and  $R(e) \simeq R$ . So  $e \in S$ , which means that  $S' \subset S$ . To prove the property, we just have to prove that  $R = \Omega(S)$ . We will start by  $R \subset \Omega(S)$ , then prove  $\Omega(S) \subset R$ .

**Lemma:** *Let  $v$  be a vector such that  $\Box(S, [k], v)$  and for all  $j < k$ , not  $\Box(S, [j], v_{([j])})$ . Then  $v^\omega \in R$  and  $R(v) \simeq R$ .*

Whatever  $e, v.e \in S$ . Because of the minimality of  $v$  with respect to the “square” property, for each  $e$ , there is a  $l$  such that  $(v.e_{([l])})^\omega \in R$  and  $R(v.e_{([l])}) = R$ . Let  $e = v_{(0)}^\omega$ . There is an  $l$  such that  $(v.v_{(0)}^l)^\omega \in R$ . Because all entries of  $R$  are equivalent, by property 4.1,  $v^\omega \in R$ . Let  $e \in R$ . Because  $R$  is an infinite relation,  $e$  is infinite, so there is an  $i \in \mathbb{N}$  that is repeated infinitely often in  $e$ . But there is an  $l$  such that  $R(v.i^l) \simeq R$ . So  $v.i^l.e \in R$ , and by the equivalence of all entries (property 4.2),  $v.e \in R$ , which means that  $e \in R(v)$ . Conversely, if  $e \in R(v)$ ,  $v.i^l.e \in R$ , and so  $e \in R$ . Thus  $R \simeq R(v)$ .  $\circ$

Let  $e \in R$ . We suppose  $e \notin \Omega(S)$ . If there is a  $k$  such that  $\Box(S, [k], e_{([k])})$ , let  $k_0$  be the least such  $k$ . Then  $e_{(\mathbb{C}[k_0])} \notin \Omega(S)$ , but  $R(e_{([k_0])}) \simeq R$ , so  $e_{(\mathbb{C}[k_0])} \in R$ . So we can iterate on  $e_{(\mathbb{C}[k_0])}$ . This iteration is finite because  $e \notin \Omega(S)$ , so we come

to a point where there is no  $k$  such that  $\square(S, [k], e_{([k])})$ . But  $e \in R$ , so there is a  $k_0$  such that  $\square(S', [k_0], e_{([k_0])})$ , and  $e_{([k_0])}^\omega \in R$  and  $R(e_{([k_0])}) \simeq R$ , and so  $\square(S, [k_0], e_{([k_0])})$ , which contradicts the hypothesis. Thus  $R \subset \Omega(S)$ .

Let  $e \in \Omega(S)$ . Let  $(k_i)_{i \in \mathbb{N}}$  be the sequence:  $k_0 = \min \{k \mid \square(S, [k], e_{([k])})\}$  and  $k_{i+1} = \min \{k \mid \square(S, [k - k_i], e_{([k_i, k]})})\}$ . We call  $u_0 = e_{([k_0])}$ , and  $u_{i+1} = e_{([k_i, k_{i+1}])}$ . Some  $u_i$ 's appear infinitely often in  $e$ , and some  $u_i$ 's appear only finitely often in  $e$ . So there is a permutation of the entries such that the result of the permutation on  $e$  is  $e_f \cdot e_\omega$ , where  $e_f$  is the concatenation of all  $u_i$ 's that appear finitely in  $e$  (times the number of times they appear), and  $e_\omega$  is composed of those  $u_i$ 's that appear infinitely in  $e$  only. By definition of  $\Omega(S)$ ,  $e_\omega \in \Omega(S)$ , and because all entries of  $R$  are equivalent,  $e_f \cdot e_\omega$  is in  $R$  if and only if  $e$  is in  $R$ . But whatever  $u_i$ ,  $R(u_i) \simeq R$  (see the lemma). So  $R(e_f) \simeq R$ . And so  $e$  is in  $R$  if and only if  $e_\omega$  is in  $R$ . Either  $e_\omega$  contains a finite number of distinct  $u_i$ 's, or an infinite one.

If  $e_\omega$  contains a finite number of distinct  $u_i$ 's, we call them  $(v_i)_{i \in [m+1]}$ . Then there is a permutation of the indexes such that the result of the permutation on  $e_\omega$  is  $(v_0 \cdot v_1 \dots v_m)^\omega$ . We know that  $v_m^\omega \in R$  (see the lemma), and for all  $i$ ,  $R(v_i) \simeq R$ , so  $v_0 \cdot v_1 \dots v_{m-1} \cdot (v_m)^\omega \in R$ . We call  $f = v_0 \cdot v_1 \dots v_{m-1} \cdot (v_m)^\omega$ . Because  $f \in R$ , there is a sequence  $(k'_i)_{i \in \mathbb{N}}$  such that:  $k'_0 = \min \{k \mid \square(S, [k], f_{([k])})\}$  and  $k'_{i+1} = \min \{k \mid \square(S, [k - k'_i], f_{([k'_i, k]})})\}$ . So there is a  $j$  such that  $f_{([k'_j])} = v_0 \cdot v_1 \dots v_{m-1} \cdot v_m^n \cdot w$  with  $w \prec v_m$ . Whatever  $i$ ,  $f_{([k'_i])}^\omega \in R$ , because  $R = \Omega(S')$ . So, by property 4.1,  $(v_0 \cdot v_1 \dots v_m)^\omega \in R$ . This, in turn, means that  $e_\omega \in R$ .

If  $e_\omega$  contains infinitely many distinct  $u_i$ 's, we call them  $(v_i)_{i \in \mathbb{N}}$ . Because the relation is over a finite set  $[n_0]$ ,  $e$  is composed of finitely many letters,  $(a_i)_{i \leq m}$  (and  $m < n_0$ ). Each  $a_i$  appears in one  $v_j$  at least, so it appears infinitely often in  $e$ . Because this set of letters is finite, there is a finite sequence  $(v_i)_{i \in J}$  such that for each letter, there is an  $i \in J$  such that  $v_i$  contains the letter. So there is a permutation of the entries that transforms  $e_\omega$  into  $(\bigodot_{i \in J}^\prec v_i)^\omega$ . So we are back to the problem with  $e_\omega$  containing a finite number of  $u_i$ 's.

Thus,  $e_\omega \in R$  whatever the case, which proves that  $e \in R$ . We started from  $e \in \Omega(S)$ , so  $\Omega(S) \subset R$ . Because we already proved  $R \subset \Omega(S)$ , we have  $R = \Omega(S)$ .  $\square$

With this property, we now have a canonical representation for an iterative relation, that is, the canonical representation (defined by  $\mathbf{dt}_{op}$ ) of the greatest open relation representing the iterative relation.

### Sets of Iterative Relations

Let  $R$  be the union of a finite set of iterative relations. Let these relations be indexed:  $R = \bigcup_{i \in C} R_i$ . Each  $R_i$  is represented by an open relation  $S_i$ . Then, it is possible to build a decision tree representing  $R$  based on the decision trees of the open relations. Here is an algorithm building such a tree:

$\text{dt}_{\text{set}}(R)$ :

**return**( $\text{dtset}((\text{dt}_{\text{op}}(S_i))_{i \in C}, 0)$ )

$\text{dtset}((t_i)_{i \in C}, p)$ :

node  $\leftarrow$  **false**

$K \leftarrow \emptyset$

$\forall i \in C$  **do**

**if**  $t_i = \text{false}$  **then**

**for**  $j \leftarrow 0$  **to**  $n_p - 1$  **do**  $t_{ij} \leftarrow \text{false}$

**else if**  $t_i = \text{true}$  **then**

**if** node = **false** **then** node  $\leftarrow$  **true**

$K \leftarrow K \cup \{i\}$

**for**  $j \leftarrow 0$  **to**  $n_p - 1$  **do**  $t_{ij} \leftarrow s_{i[j]}$

**else**  $t_i \leftarrow \begin{matrix} x \\ \swarrow \quad \searrow \\ u_0 \dots u_{n_p-1} \end{matrix}$

    node  $\leftarrow x$

**for**  $j \leftarrow 0$  **to**  $n_p - 1$  **do**  $t_{ij} \leftarrow u_j$

**if** node = **false** **then return**(**false**)

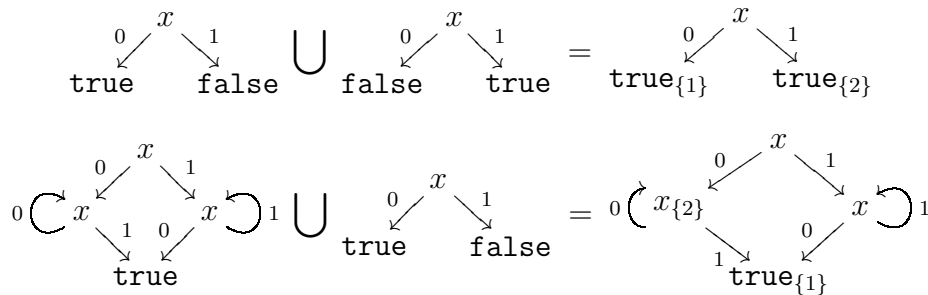
**if** node = **true** **then return**(**true** $_K$ )

**for**  $j \leftarrow 0$  **to**  $n_p - 1$  **do**  $v_j \leftarrow \text{dtset}((t_{ij})_{i \in C}, p + 1)$

**return** $\left( \begin{matrix} x_K \\ \swarrow \quad \searrow \\ v_0 \dots v_{n_p-1} \end{matrix} \right)$

The decision procedure starts with an empty stack and a set of valid indexes of iterative relations  $V = C$ . Each time we encounter a node  $x_K$ , we stack the set  $V \cap K$ . If we encounter a node **false**, we stop. If we encounter a node **true** $_K$ , we stack  $V \cap K$  and we keep going with the new set of valid indexes,  $V \cap K$ . The process is successful if it doesn't stop and if there is an element of  $C$  such that there is an infinite number of sets in the stack containing it. We can notice that **false** is the same as **true** $_{\emptyset}$ , and that we can stop as soon as the set of valid indexes is empty.

**Examples:**



◇

### Regular Relations

Let  $R$  be an  $\omega$ -regular relation. Then  $R$  is associated with a finite set of iterative relations,  $iter(R) = (R_i)_{i \in C}$ . We define a function to describe the infinite behavior of the relation:

$$\forall p \in \mathbb{N}^*, \text{infbehave}_R(p) \stackrel{\text{def}}{=} \{i \in C \mid \forall f \in R_i, \langle p.\text{word}(f) \rangle \in R\}$$

The decision tree for a regular relation is the same as for a finite relation, except that when the prefix  $p$  of a vector is such that  $\text{infbehave}_R(p)$  is not empty, we use the representation of the set of iterative relations associated with  $p$ . In order to mark the switch between the standard interpretation of the decision tree, and the interpretation as a set of iterative relations, we use a new node, which is not a decision node, **iter**. These nodes replace the **true** nodes in order to take into account the fact that even when we start a process of infinite behavior, there can still be finite decisions that lead to another infinite behavior.

$\text{dt}_{reg}(R)$ :

**return**( $\text{dtfinite}(\varepsilon)$ )

$\text{dtfinite}(p)$ :

**if**  $\text{infbehave}_R(p) = \emptyset$  **then**

$x \leftarrow \text{name}_R(|p|)$

**for**  $i \leftarrow 0$  **to**  $n_{|p|} - 1$  **do**  $t_i \leftarrow \text{dtfinite}(pi)$

**return**  $\left( \begin{array}{c} x \\ \swarrow \quad \searrow \\ t_0 \dots t_{n_{|p|}-1} \end{array} \right)$

**else**  $t \leftarrow \text{dtinf} \left( \text{dt}_{set} \left( \bigcup_{j \in \text{infbehave}_R(p)} R_j \right), p \right)$

**return**  $\left( \begin{array}{c} \text{iter}_{\emptyset} \\ \downarrow \\ t \end{array} \right)$

$\text{dtinf}(t, p)$ :

**if**  $t = \text{false}$  **then** **return**( $\text{dtfinite}(p)$ )

**else if**  $t = \text{true}_K$  **then**

$u \leftarrow \text{dtinf} \left( \text{dt}_{set} \left( \bigcup_{j \in \text{infbehave}_R(p)} R_j \right), p \right)$

**return**  $\left( \begin{array}{c} \text{iter}_K \\ \downarrow \\ u \end{array} \right)$

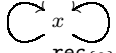
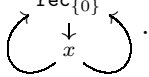
**else**  $t$  is of the form  $\begin{array}{c} X \\ \swarrow \quad \searrow \\ t_0 \dots t_{n-1} \end{array}$ , with  $n > 0$

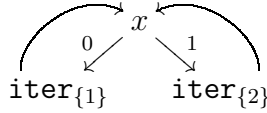
**for**  $i \leftarrow 0$  **to**  $n - 1$  **do**  $u_i \leftarrow \text{dtinf}(t_i, pi)$

**return**  $\left( \begin{array}{c} X \\ \swarrow \quad \searrow \\ u_0 \dots u_{n-1} \end{array} \right)$

The decision process is still a stacking mechanism. While reading an infinite vector, we stack  $K$  each time we encounter a  $x_K$ , and  $\text{iter}_K$  each time we go

through these nodes. The process is successful if, after some point in the stack, we can treat each  $\text{iter}_K$  as a  $\text{true}_K$  as in sets of iterative relations and this new stack is a successful one for a set of iterative relations.

**Examples:** The empty relation (**false**) will be represented by an infinite decision tree of the form  and the square relation (**true**) by an infinite decision tree of the form .



is the relation with any vector ending by  $1^\omega$  or  $0^\omega$ .

◇

### 4.2.3 $\omega$ -deterministic Relations

The representation of regular relations as decision trees is not very efficient, in the sense that there are many sources of non-uniqueness, and therefore many algorithms on these decision trees are not efficient. The worst problem is the choice of iterative relations and of the indexes of these relations. In particular, the relations must be reindexed for any basic boolean operation. These problems come from the non deterministic nature of  $\omega$ -regular languages, and in fact the decision process itself is non deterministic.

So we introduce a new class of relations,  $\omega$ -deterministic relations. The idea is to restrict the possible infinite behavior of the relation for a given prefix to at most one iterative relation. It is obvious then that this class of relations is not closed under union, but it seems to be dense enough to be used as an approximation of regular relations. This approach is easy to exploit in the framework of abstract interpretation.

We first start by a definition of the iterative behavior of a relation at a vector  $u$ : a relation  $R$  defines an iterative relation for each vector  $u$ . This relation is denoted  $R_{[u]}^\Omega$ . It is the iterative relation represented by the open relation:

$$\{e \mid \exists v \prec \text{word}(e), (\text{word}(u) \cdot v^\omega) \in R \text{ and } R(\prec \text{word}(u) \cdot v \succ) \simeq R(u)\}$$

**Definition:** A relation  $R$  is said to be  $\omega$ -deterministic if and only if  $R$  is prefix regular and for all  $u$ ,  $R_{[u]}^\Omega \subset R(u)$ . The following property justifies the denomination “deterministic”, because it shows that there is always a canonical iterative relation representing the infinite behavior at a given point:



**Property 4.8** *An  $\omega$ -deterministic relation  $R$  is  $\omega$ -regular, and whatever the relations in  $\text{iter}(R)$  representing the infinite behavior at a given point  $u$ , there is an iterative relation representing the infinite behavior at  $u$  and containing all of them.*

**Proof:** Let  $R$  be an  $\omega$ -deterministic relation. We first prove that  $R$  is  $\omega$ -regular. We define  $\text{iter}(R) = \left\{ R_{[u]}^\Omega \mid R_{[u]}^\Omega \neq \emptyset \right\}$ . This set is finite because  $R$  is prefix regular. Let  $e \in R$ . If  $e$  is ultimately periodic, as  $R$  is prefix regular, there is a  $u$  and a  $v$  such that  $R(\langle u.v \rangle) = R(\langle u \rangle)$  and  $e = \langle u.v^\omega \rangle$ . Then  $R_{[u]}^\Omega \neq \emptyset$ , and for all  $f \in R_{[u]}^\Omega$ ,  $\langle u.\text{word}(f) \rangle$  is in  $R$  because  $R$  is  $\omega$ -deterministic. If  $e$  is not ultimately periodic, as the entry names are ultimately periodic, there is a permutation of the entries which transforms  $e$  in  $f$  which is ultimately periodic. Moreover, it is possible to choose the permutation that it leaves  $u$  unchanged (we start after the last letter that appears finitely many times and after the looping of the relation) such that  $f = \langle u.v^\omega \rangle$  as above. As the  $R_{[u]}^\Omega$  have the same equivalence of entries as  $f(u)$  (at least), we have the postfix of  $e$  after  $u$  is in  $R_{[u]}^\Omega$ .

Concerning the canonicity of the iterative relation at a given point, we have the fact that for all iterative relation at  $u$  of  $R$ , the iterative relation is included in  $R_{[u]}^\Omega$ .  $\square$

## Decision Tree

Let  $R$  be an  $\omega$ -deterministic relation. Each  $R_{[u]}^\Omega$  is represented by the decision tree of the greatest possible open relation,  $\text{dt}_u$ . The idea is the same as for a regular relation, but there is no need for indexes, and each time we come to a **true**, we don't have to take into account the possibility of another infinite behavior. Thus, the only **iter** nodes will be **iter** $_\emptyset$ . We can see the use of these nodes as a tagging

of some edges. Thus, as a shorthand, we will sometime write  $\frac{x}{t}$  for **iter** $_\emptyset$ .

$\text{dt}_\omega(R)$ :

**return**( $\text{dtfinite}_\omega(\varepsilon)$ )

$\text{dtfinite}_\omega(p)$ :

**if**  $R(\langle p \rangle) = \emptyset$  **then return**(**false**)

**else if**  $R_{[p]}^\Omega = \emptyset$  **then**

$x \leftarrow \text{name}_R(|p|)$

**for**  $i \leftarrow 0$  **to**  $n_{|p|} - 1$  **do**  $t_i \leftarrow \text{dtfinite}_\omega(pi)$

**return**  $\left( \frac{x}{t_0 \dots t_{n_{|p|}-1}} \right)$

```

else
   $t \leftarrow dtinf_{\omega}(dt_p, p)$ 
  return  $\left( \begin{array}{c} \text{iter} \\ \downarrow \\ t \end{array} \right)$ 

```

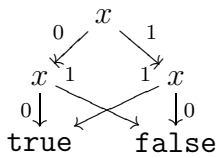
```

 $dtinf_{\omega}(t, p)$ :
  if  $t = \text{false}$  then return( $dtfinite_{\omega}(p)$ )
  else if  $t = \text{true}$  then return(true)
  else  $t$  is of the form  $\begin{array}{c} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}$ , with  $n > 0$ 
    for  $i \leftarrow 0$  to  $n - 1$  do  $u_i \leftarrow dtinf_{\omega}(t_i, pi)$ 
    return  $\left( \begin{array}{c} x \\ \swarrow \searrow \\ u_0 \dots u_{n-1} \end{array} \right)$ 

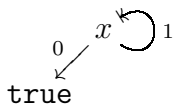
```

The decision process reads a vector and uses a stack  $S$  and a current iterative tree,  $r$ . At the beginning, the stack is empty and  $r = dt_{\omega}(R)$ . When we come to a **true** node, we stack it and start again at  $r$ . When we come to a  $\begin{array}{c} \text{iter} \\ \downarrow \\ t \end{array}$  node,  $r$  becomes  $t$  and we empty the stack. If we come to a **false** node, we stop the process. The process is a success if it doesn't stop and the stack is infinite. If  $t$  is such a decision diagram, we denote by  $R_{\omega}(t)$  the set of vectors for which this process is a success.

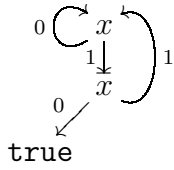
**Examples:**



when we read a 0, the current iterative tree becomes  $\begin{array}{c} x \\ \swarrow \searrow \\ \text{true} \quad \text{false} \end{array}$ . If we read a 1 after that, we stop on a failure, and if we read a 0, we stack a **true** and start again with the same iterative tree. So, after a 0, we can only have  $0^{\omega}$ . After a 1, the iterative tree becomes  $\begin{array}{c} x \\ \swarrow \searrow \\ \text{false} \quad \text{true} \end{array}$ , and this time, we can only have  $1^{\omega}$ . So this relation is  $\{0^{\omega}, 1^{\omega}\}$ .



during the decision process, the iterative tree never changes. When we read a 0, we stack a **true** and start again. But each time we read a 1, we empty the stack. So the only vectors that stack an infinite number of **true** are the vectors ending by  $0^{\omega}$ .



after the first 1 is read, the current iterative tree does not change. If we have read an odd number of 1's, when we read a 0, we stack a **true** and start the process again. So any vector with an odd number of 1's is in the relation. If we read a 1, we cannot stack another **true** before emptying the stack. So the relation contains no vector with an infinite number of 1's. Finally, if the number of 1's is even, we cannot stack any more **true** if there is no more 1. So, the relation is exactly the set of vectors with a finite odd number of 1's (and so ending with  $0^\omega$ ).

◇

Open relations are  $\omega$ -deterministic. Their representation as an  $\omega$ -deterministic relation is the same tree as  $dt_{op}$ , with a **iter** node before the **true** nodes.

### Redundant Nodes

A node is *redundant* if it is either of the form  $\begin{matrix} x \\ \swarrow \searrow \\ t \dots t \end{matrix}$  or of the form  $\begin{matrix} x \\ \swarrow \searrow \\ t \dots t \end{matrix}$ . In the first case, we can replace it by the  $t$  node, and in the second case by the  $\begin{matrix} \text{iter} \\ \downarrow \\ t \end{matrix}$  node. Having two consecutive **iter** nodes is impossible, as we forbid the empty relation in the set of iterative relations. Note that this elimination of redundant nodes may lead to a recursive process if the current node is also a subtree of  $t$  (it is in a loop). In this case, we can directly have any arrow that pointed to the node point to  $t$ .

After elimination of redundant nodes, we obtain the MDD of the relation. If we consider finite relations as open relations (which is usually the case when using BDD), we obtain the same decision diagram, except for the **iter** node before **true** nodes.

**Property 4.9** *The representation of an  $\omega$ -deterministic relation by means of the tree  $dt_\omega$ , where every redundant node have been eliminated, and represented with maximal sharing is unique.*

This property is a consequence of the canonicity of the iterative relations and the canonicity of the open relations representing these iterative relations.

### 4.2.4 MDD with Large Sets of Choice

The power of MDD comes from the sharing and the elimination of redundant nodes. But the larger the choice space for a given entry, the less likely a node for this entry is to be redundant.

A natural solution<sup>3</sup> to this problem is the representation of a given entry with choice space  $[n]$  by  $\lceil \log_2(n) \rceil$  binary variables. A given integer of  $[n]$  is written as a binary word. The only problem with this solution is that the access time is longer, since you need  $\lceil \log_2(n) \rceil$  times more indirections to read the values of a vector.

In our case, this representation is not very well suited, as we will need for each variable an entire new set of fresh variables which needs to be stored too. This results in a new indirection in the use of relations, but experience proves, that this technique provides good results as soon as the set of choice is “large enough”.

### 4.3 Algorithms on $\omega$ -deterministic Relations

These algorithms manipulate  $\omega$ -deterministic relations represented by a decision diagram, as defined in the previous section.

#### 4.3.1 Canonicity of the representation

The representation of the decision tree for an  $\omega$ -deterministic relation relies on shared regular tree (see chapter 3). By property 4.9, the decision tree for a relation is unique, so we have a unique representation of  $\omega$ -deterministic relations. But the problem is that not all regular trees are valid decision trees of a relation, so we must maintain some properties on the decision trees when manipulating relations. Those properties concern the “special” nodes of the decision trees: **true**, **false** and **iter**. For example, if during the computation, we obtain  $\begin{array}{c} \curvearrowright \\ \text{\scriptsize } x \end{array}$  or  $\begin{array}{c} \curvearrowleft \\ \text{\scriptsize } x \end{array}$ , we must recognize **false**.

The idea is that we will describe the algorithms on  $\omega$ -deterministic relations as regular tree transformations. The problem of the uniqueness will be supposed to be taken care of by treating the “special” nodes in the way described below. In this way, we will separate the two algorithmic problems. Moreover, the treatment of special nodes is not too hard to handle as they correspond to graph properties that can easily be verified.

#### The iter Node

From the decision procedure for  $\omega$ -regular relations, it is easy to see when the **iter** node is not useful. It is not useful if the iterative tree it points to is never used to restart the decision process. That is, there is no path from the **iter** node to a **true** node that does not go through another **iter** node. Thus, when we create a new **iter** node, the only information we need is whether the node it points to may lead to a **true** node, which is easily performed by returning

<sup>3</sup>See [SKMB90] for an example of implementation.

this information for each node. In order to exclude paths going through an `iter` node, the `iter` node then returns `false`.

If the `iter` node is inside a cycle, though, deciding whether there is a path from this node to a `true` node is a little more complex. The point is that we must avoid infinite computation by considering some edges as not relevant because they cannot add a correct path to `true` if there is not one revealed by the edges considered as relevant. If the `iter` node is the root of the cycle we consider, the return edges are exactly the non relevant edges. Thus all we have to do is consider the `iter` node as the root of the cycle, that is we forget any node of the cycle that was already visited to compute whether this node is useful.

If the `iter` nodes are correct, then we can use them to help the computation of the other special nodes. For example, we know that there is a path leading from the `iter` node to a `true` node, so the `iter` node is not equivalent to `false`.

### The false Node

Once again, the decision procedure gives a straightforward algorithm to determine whether a newly created node is equivalent to `false`. It is the case if there is no path leading from this node to `true`. This information is very easy to extract from the graph we are building too. If the node is not in a cycle, then we suppose that its children are uniquely represented, thus the node is equivalent to `false` if and only if all its children are `false`. Note that this operation is already automatically performed by the elimination of redundant nodes.

If the node is in a cycle, we just return the pertinent information, that is, whether there is a path leading from this node to `true`. If any node in the cycle is equivalent to `false`, then every node in the cycle is equivalent to `false`. Thus, the information that will decide whether the entire cycle is equivalent to `false` is the information returned by the root of the cycle. In the cycle, return edges return the information that they don't lead to a `true` node, because they don't add no new path that would lead to `true`.

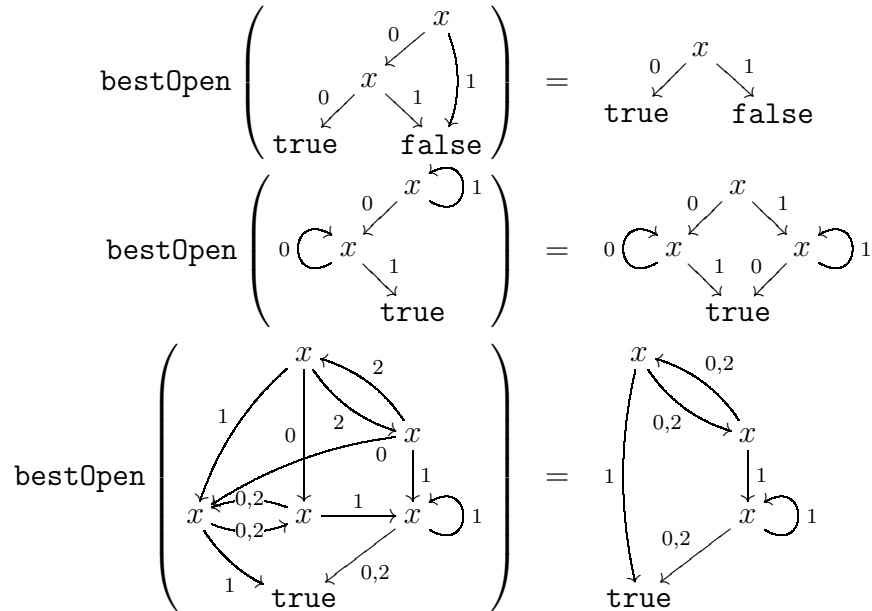
### The true Node

In fact `true` nodes can be seen as tags on some return edges. Thus striving for canonicity of the representation will raise again the problems of cycle growth and root unfolding that are inherent in regular trees. The problems are a little different, though, because the equivalent of cycle growth reduction for tagged cycles is the determination of the greatest open relation. Concerning root unfolding, the problem is a little simpler in the case of these cycles, though, because the root of the tagged cycle is always an `iter` node (although it is not always the root of non tagged cycles, as the representation of the relation containing a finite odd number of 1 shows).

**The Greatest Open Relation** Finding the greatest open relation representing a given iterative relation is very similar to cycle growth reduction, because if an open relation is greater than another one representing the same iterative relation, the length of the cycles is smaller for the greater relation (we arrive sooner to a `true`).

In order to keep things simple, we start by describing the algorithm to build the greatest open relation for an iterative relation. In the general case, this corresponds to an iterative relation which never leads to an `iter` node.

**Examples:** We present a few examples of open relations and the greatest open relation representing the same iterative relation. The first two examples have already been presented when proving that such a greatest open relation exists (page 56). The first one corresponds to the relation  $\{0^\omega\}$  and the second one to the relation with every vectors containing an infinite number of 0's and an infinite number of 1's. The last example is more complicated and shows the interest of some parts of the algorithm, namely the distinction between the different  $R(v)$ . This relation  $R$  is the set of all vectors with an infinite number of 1's, and if the total number of 0's and 1's in the vector is finite, the number of 0's is equal to the number of 2's modulo 2. Here,  $R \simeq R(\langle 1 \rangle) \simeq R(\langle 02 \rangle)$  but is different from  $R(\langle 0 \rangle)$ .



◇

$\text{bestOpen}(R)$ :

$R$  is a relation on  $\bigotimes_{i \in \mathbb{N}} [n_i]$

$L$  is the function defined as  $L(R(e)) = \emptyset$ , for all  $e$  finite vector  
**for**  $i \leftarrow 0$  **to**  $n_0 - 1$  **do**  $t_i \leftarrow \text{bestOpenRec}(\varepsilon, i, \emptyset, L, \emptyset, \text{false})$   
**if** all  $t_i$  are equal **then return**( $t_0$ )  
 $x \leftarrow \text{name}_R(0)$   
**return**  $\left( \begin{array}{c} x \\ \swarrow \searrow \\ t_0 \dots t_{n_0-1} \end{array} \right)$

$\text{bestOpenRec}(v, a, V, L, T, \text{omega})$ :

**if**  $R(\langle va \rangle) = \emptyset$  **then return**(**false**)  
 $x \leftarrow \text{name}_R(|va|)$  and  $n \leftarrow n_{|va|}$   
**if**  $\text{nameiter}(R, |a|)$  **then**  
  **if**  $\text{omega}$  or  $\langle va^\omega \rangle \in R$  **then**  
    **if**  $R(\langle va \rangle) \simeq R$  **then return**(**true**)  
    **else**  $\text{omega}' \leftarrow \text{true}$   
  **else**  $\text{omega}' \leftarrow \text{false}$   
   $V' \leftarrow V \cup \{a\}$   
  **if**  $V' \subset L(R(\langle va \rangle))$  **then return**( $T(R(\langle va \rangle))$ )  
  Let  $X$  be a new label  
   $T' \leftarrow T$  with  $R(\langle va \rangle) \rightarrow X$   
   $L' \leftarrow L$  with  $R(\langle va \rangle) \rightarrow V'$   
  **for**  $i \leftarrow 0$  **to**  $n - 1$  **do**  $t_i \leftarrow \text{bestOpenRec}(va, i, V', L', T', \text{omega}')$   
  **if** all  $t_i$  are equal **then return**( $\text{recCons}(t_0, X)$ )  
  **return**  $\left( \text{recCons} \left( \begin{array}{c} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}, X \right) \right)$   
**else**  
  **for**  $i \leftarrow 0$  **to**  $n - 1$  **do**  $t_i \leftarrow \text{bestOpenRec}(v, ai, V, L, T, \text{omega})$   
  **if** all  $t_i$  are equal **then return**( $t_0$ )  
  **return**  $\left( \begin{array}{c} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array} \right)$

**Proof:** Let  $\text{dt} = \text{bestOpen}(R)$ .  $S$  is the greatest open relation representing  $R$ . In order to simplify the proof, we suppose all entries of  $R$  are equivalent.

Whatever the word  $v$  such that the algorithm actually reaches  $v$  and decides that  $\text{dt}(v) = \text{true}$  or **false**,  $v \in \text{pos}(\text{dt}_{op}(S))$  and  $\text{dt}(v) = \text{dt}_{op}(S)(v)$ . The reason of this fact is: whatever  $u \prec v$ ,  $u$  is actually reached by the algorithm, so ( $u^\omega \in R$  and  $R(u) \simeq u$ ) is false, and  $R(u) = \emptyset$  is false (or else the algorithm would have stopped at  $u$ ), so  $v \in \text{pos}(\text{dt}_{op}(S))$ . If  $\text{dt}(v) = \text{false}$ , then  $R(\langle v \rangle) = \emptyset$ , so  $\text{dt}_{op}(S)(v) = \text{false}$ . If  $\text{dt}(v) = \text{true}$ , then  $R(\langle v \rangle) \simeq R$ , and either  $\langle v^\omega \rangle \in R$  or  $\text{omega}$  is **true**. But if  $\text{omega}$  is **true**, then there is  $u \prec v$  such that  $\langle u^\omega \rangle \in R$ .  $v = uw$ . Because  $R(\langle v \rangle) \simeq R$ ,  $\langle uw(u)^\omega \rangle \in R$ . By equivalence of the entries of  $R$ ,  $\langle w(u)^\omega \rangle \in R$ , so  $\langle (wu)^\omega \rangle \in R$ , and finally  $\langle (uw)^\omega \rangle \in R$ . Thus  $\text{dt}_{op}(S)(v) = \text{true}$ .

All we have to prove now, to show that  $\mathbf{dt} = \mathbf{dt}_{op}(S)^4$ , is that every return edge introduced in the algorithm (by the new labels and `recCons` construction) is valid. We prove that whatever such return edge at word  $v$ , going back to  $u \prec v$ , whatever  $w$ ,  $\mathbf{dt}_{op}(S)(vw) = \mathbf{dt}(vw)$ . Because  $\mathbf{dt}_{[v]} = \mathbf{dt}_{[u]}$ , we just have to prove that  $\mathbf{dt}_{op}(S)(vw) = \mathbf{dt}(uw)$ . We use the property that  $R(\langle u \rangle) \simeq R(\langle v \rangle)$ , and every letter in  $v$  is in  $u$ .

Suppose  $\mathbf{dt}_{op}(S)(vw) = \mathbf{false}$ . Then  $R(\langle vw \rangle) = \emptyset$  and for all  $v' \prec vw$ ,  $R(\langle v' \rangle) \neq \emptyset$ , and  $\langle v'^\omega \rangle \notin R$  or  $R(\langle v' \rangle) \not\subseteq R$ . Because  $R(\langle u \rangle) \simeq R(\langle v \rangle)$ ,  $R(\langle uw \rangle) = \emptyset$ . If there is  $u' \prec uw$  such that  $\mathbf{dt}(u') = \mathbf{false}$ ,  $u'$  cannot be a prefix of  $u$ , else  $u'$  is a prefix of  $v$ , and so of  $vw$ , and we cannot have a prefix of  $vw$  such that the relation is empty on that prefix. So  $u' = uw'$  with  $w' \prec w$ . But  $R(\langle uw' \rangle) \simeq R(\langle vw' \rangle)$  and  $vw' \prec vw$ . Suppose  $\mathbf{dt}(u') = \mathbf{true}$ , we still have  $u' = uw'$ . We prove in the next paragraph that  $\mathbf{dt}_{op}(uw')$  is **true**. So  $\mathbf{dt}(uw) = \mathbf{false}$ . The converse is obvious.

Suppose  $\mathbf{dt}_{op}(S)(vw) = \mathbf{true}$ . Then  $R(\langle vw \rangle) \simeq R$  and  $\langle (vw)^\omega \rangle \in R$  and for all  $v' \prec vw$ ,  $R(v') \neq \emptyset$ , and  $\langle v'^\omega \rangle \notin R$  or  $R(\langle v' \rangle) \not\subseteq R$ . Because  $R(\langle u \rangle) \simeq R(\langle v \rangle)$ ,  $R(\langle uw \rangle) \simeq R$ . Because every letter of  $v$  is in  $u$  and every entry of  $R$  is equivalent, we know from property 4.1 that  $\langle (uw)^\omega \rangle \in R$ . If there is  $u' \prec uw$  such that  $\mathbf{dt}(u') = \mathbf{false}$ , then  $R(\langle uw \rangle)$  would be empty. If  $\mathbf{dt}(u') = \mathbf{true}$ ,  $u' = uw'$ . We would have  $R(\langle vw' \rangle) \simeq R(\langle uw' \rangle) \simeq R$ , and because  $\langle (uw')^\omega \rangle \in R$ ,  $\langle (vw')^\omega \rangle \in R$  (because  $u \prec v$ ), which is forbidden by maximality of  $S$ . So  $\mathbf{dt}(uw) = \mathbf{true}$ . Conversely, we already proved that if  $\mathbf{dt}(uw) = \mathbf{true}$ , then  $R(\langle vw \rangle) \simeq R$  and  $\langle (vw)^\omega \rangle \in R$ . If  $v' \prec vw$  is such that  $\mathbf{dt}_{op}(S)(v') = \mathbf{false}$ , we cannot have  $R(\langle vw \rangle) \simeq R$ . If there is a  $v' \prec vw$  such that  $\mathbf{dt}_{op}(S)(v') = \mathbf{true}$ , we cannot have  $v' \prec v$ , or else the algorithm would not have gone beyond  $v'$ , so  $v' = vw'$ . We already proved that in this case,  $\mathbf{dt}(uw') = \mathbf{true}$ , and  $uw' \prec uw$ . So  $\mathbf{dt}_{op}(S)(vw) = \mathbf{true}$ .  $\square$

Note that the algorithm terminates because, at each recursive call, one element of the function  $L$  increases, and by prefix regularity of  $R$ , the number of distinct  $R(e)$  is finite, and the number of different choices over a period of the entries of  $R$  is finite too.

For this algorithm to be complete, we must also explain how we test  $\langle v^\omega \rangle \in R$  and the equivalence of different  $R(e)$ .  $R$  is represented by the decision tree  $\mathbf{dt}$  of an open relation, which, at this point, is not necessarily the greatest one.

Concerning  $\langle v^\omega \rangle \in R$ , we use the fact that it is a necessary and sufficient condition that there is a  $k > 1$  such that in the decision process,  $v^k$  reaches a **true**. All we have to do is to perform the decision process while storing the pairs  $(u, t)$  where  $u \preceq v$  and  $t \prec \mathbf{dt}$ , and if we come to such a pair twice, we stop in failure, and if we reach a **true** we stop in success.

Concerning the equivalence of  $R(e)$  and  $R(f)$ , we start from the two subtrees

---

<sup>4</sup>up to elimination of redundant nodes, which is not performed in  $\mathbf{dt}_{op}(S)$ .



of  $dt$ ,  $t_e$  and  $t_f$  which are reached respectively from the decision process on  $e$  and  $f$ . We compare every pair of subtrees which we can reach from  $t_e$  and  $t_f$ , and every time we come again on this pair on the decision process, we must have passed through **true** in both subtree or in none. The algorithm is:

```

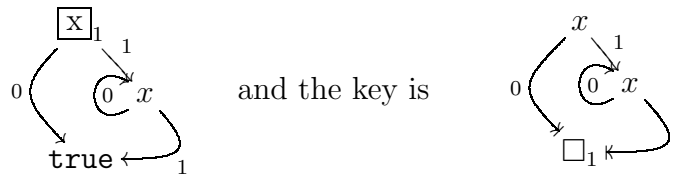
compareRel( $t, u, S, dt$ ):
  if  $t = u$  then return(true)
  if  $(t, u) \in \text{dom}(S)$  then
     $(b_1, b_2) \leftarrow S(t, u)$ 
    return( $b_1 = b_2$ )
   $S' \leftarrow S$  with  $(t, u) \rightarrow (\mathbf{false}, \mathbf{false})$ 
   $t$  is  $\begin{array}{c} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}$ , and  $u$  is  $\begin{array}{c} x \\ \swarrow \searrow \\ u_0 \dots u_{n-1} \end{array}$ 
  for  $i \leftarrow 0$  to  $n - 1$  do
    if  $u_i = \mathbf{false}$  then
      | if  $t_i \neq \mathbf{false}$  then return(false)
      | else if  $t_i = \mathbf{false}$  then return(false)
      | else if  $t_i = \mathbf{true}$  then
        | if  $u_i \neq \mathbf{true}$  then
          |  $S' \leftarrow S'$  with all values  $(b_1, b_2)$  of  $S'$  become (true,  $b_2$ )
          | if compareRel( $dt, u_i, S', dt$ ) = false then return(false)
        | else if  $u_i = \mathbf{true}$  then
          |  $S' \leftarrow S'$  with all values  $(b_1, b_2)$  of  $S'$  become  $(b_1, \mathbf{true})$ 
          | if compareRel( $t_i, dt, S', dt$ ) = false then return(false)
        | else
          | if compareRel( $t_i, u_i, S', dt$ ) = false then return(false)
    return(true)

```

If the iterative relation is inside a loop, or if there is a path from the decision tree of the iterative relation to an **iter** node, we must consider this node as a **false**while performing the construction of the greatest open relation. We must also propagate this information upward, as in the general treatment of the **false** nodes. Because this last node must be restored after the computation, we introduce **false** nodes subscripted by the nodes to which they are equivalent in the general decision diagram. In this way, they are treated as different nodes by the sharing algorithm on trees. If the iterative relation is inside a loop, it is possible that there is a path from a node  $N$  in one of the **false** <sub>$N$</sub>  to the original **iter** node starting the iterative relation. In this case, we must replace in  $N$  this **iter** node by the new one corresponding to the greatest open relation representing the iterative relation.

**Root Unfolding** In classical cycles, we need to build a complex key (a tree) and a simple key (a label plus the children nodes) to deal with root unfolding, for each node in the cycle. In the case of tagged cycles, there is no need for the complex key, as they are already represented by trees. But we must still define the simple keys, and for each simple key the tree representing the tagged cycle rooted at this node. The problem with tagged cycles, is that not every node in the cycle can be the root of the cycle. Only the nodes corresponding to iterative relations can be the root of the cycle.

**Example:** This representation has only one possible key, because, if  $R$  is the relation it represents,  $R(\langle 1 \rangle)$  is not iterative. ( $R$  is the set of vectors with either an infinite number of 1's, or an even number of 1's.)



◇

**Property 4.10** Let  $R$  be an iterative relation represented by the tree  $\mathbf{dt}$ . The relations  $R(\langle u \rangle)$  is iterative if and only if for every path  $v$  that goes from the root of  $\mathbf{dt}$  to  $\mathbf{true}$ , there is a  $w \prec v$  such that  $R(\langle w \rangle) \simeq R(\langle u \rangle)$ . Such a relation is called unavoidable<sup>5</sup> for  $\mathbf{dt}$ .

**Proof:** Let  $R(\langle u \rangle)$  be unavoidable. Then  $R(\langle u \rangle)$  is iterative and represented by the decision tree defined as  $\mathbf{dt}(u)$  where  $\mathbf{true}$  have been  $\mathbf{dt}$  in which every node equivalent to  $R(\langle u \rangle)$  is replaced by  $\mathbf{true}$ . It is obvious that any vector recognized by this “sliding” of the decision tree is in  $R(\langle u \rangle)$ , and reciprocally.

If  $R(\langle u \rangle)$  is not unavoidable, then there is a path  $v$  such that  $\mathbf{dt}(v) = \mathbf{true}$  and for all  $w \prec v$ ,  $R(\langle w \rangle) \not\simeq R(\langle u \rangle)$ . There is a path  $u'$  such that  $\mathbf{dt}(u'.u') = \mathbf{true}$ . Then  $\langle u'.v^\omega \rangle \in R(\langle u \rangle)$ . But in this path, there is no point such that  $R(\langle u'.v^k.w \rangle) \simeq R(\langle u \rangle)$ , so  $R(\langle u \rangle)$  is not iterative. □

Because we already performed the `bestOpen` algorithm, we know already which nodes correspond to equivalent relations. We note  $R_t$  for the relation corresponding to the subtree  $t$  of the decision tree. The following algorithm computes the set of unavoidable relations.

<sup>5</sup>Such sets of nodes would be called dominators of  $\mathbf{true}$  in [AHU74], if we were in an acyclic graph.

```

unavoidableSet(dt):
  return(unavoidableSetRec(dt, {false})

```

```

unavoidableSetRec(t, T):
  if  $t \in T$  then return ( $\{\emptyset\}$ )
  if  $t = \mathbf{true}$  then return ( $\emptyset$ )
   $t$  is  $\begin{matrix} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{matrix}$ 
   $T' \leftarrow T \cup \{t\}$ 
   $i \leftarrow 0$ 
  do
     $E \leftarrow \text{unavoidableSetRec}(t_i, T')$ 
     $i \leftarrow i + 1$ 
  while  $E = \{\emptyset\}$  and  $i < n$ 
  if  $E = \{\emptyset\}$  then return ( $E$ )
  for  $j \leftarrow i$  to  $n - 1$  do
     $E' \leftarrow \text{unavoidableSetRec}(t_j, T')$ 
    if  $E' \neq \{\emptyset\}$  then  $E \leftarrow E \cap E'$ 
  return ( $E \cup R_t$ )

```

When this set of relations is computed, we just add in the key dictionary the trees and unfoldings corresponding to each unavoidable node. The decision tree corresponding to the unavoidable relation  $R$  is called  $\text{dt}_R$ . Remind that  $D$  is the set of keys used for the uniqueness of the trees.

```

addKeys(dt):
   $US \leftarrow \text{unavoidableSet}(\text{dt})$ 
   $S$  is the function  $\mathbf{true} \rightarrow \begin{matrix} \text{iter} \\ \downarrow \\ \text{dt} \end{matrix}$ ,  $\mathbf{false} \rightarrow \mathbf{false}$ 
  return(addKeysRec(dt,  $S$ ))

```

```

addKeysRec(t, S):
  if  $t \in \text{dom}(S)$  then return ( $S(t)$ )
   $t$  is  $\begin{matrix} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{matrix}$ 
  if  $R_t \in US$  then
     $w \leftarrow \begin{matrix} \text{iter} \\ \downarrow \\ \text{dt}_{R_t} \end{matrix}$ 
     $S' \leftarrow S$  with  $t \rightarrow w$  and without all  $s \rightarrow Y$ 
    for  $i \leftarrow 0$  to  $n - 1$  do  $u_i \leftarrow \text{addKeysRec}(t_i, S')$ 
     $k$  is the key  $(x, u_0, \dots, u_n)$ 
    change in  $D$  the value associated with  $k$  to be  $w$ 
  return( $w$ )

```

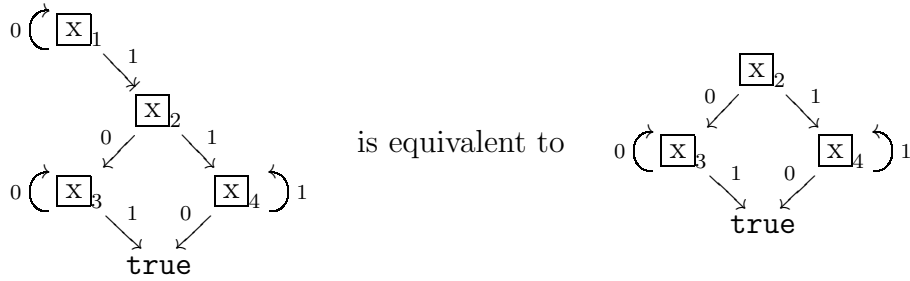
```

else
   $X$  is a new label of arity 0
   $S' \leftarrow S$  with  $t \rightarrow X$ 
  for  $i \leftarrow 0$  to  $n - 1$  do  $u_i \leftarrow \text{addKeysRec}(t_i, S')$ 
  return  $\left( \text{recCons} \left( \begin{array}{c} x \\ \swarrow \quad \searrow \\ u_0 \dots u_{n-1} \end{array}, X \right) \right)$ 

```

We must also deal with root unfolding each time we build a new cycle, either tagged or not. This is performed for generic trees through the use of the `shareWithDone` algorithm. We can still use the same algorithm (it must be used each time we introduce a `iter` node also), but with the modification that each time we go through a `iter` node in the sub-algorithm `tryShare`, we must use a comparison based on `compareRel` instead of `compareWithDone`.

**Example:**



The algorithm will find that the relation at  $\square_1$  is the same as the relation at  $\square_2$ .  $\diamond$

### Recapitulation on Canonicity

All the algorithms that have been describing in this subsection are to be integrated in the `representation` algorithm for any tree representing an  $\omega$ -deterministic relation, so that the representation is indeed unique. Here is a sketch of the new algorithm, including the elimination of redundant nodes:

- In addition to other useful informations, `represent` returns a boolean  $r$  indicating whether there is a path from the current node to a `true` node without going through an `iter` node.
- if the current node is not in a cycle (line 11 of `represent`) then
  - if the node is an `iter` node

- \* if the boolean returned by `represent` on its child is false, then return the child.
- \* look for root unfolding. If there is an equivalent node, return the equivalent node (don't perform the sequel).
- \* perform `bestOpen` on the relation represented by the node, where every `iter` node  $M$  directly reachable from the current node is replaced by `falseM`, and recursively for every node such that all its children is a `falseM`.
- \* on the result of `bestOpen`, restore every `falseM` to  $M$ .
- \* perform `addKeys` on the current node to update  $D$ .
- if every child of the current node is the same, return the first child (redundant nodes elimination and treatment of `false`).
- if we have identified an independent cycle (test of line 24 of `represent` is true)
  - look all nodes in the cycle. If no partial key leads to a non `false` node, then the entire cycle is equivalent to `false`.
  - perform `shareWithDone` with the new comparison in case of `iter` node.
  - for all `iter` nodes in the cycle,
    - \* look for a path starting from this node and going to a `true` node without going through an `iter` node. If there is no one, suppress this `iter` node, and every node in the cycle that pointed to the `iter` node now points to its child
    - \* perform the same operations as when the `iter` node is not in a cycle
    - \* if the node changes, make any node that pointed to the old node in the cycle point to the new one.
- after sharing (line 10 of `representCycle`), eliminate redundant nodes in the cycle.

### 4.3.2 Entry Names

At the beginning of the chapter we described the relations as “coming with entry names”, and indeed, entry names are determined by the applications using the relations. But when we want to perform any operation between two relations, we need to compute valid entry names for the result of the operation. If the operation preserves the action of entry substitution, we can take any entry name that is valid for both relations. So, we describe a kind of name merging algorithm, that will give such a set of entry names.

## Representation

In many algorithms of this chapter, we used the function  $\text{name}_R$  that is supposed to come with  $R$ . To know the values of this function, we cannot rely on the decision diagram, because, due to redundant nodes elimination, every entry is not necessarily represented in the diagram. In the case of classical BDDs, one can see the ordering on the variables as the representation of the function  $\text{name}_R$  for every  $R$  in a given BDD manager.

In the general case, we cannot use an ordering, as different entries may have the same name (which would lead to  $x < y < x$  if the entries 0 and 2 have the same name  $x$ , for example). In order to keep a unique representation of  $\text{name}_R$ , we use a regular tree, which is in fact equivalent to a regular word: each node is labeled by an entry name, of arity 1. We call this tree  $\text{nt}_R$ . It is defined formally as:

$$\begin{aligned} \text{pos}(\text{nt}_R) &\stackrel{\text{def}}{=} \{0\}^* \\ \text{nt}_R(0^n) &\stackrel{\text{def}}{=} \text{name}_R(n) \end{aligned}$$

Thus, the name tree of an infinite relation with all entries equivalent named  $x$  is  $x \circlearrowleft$ .

Once we have the name tree of the relation, it is easy to restore the decision tree with the redundant nodes. So, while performing most algorithm on decision trees, we go through the name tree at the same time, so that we can keep track of the actual entry number associated with a given node. This technique is presented in the first basic algorithm, **instantiate**. In the other ones, we will describe the algorithm as if there were no elimination of the redundant nodes, ignoring the name trees so that the description is a little more simple.

## Entry Names Merging

Let  $R$  and  $S$  be two regular relations. We suppose that we have a way of determining for the entry  $e$  and the entry names  $\text{name}_R(e)$  and  $\text{name}_S(e)$  what entry name **choose**( $e, \text{name}_R(e), \text{name}_S(e)$ ). It can be one of the entry names of  $R$  or  $S$ , or an entirely new one. The idea is that for every couple  $(x, y)$  of entry names of  $R$  and  $S$ , we must have a distinct entry name. In this way, the new set of entry names is compatible with both  $R$  and  $S$ .

**mergeEntries**( $t, s, n, S$ ):

**if**  $(t, s) \in \text{dom}(S)$  **then return**( $S(t, s)$ )

$t$  is of the form  $\begin{array}{c} x \\ \downarrow \\ t' \end{array}$  and  $s$  of the form  $\begin{array}{c} y \\ \downarrow \\ s' \end{array}$

$z \leftarrow \text{choose}(n, x, y)$

add  $(t, s) \rightarrow X_n$  to  $S$  ( $X_n$  is a label of arity 0)

$u \leftarrow \text{mergeEntries}(t', s', n + 1, S)$

```
return  $\left( \text{recCons} \left( \begin{array}{c} z \\ \downarrow \\ u \end{array}, X_n \right) \right)$ 
```

When we use this name tree in algorithms, we refer to a given `choose`( $n, x, y$ ) as  $xy$ .

### 4.3.3 Basic Algorithms

#### Emptiness

Thanks to the algorithms guarantying the uniqueness of the representation up to entry names, there is only one MDD representing the empty relation, `false`.

#### Equality Testing

Because of the uniqueness of the representation, equality testing is trivial. Note that we consider that two relations with different entry names cannot be equal.

#### Inclusion Testing

The following algorithm returns `true` if the first relation, considered as a set of vectors, is a subset of the second relation. It is similar in the spirit to the `compareRel` algorithm.

```
inclusionRel(dt1, dt2):
  return(inclusionRelRec(dt1, dt2, dt1, dt2,  $\emptyset$ ))

inclusionRelRec(t, u, r1, r2, S):
  if t = false then return(true)
  if u = false then return(false)
  if u =  $\begin{array}{c} \text{iter} \\ \downarrow \\ u' \end{array}$  then
    if u' = true then return(true)
    S'  $\leftarrow$  S where all (b1, b2, o1, o2) have been replaced by (b1, false, o1, false)
    return(inclusionRelRec(t, u', r1, u', S'))
  if t =  $\begin{array}{c} \text{iter} \\ \downarrow \\ t' \end{array}$  then
    if t' = true then return(false)
    S'  $\leftarrow$  S where all (b1, b2, o1, o2) have been replaced by (false, b2, false, o2)
    return(inclusionRelRec(t', u, t', r2, S'))
  if t = true then
    S'  $\leftarrow$  S where all (b1, b2, true, o2) have been replaced by (true, b2, true, o2)
    return(inclusionRelRec(r1, u, r1, r2, S'))
  if u = true then
```

```

     $S' \leftarrow S$  where all  $(b_1, b_2, o_1, \mathbf{true})$  have been replaced by  $(b_1, \mathbf{true}, o_1, \mathbf{true})$ 
    return(inclusionRelRec( $t, r_2, r_1, r_2, S'$ ))
if  $(t, u) \in \text{dom}(S)$  then
    ( $b_1, b_2, o_1, o_2$ )  $\leftarrow S(t, u)$ 
    if  $b_1 = \mathbf{true}$  then return( $b_2 = \mathbf{true}$ ) else return(true)
 $S' \leftarrow S$  with  $(t, u) \rightarrow (\mathbf{false}, \mathbf{false}, \mathbf{true}, \mathbf{true})$ 
 $t$  is  $\begin{array}{c} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}$  and  $u$  is  $\begin{array}{c} y \\ \swarrow \searrow \\ u_0 \dots u_{m-1} \end{array}$ 
if  $m < n$  then return(false)
for  $i \leftarrow 0$  to  $n - 1$  do
    if inclusionRelRec( $t_i, u_i, r_1, r_2, S'$ ) = false then return(false)
return(true)

```

### Instantiation

Given a relation  $R$  represented by the decision diagram  $\text{dt}$  and the name tree  $\text{nt}_R$ , an entry name  $x$  in  $\text{ename}(R)$ , a possible value  $v$  for the entries named  $x$ ,  $R_{:x=v}$  is represented by:

```

instantiate( $\text{dt}, \text{nt}_R, x, v$ ):
    return(instantiateRec( $\text{dt}, \text{nt}_R, x, v, \text{dt}$ ))

instantiateRec( $t, s, x, v, r$ ):
    if  $t = \begin{array}{c} \text{iter} \\ \downarrow \\ t' \end{array}$  then return(instantiateRec( $t', s, x, v, t'$ ))
    if  $t = \mathbf{false}$  then return(false, instNameTree( $s, x$ ))
    if  $t = \mathbf{true}$  then return $\left( \begin{array}{c} \text{iter} \\ \downarrow \\ r \end{array}, \text{instNameTree}(s, x) \right)$ 
 $t$  is of the form  $\begin{array}{c} y \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}$ , and  $s$  of the form  $\begin{array}{c} z \\ \downarrow \\ s' \end{array}$ 
if  $x = z$  then
    if  $x = y$  then return(unfoldTrue( $t_v, r$ ),  $s'$ )
    else return(unfoldTrue( $t, r$ ),  $s'$ )
else if  $y = z$  then
    for  $i \leftarrow 0$  to  $n - 1$  do ( $t'_i, u$ )  $\leftarrow$  instantiateRec( $t_i, s', x, v, r$ )
    if all the  $t'_i$  are equal then return $\left( t_0, \begin{array}{c} y \\ \downarrow \\ u \end{array} \right)$ 
    return $\left( \begin{array}{c} y \\ \swarrow \searrow \\ t'_0 \dots t'_{n-1} \end{array}, \begin{array}{c} y \\ \downarrow \\ u \end{array} \right)$ 
 $(t', u) \leftarrow$  instantiateRec( $t, s', x, v, r$ )
return $\left( t', \begin{array}{c} y \\ \downarrow \\ u \end{array} \right)$ 

```



```

instNameTree( $\begin{pmatrix} y \\ \downarrow \\ s \end{pmatrix}, x$ ):
  if  $x = y$  then return( $s$ )
   $t \leftarrow$  instNameTree( $s, x$ )
  return( $\begin{pmatrix} y \\ \downarrow \\ t \end{pmatrix}$ )

unfoldTrue( $t, r$ ):
  if  $t(\varepsilon) = \text{iter}$  then return( $t$ )
  if  $t = \text{false}$  then return(false)
  if  $t = \text{true}$  then return( $\begin{pmatrix} \text{iter} \\ \downarrow \\ r \end{pmatrix}$ )

 $t$  is  $\begin{matrix} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{matrix}$ 
for  $i \leftarrow 0$  to  $n - 1$  do  $u_i \leftarrow$  unfoldTrue( $t_i, r$ )
return( $\begin{pmatrix} x \\ \swarrow \searrow \\ u_0 \dots u_{n-1} \end{pmatrix}$ )

```

#### 4.3.4 Intersection

Let  $\text{dt}_1$  and  $\text{dt}_2$  be decision trees of  $\omega$ -deterministic relations. As was announced earlier, to simplify the presentation of the algorithms, we get rid of the elimination of redundant nodes from now on.

The following algorithm returns the decision tree representing the intersection of the relations represented by  $\text{dt}_1$  and  $\text{dt}_2$ . It shows that  $\omega$ -deterministic relations are closed by finite intersection.

```

interRel( $\text{dt}_1, \text{dt}_2$ ):
  return(interRelRec( $\text{dt}_1, \text{dt}_2, \text{dt}_1, \text{dt}_2, \emptyset$ ))

interRelRec( $t, u, r_1, r_2, S$ ):
  if  $t = \text{false}$  or  $u = \text{false}$  then return(false)
  if  $t = \begin{pmatrix} \text{iter} \\ \downarrow \\ t_0 \end{pmatrix}$  then
    if  $t_0 = \text{true}$  then return(unfoldTrue( $u, r_2$ ))
     $S' \leftarrow S$  with every value  $(X, \text{old}_1, \text{old}_2, \text{new}_1, \text{new}_2)$  becomes
     $(X, \text{old}_1, \text{old}_2, \text{false}, \text{false})$ 
    if  $u = \begin{pmatrix} \text{iter} \\ \downarrow \\ u_0 \end{pmatrix}$  then  $s \leftarrow$  interRelRec( $t_0, u_0, t_0, u_0, S'$ )
    else  $s \leftarrow$  interRelRec( $t_0, u, t_0, r_2, S'$ )
    return( $\begin{pmatrix} \text{iter} \\ \downarrow \\ s \end{pmatrix}$ )
  if  $u = \begin{pmatrix} \text{iter} \\ \downarrow \\ u_0 \end{pmatrix}$  then

```

```

if  $u_0 = \mathbf{true}$  then return(unfoldTrue( $t, r_1$ ))
 $S' \leftarrow S$  with every value  $(X, \mathbf{old}_1, \mathbf{old}_2, \mathbf{new}_1, \mathbf{new}_2)$  becomes
                                      $(X, \mathbf{old}_1, \mathbf{old}_2, \mathbf{false}, \mathbf{false})$ 

 $s \leftarrow \mathbf{interRelRec}(t, u_0, r_1, u_0, S')$ 
return  $\left( \begin{array}{c} \mathbf{iter} \\ \downarrow \\ s \end{array} \right)$ 
if  $t = \mathbf{true}$  then
   $S' \leftarrow S$  with every value  $(X, \mathbf{old}_1, \mathbf{old}_2, \mathbf{new}_1, \mathbf{new}_2)$  becomes
                                                $(X, \mathbf{old}_1, \mathbf{old}_2, \mathbf{true}, \mathbf{new}_2)$ 

  return( $\mathbf{interRelRec}(r_1, u, r_1, r_2, S')$ )
if  $u = \mathbf{true}$  then
   $S' \leftarrow S$  with every value  $(X, \mathbf{old}_1, \mathbf{old}_2, \mathbf{new}_1, \mathbf{new}_2)$  becomes
                                                $(X, \mathbf{old}_1, \mathbf{old}_2, \mathbf{new}_1, \mathbf{true})$ 

  return( $\mathbf{interRelRec}(t, r_2, r_1, r_2, S')$ )
 $X$  is a new label of arity 0
if  $(t, u) \in \mathit{dom}(S)$  then
   $(Y, \mathbf{old}_1, \mathbf{old}_2, \mathbf{new}_1, \mathbf{new}_2) \leftarrow S(t, u)$ 
  if  $\mathbf{new}_1 = \mathbf{new}_2 = \mathbf{true}$  then return( $\mathbf{true}$ )
  if  $(\mathbf{old}_1 = \mathbf{true}$  or  $\mathbf{new}_1 = \mathbf{false})$  and
       $(\mathbf{old}_2 = \mathbf{true}$  or  $\mathbf{new}_2 = \mathbf{false})$  then return( $Y$ )
   $S' \leftarrow S$  with  $(t, u) \rightarrow (X, \mathbf{new}_1, \mathbf{new}_2, \mathbf{new}_1, \mathbf{new}_2)$ 
else  $S' \leftarrow S$  with  $(t, u) \rightarrow (X, \mathbf{false}, \mathbf{false}, \mathbf{false}, \mathbf{false})$ 

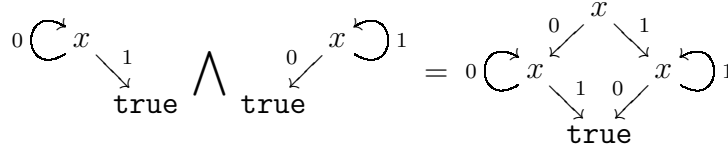
 $t$  is  $\begin{array}{c} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}$ , and  $u$  is  $\begin{array}{c} y \\ \swarrow \searrow \\ u_0 \dots u_{n-1} \end{array}$ 
for  $i \leftarrow 0$  to  $n - 1$  do  $s_i \leftarrow \mathbf{interRelRec}(t_i, u_i, r_1, r_2, S')$ 
return  $\left( \mathbf{recCons} \left( \begin{array}{c} xy \\ \swarrow \searrow \\ s_0 \dots s_{n-1} \end{array}, X \right) \right)$ 

```

**Proof:** Let  $\mathbf{dt} = \mathbf{interRel}(\mathbf{dt}_1, \mathbf{dt}_2)$ . Remember that a vector is an element of the  $\omega$ -deterministic relation represented by a decision diagram if and only if, in the decision process, after some point, we don't go through any **iter** node, and we go through an infinite number of **true** nodes. Because we add a **iter** node in  $\mathbf{dt}$  for any **iter** node in either  $\mathbf{dt}_1$  or  $\mathbf{dt}_2$ , we don't go through any **iter** node in  $\mathbf{dt}$  after some point if and only if it is the case for both  $\mathbf{dt}_1$  and  $\mathbf{dt}_2$ . If there is an infinite number of **true** in the decision process in  $\mathbf{dt}$ , then we go infinitely often through pairs of subtrees of  $\mathbf{dt}_1$  and  $\mathbf{dt}_2$ ,  $(t, u)$ , such that when going back to  $t$  and  $u$ , we have been through a **true** in both  $\mathbf{dt}_1$  and  $\mathbf{dt}_2$ . Conversely, if the vector is in both  $\mathbf{dt}_1$  and  $\mathbf{dt}_2$ , because the decision trees are regular, during the decision process performed simultaneously on both trees, we go infinitely often through at least one pair of subtrees of the decision trees,  $(t, u)$ . Because we stack an infinite number of **true**, there is an infinite number of paths that goes from  $(t, u)$  to  $(t, u)$  in the vector, such that the decision process of  $\mathbf{dt}_1$  goes through a

**true**, and an infinite number such that the decision process of  $\mathbf{dt}_2$  goes through **true**. So there is an infinite number of paths that go from  $(t, u)$  to  $(t, u)$  and goes through a **true** in both  $\mathbf{dt}_1$  and  $\mathbf{dt}_2$ . So  $R_\omega(\mathbf{dt}) = R_\omega(\mathbf{dt}_1) \wedge R_\omega(\mathbf{dt}_2)$ .  $\square$

**Example:**

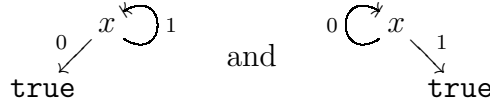


$\diamond$

### 4.3.5 Approximation

$\omega$ -deterministic relations are not closed by union. It is proved by the following example:

**Example:** Let  $R_1$  be the set of all vectors with a finite number of 1's, and  $R_2$  be the set of all vectors with a finite number of 0's.  $R_1$  and  $R_2$  are represented by:



Let  $R = R_1 \vee R_2$ . We have  $R_{[\varepsilon]}^\Omega$  is the set of all possible vectors, because  $R(\langle 0 \rangle) \simeq R$ ,  $\langle 0^\omega \rangle \in R$ ,  $R(\langle 1 \rangle) \simeq R$  and  $\langle 1^\omega \rangle \in R$ . But  $\langle (01)^\omega \rangle \notin R$ , so  $R_{[\varepsilon]}^\Omega \not\subseteq R$ , and so  $R$  is not  $\omega$ -deterministic (although it is  $\omega$ -regular).  $\diamond$

The case is not desperate, though, because this representation is intended to be used in the framework of abstract interpretation. We consider as concrete domain the set of regular relations (which are closed by all boolean operations) and as abstract domain the set of  $\omega$ -deterministic relations, each of them ordered by the inclusion of the relations. It is a consequence of the following property that there is a Galois insertion between the two domains.

**Property 4.11** *Whatever the regular relation  $R$ , there is a least  $\omega$ -regular relation  $S$  such that  $R \subset S$ .*

The least  $\omega$ -regular relation is the decision tree  $\mathbf{dt}_\omega(R)$ . As the definition of  $\mathbf{dt}_\omega(R)$  is a semi-algorithm we present the algorithm to compute the least  $\omega$ -regular relation containing the regular relation  $R$ :

```

leastDeterministic( $R$ ):
   $S$  is the function  $\emptyset \rightarrow \mathbf{false}$ 
  return(leastDeterFinite( $R, S$ ))

leastDeterFinite( $R, S$ )
  if  $R \in \text{dom}(S)$  then return( $S(R)$ )
   $X$  is a new label of arity 0
   $S' \leftarrow S$  with  $R \rightarrow X$ 
  if  $R_{[\varepsilon]}^\Omega = \emptyset$  then
     $x \leftarrow \text{name}_R 0$ , and  $n$  is the arity of  $x$ 
    for  $i \leftarrow 0$  to  $n - 1$  do  $t_i \leftarrow \text{leastDeterFinite}(R(\langle i \rangle), S')$ 
    return  $\left( \text{recCons} \left( \begin{array}{c} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}, X \right) \right)$ 
  else
     $t \leftarrow \text{leastDeterInf} \left( \text{bestOpen} \left( R_{[\varepsilon]}^\Omega \right), R, S' \right)$ 
    return  $\left( \text{recCons} \left( \begin{array}{c} \text{iter} \\ \downarrow \\ t \end{array}, X \right) \right)$ 

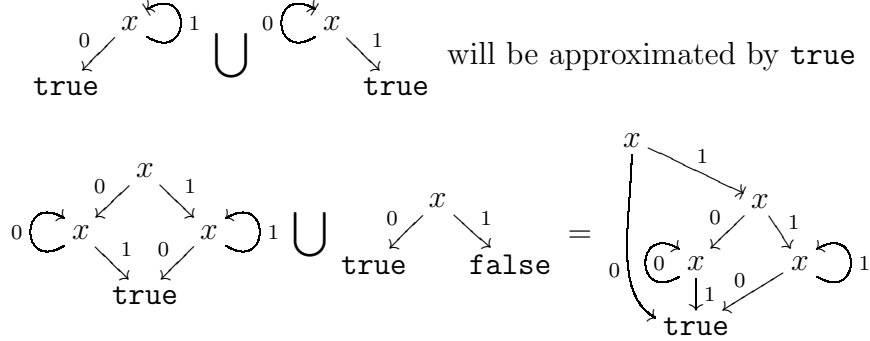
leastDeterInf( $t, R, S$ ):
  if  $t = \mathbf{false}$  then return(leastDeterFinite( $R, S$ ))
  if  $t = \mathbf{true}$  then return(true)
   $t$  is  $\begin{array}{c} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}$ 
  for  $i \leftarrow 0$  to  $n - 1$  do  $u_i \leftarrow \text{leastDeterInf}(t_i, R(\langle i \rangle), S)$ 
  return  $\left( \begin{array}{c} x \\ \swarrow \searrow \\ u_0 \dots u_{n-1} \end{array} \right)$ 

```

**Proof:** The algorithm terminates because the relation  $R$  is prefix regular. The `bestOpen` algorithms terminate because  $R$  is regular, so at a given point, the infinite behavior is prefix regular. Let  $e \in R$ . If  $e$  is ultimately periodic, because  $R$  is prefix regular, there is a  $u$  and a  $v$  such that  $e = \langle u.v^\omega \rangle$  and  $R(\langle u.v \rangle) = R(\langle u \rangle)$ . So  $e$  is recognized by the decision tree `leastDeterministic( $R$ )`. If  $e$  is not ultimately periodic, as the entry names are ultimately periodic, there is a permutation of the entries which transforms  $e$  in  $f$  which is ultimately periodic. Moreover, it is possible to choose the permutation such that it leaves  $u$  unchanged and  $f = \langle u.v^\omega \rangle$  as above.  $f$  is in  $R$ , so  $f$  is recognized by the decision tree, and by equivalence of the entries, so is  $e$ . This proves that  $R$  is contained in the  $\omega$ -deterministic relation. It is obvious that it is the least such  $\omega$ -deterministic relation.  $\square$

A consequence of this property is that we can compute the least  $\omega$ -deterministic relation containing the union of two  $\omega$ -deterministic relations  $\mathbf{dt}_1$  and  $\mathbf{dt}_2$ .

Examples:



◇

This operation is performed by the following algorithm:

```

unionRel(dt1, dt2):
  return(unionRelFinite(dt1, dt2, dt1, dt2, ∅))

unionRelFinite(t, u, r1, r2, S):
  if t = false then return(unfoldTrue(u, r2)
  if u = false then return(unfoldTrue(t, r1)
  if t = true then t ← r1
  else if t =  $\downarrow_{t_0}^{iter}$  then
    if t0 = true then return(t)
    r1 ← t ← t0
  if u = true then u ← r2
  else if u =  $\downarrow_{u_0}^{iter}$  then
    if u0 = true then return(u)
    r2 ← u ← u0
  if (t, u) ∈ dom(S) then return(S(t, u))
  X is a new label of arity 0
  S' ← S with (t, u) → X
  t is  $\begin{matrix} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{matrix}$ , and u is  $\begin{matrix} y \\ \swarrow \searrow \\ u_0 \dots u_{n-1} \end{matrix}$ 
  for i ← 0 to n - 1 do
    (si, bi) ← unionRelInf(ti, ui, r1, r2, S', t, u, false, false)
  s ← recCons  $\left( \begin{matrix} xy \\ \swarrow \searrow \\ s_0 \dots s_{n-1} \end{matrix}, X \right)$ 
  if  $\bigvee_{i < n} b_i = \text{false}$  then return(s)
  else return  $\left( \downarrow_s^{iter} \right)$ 

```

```

unionRelInf( $t, u, r_1, r_2, S, \text{iter}_1, \text{iter}_2, \omega_1, \omega_2$ ):
  if  $t = \text{false}$  then return(unfoldTrue( $u, r_2$ ), false)
  if  $u = \text{false}$  then return(unfoldTrue( $t, r_1$ ), false)
  if  $t = \text{true}$  then
     $t \leftarrow r_1$ 
    if  $\omega_1 = \text{false}$  then  $\omega'_1 \leftarrow \text{true}$ 
    else  $\omega'_1 \leftarrow \omega_1$ 
  else if  $t = \begin{matrix} \text{iter} \\ \downarrow \\ t_0 \end{matrix}$  then
    if  $t_0 = \text{true}$  then return( $t, \text{false}$ )
     $r_1 \leftarrow t \leftarrow t_0$ 
     $\omega'_1 \leftarrow \text{iter}$ 
  else  $\omega'_1 \leftarrow \omega_1$ 
  if  $u = \text{true}$  then
     $u \leftarrow r_2$ 
    if  $\omega_2 = \text{false}$  then  $\omega'_2 \leftarrow \text{true}$ 
    else  $\omega'_2 \leftarrow \omega_1$ 
  else if  $u = \begin{matrix} \text{iter} \\ \downarrow \\ u_0 \end{matrix}$  then
    if  $u_0 = \text{true}$  then return( $u, \text{false}$ )
     $r_2 \leftarrow u \leftarrow u_0$ 
     $\omega'_2 \leftarrow \text{iter}$ 
  else  $\omega'_2 \leftarrow \omega_2$ 
  if  $\omega'_1 = \omega'_2 = \text{iter}$  then
     $S' \leftarrow S$  with  $(\text{iter}_1, \text{iter}_2) \rightarrow \begin{matrix} \text{iter} \\ \downarrow \\ S(\text{iter}_1, \text{iter}_2) \end{matrix}$ 
    return(unionRelFinite( $t, u, r_1, r_2, S'$ ), false)
  if  $(t, u) = (\text{iter}_1, \text{iter}_2)$  then
    if  $\omega'_1 = \text{true}$  or  $\omega'_2 = \text{true}$  then return(true, true)
    else return( $S(t, u), \text{false}$ )
  if  $(t, u) \in \text{dom}(S)$  then return( $S(t, u), \text{false}$ )

```

$X$  is a new label of arity 0

$S' \leftarrow S$  with  $(t, u) \rightarrow X$

$t$  is  $\begin{matrix} x \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{matrix}$ , and  $u$  is  $\begin{matrix} y \\ \swarrow \searrow \\ u_0 \dots u_{n-1} \end{matrix}$

**for**  $i \leftarrow 0$  **to**  $n - 1$  **do**

$(s_i, b_i) \leftarrow \text{unionRelInf}(t_i, u_i, r_1, r_2, S', \text{iter}_1, \text{iter}_2, \omega'_1, \omega'_2)$

**if**  $\bigvee_{i < n} b_i = \text{false}$  **then**

$S' \leftarrow S$  with  $(\text{iter}_1, \text{iter}_2) \rightarrow \begin{matrix} \text{iter} \\ \downarrow \\ S(\text{iter}_1, \text{iter}_2) \end{matrix}$

**return**(unionRelFinite( $t, u, r_1, r_2, S'$ ), **false**)

**else return**  $\left( \text{recCons} \left( \begin{array}{c} xy \\ \swarrow \quad \searrow \\ s_0 \quad \dots \quad s_{n-1} \end{array}, X \right) \right)$

## 4.4 Other Representations

Decision diagrams have proved very useful for finite relations, and they seem quite efficient for infinite relations too. But other representations might have different qualities, for example more efficiency and less expressive power, or different operations that can be performed more easily. Because we separate the representation of the relations from the representation of the structure of the sets of trees, we can parameterize the representation of the sets of trees by the representation of relations as defined at the beginning of this chapter. We present briefly two such representations that can be of interest.

### 4.4.1 A Set of Vectors is a Set of Trees

A relation is defined as a set of vectors. But a set of vectors is a set of words, which are trees, so we can use the techniques developed in this thesis to represent sets of trees in order to represent relations. This is legitimate, because these techniques don't use relations in the representation of sets of words, except possibly for the infinite behavior.

Of course, this representation will be less efficient than MDD, because it is not specialized on relations. In fact, it looks like a MDD without the **false** node (see the small example). But we introduce in chapter 7 the possibility of having counters in the representation. This might add interesting possibilities of widening on sequences of relations. Such widening would be more intelligent than the crude ones of [Mau98] for finite relations. The decision to use this representation in addition to decision diagrams (which are still needed for the infinite behavior) would reduce the efficiency but give more precise results. Its usefulness depends on the application.

### 4.4.2 Exact Relations

The most common relation introduced in the representation of sets of trees is the equality relation. So we propose a class of relations containing equality relations, simple enough to have a more efficient representation than decision diagrams, and such that they can approximate any relations.

Let  $R$  be a relation of domain  $\bigotimes_{i \in I} E_i$  with each  $E_i$  finite. We say that  $R$  is a *basic exact* relation if and only if there is a family  $(E_{ij})_{i \in I, j \in J}$  such that  $R = \bigcup_{j \in J} \bigotimes_{i \in I} E_{ij}$  and for any  $i$ ,  $E_{ij} \cap E_{ij'} \neq \emptyset \Leftrightarrow j = j'$ . The relation is said to be *exact* if it can be independently decomposed in basic exact relations.

A basic exact relation can be represented using the words  $(E_{i,j})_{i \in I}$  and as usual the word representing the naming of the entries. As a consequence, we can represent infinite basic exact relations such that these words are ultimately periodic. In this case, we can use the techniques to represent a regular tree developed in chapter 3.

**Examples:** The relation  $x = y = z$  over the domain  $[2]$  is represented by the representation of the names,  $xyz$ , and the set  $\{000, 111, 222\}$ .

The infinite relation such that all entries  $2i$  equals the entry  $2i + 1$  will be represented by  $\begin{matrix} x \\ \downarrow \\ y \end{matrix}$  and the set  $\{00, 11\}$ . Note the difference with the relation where all entries are equal:  $x \curvearrowright$  and  $\{0 \curvearrowright, 1 \curvearrowright\}$ .  $\diamond$

This class of relations is not closed under union, but it is closed by intersection and we can find the exact closure and the exact part of any relation. It means that we can find a least exact relation containing a given regular relation, as for  $\omega$ -deterministic relations.

Such a representation might be used when we want more efficiency at the expense of the precision.

## 4.5 Conclusion

This chapter presented the class of relations which will be used in the representation of sets of trees, that is regular relations. Such relations might be infinite and we introduced the notion of equivalent entries to cope with the problem of representing infinite relations. Regular relations are nearly as expressive as  $\omega$ -regular languages of Büchi. They are closed by all boolean operations. They can be represented by reduced decision diagrams in the way BDD are represented, but the representation of regular relations in general is not very efficient [SVW85]. Thus we introduced a subclass of regular relations, the  $\omega$ -deterministic relations which can be used as an approximation (for relation inclusion) of regular relations. Their representation as a reduced decision diagram is canonical. They are an extension of the BDDs and have great expressive power for the infinite behavior of the relation. Because this class seems to give a good tradeoff between accuracy and efficiency, we developed the algorithms that are used to manipulate  $\omega$ -deterministic relations. They can be used as such to represent finite or infinite relations over finite domains. They will be used in the next chapter for an efficient representation of sets of trees.



# Chapter 5

## Tree Schemata

In this chapter, we attack the problem central to this thesis, namely representation of sets of trees. To be more precise, we try to represent any (or as many as possible) possibly infinite set of possibly infinite trees. To concentrate on the set of trees representations problem, we suppose that we only have a finite ranked set of labels. In this chapter,  $F$  will denote this set.

Because the problem is difficult, there can be different solutions, each of them with their advantages and drawbacks. It is the reason why we start this chapter by a presentation of other possible approaches, either found in the literature or explored during the course of the thesis. After this presentation, we expose our solution, which we call tree schemata.

A tree schema can be decomposed in two parts, the skeleton and the links. The skeleton can be seen as an upper approximation<sup>1</sup>. This approximation is easy to understand and most operations will be very efficient on the skeleton. The links are a way to reduce the approximation at the cost of a worse complexity.

### 5.1 Different Approaches

#### 5.1.1 Representations Based on Automata

The most classical way of representing a regular set of trees is the tree automaton. Tree automata for finite trees have been studied for a long time [TW68, GS84]. Although their practical use is not so easy, they have been used in practice, and efficiency of the implementations have been investigated [BMW91, HJJ<sup>+</sup>96, BKR97]. The problem is that bottom-up tree automata, the most general ones, are complex to manipulate<sup>2</sup>. For example, Seidl showed in

---

<sup>1</sup>That is, the skeleton represents a superset of the set represented by the schema.

<sup>2</sup>In fact, in order to have a realistic implementation, a deterministic top-down tree automaton is used in [BKR97] to partition the set of trees, each subset being represented by a more powerful bottom-up tree automaton.

[Sei90] that deciding the equivalence of two tree automata is EXPTIME-hard. Moreover, tree automata lack relational expressive power, that is they cannot represent sets of terms like  $\left\{ \begin{array}{c} f \\ \swarrow \downarrow \\ t \quad t \end{array} \mid t \in \mathcal{H}(F) \right\}$ . Of course, they cannot be used to represent infinite trees either.

To increase the expressive power of tree automata, we can add equality constraints between subtrees, as proposed in [MS81]. The problem is that with this extension the emptiness problem is no longer decidable. It seems that these constraints must be restricted to brother subtrees for this problem to be decidable [BT92]. To represent infinite trees, we can use infinite tree automata, as proposed in [Rab69]. These automata are even more complex [Saf88], and although their potential practical usefulness has been pointed out many times [Var94, Sch90, ES88], I do not know of any efficient implementation<sup>3</sup>.

Another problem with tree automata (and even word automata) is the difficulty of approximating accurately limits of sequences of automata (of growing size for example). We tried to investigate automata based on cycles, where a tree is presented as a cycle of trees, these trees being presented as cycles until we come to basic cycles. In this way, we could have detected the growth of any level of cycle and approximated it by an infinite cycle, or approximate the depth of these cycles<sup>4</sup>. To give an idea of this kind of automata, we present the decomposition in cycles of a regular infinite word. A basic cycle is a finite or infinite sequence of the same letter, say  $a$ . It is represented by  $a^n$  if the sequence is finite of length  $n$  and  $a^\omega$  if the sequence is infinite. If a word is not a basic cycle, it is represented by a number, a letter and a tuple of words, one for each different letter in the word. The number and the letter in this representation are the first letter of the word and the number of times it appears before the first other letter of the word. Each word in the tuple corresponds to the followers of the letter with which it is associated. The letters of these words are the basic cycles representing the followers. These words are themselves represented in the same way. For example, the word  $(ababb)^\omega$  is represented by  $\left(1, a, b \rightarrow a^{1\omega}, a \rightarrow \left(1, b^1, b^1 \rightarrow b^{2^{1\omega}}, b^2 \rightarrow b^{1^{1\omega}}\right)\right)$ . In the case of trees, we must use bigger ordinal numbers to deal with infinite cycles containing infinite cycles. We don't describe these automata (or in fact hyper-automata) further because our investigations showed that these representations, although very precise, are too inefficient. For example tree substitution is exponential. This is because we tried to achieve too much sharing, and for each little change in the set, we had to perform too much calculation for a very high sharing.

---

<sup>3</sup>In fact, even infinite words automata seem difficult, and other representations may be used to reduce the complexity (*e.g.* [CNP93]).

<sup>4</sup>In [Ven97], Arnaud Venet uses classical word automata with counters on productions of the automata to deal with cycles.

### 5.1.2 Representations Based on Expressions

A first motivation to tree automata [TW68] was to prove decidability of some logics. Thus sets of trees can be represented by logical formulae using expressions on set of trees variables. To represent sets of possibly infinite trees,  $\mu$ -calculus [EC80, Koz83, AN92] seems a good framework. The problem is that it seems better as a description (by the user or for the user) than as a representation. It is the same problem as with boolean functions: it is easier to write or read a boolean formula, but it is well-known such formulae are not well-suited for the manipulation of boolean functions. That is the reason why, when manipulating sets of trees,  $\mu$ -calculus formulae are usually translated into tree automata<sup>5</sup>, or we just consider fragments of the  $\mu$ -calculus [CS91, EJS93].

But with the notion of expressions on set of trees variables, we can go further. Exploiting the fact that variables represent sets, we can express sets constraints over these variables. The first use of this notion is in regular tree grammars, where non-terminals can be seen as the variables, productions as inclusion constraints, and right-hand sides as simple expressions with just union and terms. In fact, such representations of sets of trees have been used (for example in [Rey69, JM79, And86, Sør94]) to approximate more complex sets of trees. This representation, though, is far from ideal, mainly because of the use of variable names. Indeed this extensive use of variable names raises the problem of the uniqueness of the representation. It leads to inefficiency when comparing two sets of trees, and when performing some basic operations. For example intersection may require the creation of an exponential number of new variable names (in the implementation of regular tree expressions of [AM91] for example). Of course, we still lack the relational expressive power.

Other systems of constraints with more complex expressions have been studied since the reformulation of some program analyzes in term of “set based analysis” by Heintze [Hei92]. The representations they propose suffer from the same drawback as regular tree grammars (with the additional constraint, if we follow their formalism, that we must use the approximation of no relation between variables). For example, in [HJ90], there is a new variable for each union, and intersection requires an exponential number of variables. Thanks to the studies motivated by this work, there are many results of theoretical complexity or decidability of many set constraints systems. For example, in [AW92], Aiken and Wimmers showed that sets of constraints with union and intersection lead to an emptiness problem of complexity EXPTIME-hard. To represent infinite trees, we can use the co-definite set constraints studied in [CP98a], but the complexity is of course still EXPTIME-hard. A survey of different systems of set constraints is proposed in [Aik94]. Some extensions are studied in [Cha94], but they don’t seem very encouraging. In order to achieve efficiency, some authors propose the use of tree

---

<sup>5</sup>In fact, Emerson and Jutla showed in [EJ91] that the  $\mu$ -calculus is as expressive as infinite tree automata.

automata [DTT97], as for the  $\mu$ -calculus.

In a more expressive framework, sets of trees have been studied in [AN80] from the point of view of recursive program schemes. Such program schemes can have non deterministic choice, and as such look like the skeletons we introduce in this chapter. They are a lot more expressive, as they amount to context-free tree grammars. Such models have been studied from a theoretical point of view, and to my knowledge, no implementation uses program schemes.

### 5.1.3 Tree Schemata

Tree schemata are based on skeletons, which can be seen as tree automata. The first difference between skeletons and classical automata is that it is the states of the skeleton that are labeled, and not the arrows. A first consequence is that the skeleton is closer to the trees of the set it represents. For instance, the skeleton representing one tree is the tree itself. Skeletons have the expressive power of deterministic top down tree automata (except for the unusual greatest fixpoint semantics). They share the common prefixes and subtrees of the trees they represent.

Full tree schemata are skeletons augmented with links. This decomposition separates the tree structure of the set of trees from the relations induced by the set of trees. For example, in the set  $\left\{ \begin{array}{c} f \\ \swarrow \searrow \\ a \quad b \end{array}, \begin{array}{c} f \\ \swarrow \searrow \\ c \quad d \end{array} \right\}$ , the tree structure is that every tree starts with a  $f$ , and its first child is either  $a$  or  $c$ , and its second child  $b$  or  $d$ . The relation is that when the first child is a  $a$ , the second child is a  $b$ , and when it is a  $b$ , the second child is a  $d$ . This separation has two advantages: first it allows structural sharing of the trees, and second we can use the expressive power of regular relations. It allows for example the representation of the set  $f(a^n, b^n, c^n)$ ,  $n \in \mathbb{N}$ . The automaton part of the representation (either in the tree structure or the relations) is always maintained in a kind of minimal state number, through the use of the techniques of chapter 3.

## 5.2 The Skeleton

The skeletons are very intuitive representations of some sets of trees. They are the structures on which links are built to represent more complex sets. In the sequel, the set of all possible skeletons for a tree schema over  $F$  will be denoted  $Sk(F)$ .

### 5.2.1 A Set of Trees is a Tree

The easiest way of representing a set of objects consists in representing the elements of the set, and then using a sequence of arrows towards these representa-

tions. This presents two drawbacks: The sequential aspect of the representation allows many ways of representing the same set; the representation does not forbid the same object to be represented twice in a given set.

In our case, though, this method shows a very useful quality: a sequence of arrows is nothing less than a tree and representing a tree is easy. Thus, to represent a set of trees, we start by adding a new label to  $F$ ,  $\bigcirc$ , called the choice label and representing a kind of union. This new label does not have any fixed arity, but we shall see in the sequel that we can consider this label as representing  $|F| + 1$  labels  $(\bigcirc_i)_{i \leq |F|}$ ,  $\bigcirc_i$  being of arity  $i$ . The set of trees labeled by  $F \cup \{\bigcirc\}$  with this special convention for the arity of  $\bigcirc$  will be denoted  $\mathcal{H}_{\bigcirc}(F)$ .

The set of trees represented by a tree of  $\mathcal{H}_{\bigcirc}(F)$  is defined by the function  $Ens : \mathcal{H}_{\bigcirc}(F) \rightarrow \wp(\mathcal{H}(F))$ :

$$\forall f \in F, Ens \left( \begin{array}{c} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array} \right) \stackrel{\text{def}}{=} \left\{ \begin{array}{c} f \\ \swarrow \searrow \\ u_0 \dots u_{n-1} \end{array} \mid \forall i < n, u_i \in Ens(t_i) \right\}$$

$$Ens \left( \begin{array}{c} \bigcirc \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array} \right) \stackrel{\text{def}}{=} \bigcup_{i < n} Ens(t_i)$$

The functions of  $\mathcal{H}_{\bigcirc}(F) \rightarrow \wp(\mathcal{H}(F))$  are ordered pointwise by the inclusion of the image. With this ordering,  $\mathcal{H}_{\bigcirc}(F) \rightarrow \wp(\mathcal{H}(F))$  is a complete lattice and the function defining  $Ens$  is monotone, so its greatest fixpoint is well defined.

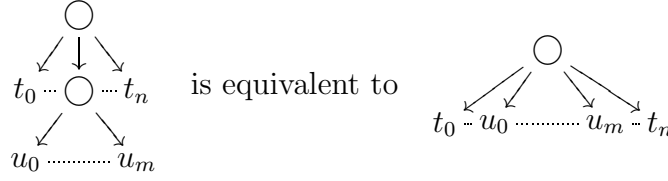
In order to solve the uniqueness problem of this representation, when keeping the same easy semantic, the set of trees of  $\mathcal{H}_{\bigcirc}(F)$  considered as valid must be restrained. These restrictions concern the subtrees of the choice nodes only.

Three types of restrictions must be considered to define the final version of the tree schemata skeletons. The obvious restrictions suppress the trees that are equivalent to smaller trees. The conventional restrictions choose some trees among many equivalent ones of the same size. Finally, the simplification restrictions decrease the class of sets of trees that can be represented by a skeleton.

### Obvious Restrictions:

Starting from any element of  $\mathcal{H}_{\bigcirc}(F)$ , it is possible to build a sequence of elements of  $\mathcal{H}_{\bigcirc}(F)$  of increasing size, representing the same tree of  $\mathcal{H}(F)$ . We can always add useless choice nodes for example. Two kinds of useless choice nodes can be distinguished: the choices of a unique subtree and successive choices. These kind of choices and their simplified equivalents are:

$$\begin{array}{c} \bigcirc \\ \downarrow \\ t \end{array} \text{ is equivalent to } t$$

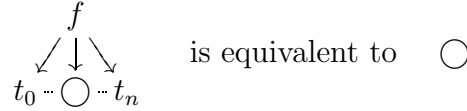


To avoid these unwanted trees, the following rules are respected by every skeleton of tree schemata of  $F$ :

**Rule 1 :**  $\forall t \in \mathcal{Sk}(F), \forall \begin{array}{c} \circ \\ \swarrow \downarrow \searrow \\ t_0 \dots t_{n-1} \end{array} \triangleleft t, \quad n > 1$

**Rule 2 :**  $\forall t \in \mathcal{Sk}(F), \forall \begin{array}{c} \circ \\ \swarrow \downarrow \searrow \\ t_0 \dots t_{n-1} \end{array} \triangleleft t, \forall i < n, \quad t_i(\varepsilon) \neq \circ$

We can also build a sequence of elements of  $\mathcal{H}_\circ(F)$  of increasing size starting from  $\circ$  of zero arity, which represents the empty set. The equivalence is:



The corresponding rule, assuming rule number 2 is enforced, is:

**Rule 3 :**  $\forall t \in \mathcal{Sk}(F), \text{ if } \circ \triangleleft t, \text{ then } t = \circ$

### Conventional Restrictions:

Another problem with this representation —we already mentioned it— is its sequential aspect. In order to limit the number of possible representations of a given set, when possible, an order relation on the elements of the set can be used. The restriction consists in considering only increasing sequences of object instead of any sequence. If the ordering is total, the representation is then unique. In addition to allowing the sharing of equal sets, this convention allows algorithms on the sets to use the ordering.

This technique can be efficient if the algorithm deciding the order between the elements is fast. In the case of sets of trees, it would be too costly in the perspective of a first approximation to use a total ordering that could require to cover the entire trees before deciding which one is the smaller. Thus a mere preorder will be used to reduce the number of possible representations.

Let  $<_F$  be a total ordering on  $F$ . The preorder imposed to the subtrees of a choice node will be the order of their root. Thus the rule:

**Rule 4 :**  $\forall t \in \mathcal{Sk}(F), \forall \begin{array}{c} \circ \\ \swarrow \downarrow \searrow \\ t_0 \dots t_{n-1} \end{array} \triangleleft t, \forall 0 < i < n, \quad t_{i-1}(\varepsilon) \leq_F t_i(\varepsilon)$

### Simplification Restrictions:

This time, only elements of  $\mathcal{H}_\circ(F)$  that represent efficiently some sets of trees will be chosen. The choice made in this thesis consists in favoring prefix sharing.

Incidentally, this restriction allows the resolution of the uniqueness problem of the representation. The rule is the following:

**Rule 5 :**  $\forall t \in \mathcal{Sk}(F), \forall \begin{array}{c} \circ \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array} \triangleleft t, \forall i < n, \forall j < n, \quad i \neq j \Leftrightarrow t_i(\varepsilon) \neq t_j(\varepsilon)$

This rule may suppress some representations that are equivalent to smaller valid ones, but in general it forces to approximate some sets. For example  $\begin{array}{c} \circ \\ \swarrow \searrow \\ g \quad g \\ \downarrow \downarrow \\ a \quad b \end{array}$

and  $\begin{array}{c} g \\ \downarrow \\ \circ \\ \swarrow \searrow \\ a \quad b \end{array}$  represent the same set, but  $\begin{array}{c} \circ \\ \swarrow \searrow \\ f \quad f \\ \swarrow \searrow \quad \swarrow \searrow \\ a \quad b \quad c \quad d \end{array}$  cannot be represented in  $\mathcal{Sk}(F)$ .

The best upper approximation will be  $\begin{array}{c} f \\ \swarrow \searrow \\ \circ \quad \circ \\ \swarrow \searrow \quad \swarrow \searrow \\ a \quad c \quad b \quad d \end{array}$ . This approximation may seem crude, but it is commonly used in set based analysis [HJ90] under the name of cartesian approximation.

### 5.2.2 Definition of the Set of Tree Schema Skeletons

The set of tree schema skeletons is denoted  $\mathcal{Sk}(F)$ . It is defined using rules 1 to 5:

$$\mathcal{Sk}(F) \stackrel{\text{def}}{=} \left\{ t \in \mathcal{H}_\circ(F) \left| \begin{array}{l} t \text{ is regular and } \forall \begin{array}{c} \circ \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array} \triangleleft t, \quad n > 1 \quad \wedge \\ \forall i < n, t_i(\varepsilon) \neq \circ \quad \wedge \quad \forall 0 < i < n, t_{i-1} <_F t_i \end{array} \right. \right\} \cup \{\circ\}$$

The set of trees represented by a skeleton is defined using the same function  $Ens$  as the other elements of  $\mathcal{H}_\circ(F)$ . The set of sets of trees representable by a skeleton will be denoted  $\mathcal{Sk}^\cdot(F)$ . Its formal definition is:

$$\mathcal{Sk}^\cdot(F) \stackrel{\text{def}}{=} \{A \subset \mathcal{H}(F) \mid \exists s \in \mathcal{Sk}(F), Ens(s) = A\}$$

Thanks to the rules defining the valid skeletons, every set of trees representable by a skeleton is uniquely represented. So the representation allows maximal sharing of prefix parts and subtrees of these sets. In this way, it is at the same time compact and easy to use.

Note that the representation of a singleton  $\{t\}$  is the same as the representation of  $t$ , even if  $t$  is an infinite regular tree, thanks to the greatest fixpoint semantics of the skeletons.

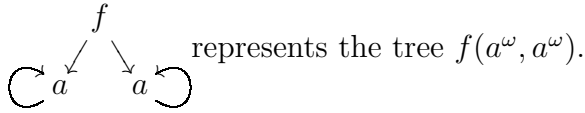
The empty set is in  $\mathcal{Sk}^\cdot(F)$ . It is represented by  $\circ$ . The set of all trees  $\mathcal{H}(F)$  is also in  $\mathcal{Sk}^\cdot(F)$ , because  $F$  is finite. It is represented by a choice node, with all possible labels as its children, and each child then goes back to the choice

node. For example, if  $F = \{a, b, f\}$  of arity 0, 0 and 2,  $\mathcal{H}(F)$  is represented by



We denote this skeleton  $\mathbf{1}_F$ . The main flaw of this representation is its lack of algebraic property:  $\mathcal{Sk}(F)$  is not closed for union, neither for infinite intersection. In particular, there is no best approximation of every set of  $\mathcal{H}(F)$  by an element of  $\mathcal{Sk}(F)$ .

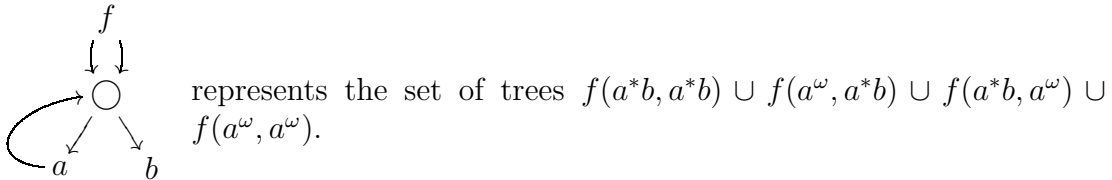
**Examples:**



represents the tree  $f(a^\omega, a^\omega)$ .



represents the set  $a^*b \cup a^\omega$ .



represents the set of trees  $f(a^*b, a^*b) \cup f(a^\omega, a^*b) \cup f(a^*b, a^\omega) \cup f(a^\omega, a^\omega)$ .

◇

### 5.2.3 The Choice Nodes

The use of rule 4 is not the only possibility to ensure the uniqueness of the representation. It allows the representation of the set of labels in the choice as a vector (following the total ordering). But when the set of labels is really big, it is not the best representation possible for this set. An alternative representation is the use of a BDD ([Bry92]), but this representation uses variable names which may jeopardize the canonicity of the representation. Moreover, we don't need BDDs, but multi-valued BDDs, that is BDDs with other leaf values than **true** and **false**. Such BDDs are less efficient than classical ones.

We have the possibility of using as variable names for the choice nodes the same names as the variables representing the entry<sup>6</sup> corresponding to the choice node in the relation which is linked to the choice node (see next section). This implementation gives a satisfying uniformity which can ease some operations on tree schemata. The problem is that it forces the range of the choice to be the

<sup>6</sup>See Sect. 4.2.4, page 63, for the possibility of using many binary variables instead of one variable ranging over a large set, thus transforming an MDD into a BDD.



whole set of labels, which might be much bigger than the actual choice at this point. The result is a bigger relation representing the link.

The choice between an ordered set and a BDD to represent the choice node is a parameter of the tree schemata. We can choose one of the possibilities depending on the kind of sets of trees we represent. If the set of trees present choice nodes over a small number of labels in general, we should use the ordered sets.

## 5.3 The links

As the tree schema skeletons are not very powerful, they must be enriched to be able to represent a greater diversity of sets of trees. The idea of the links of the tree schemata is to increase the number of sets a skeleton can represent by suppressing some elements of these sets. Thus the skeleton can be seen as an upper approximation of the set represented by the tree schema, and the links as a set of constraints on the set represented by the skeleton.

### 5.3.1 The Choice Space

In order to restrain the elements of the sets represented by a tree schema, we must define the set of possible restrictions. Restrictions will only be imposed on choices. Let  $\underset{t_0 \dots t_{n-1}}{\vee} \overset{\circ}{\vee}$  be a choice node. The choice space of this node will be described by a choice in  $[n]$ . Now, let  $S$  be a skeleton. The set of paths of  $S$  labeled by a choice will be denoted:

$$cps(S) \stackrel{\text{def}}{=} \{p \in pos(S) \mid S(p) = \overset{\circ}{\vee}\}$$

Let  $p \in cps(S)$ . Then  $S_{[p]} = \underset{t_0 \dots t_{n-1}}{\vee} \overset{\circ}{\vee}$ . We define  $choice(p) \stackrel{\text{def}}{=} [n]$ . The *choice space* of  $S$ ,  $\mathcal{CS}(S)$ , is defined by:

$$\mathcal{CS}(S) \stackrel{\text{def}}{=} \bigotimes_{p \in cps(S)} choice(p)$$

A skeleton  $S$  can be seen as a function  $\langle S \rangle : \mathcal{CS}(S) \rightarrow \mathcal{H}(F)$  defined by:

$$\langle S \rangle \stackrel{\text{def}}{=} \widetilde{\langle S \rangle}(\varepsilon)$$

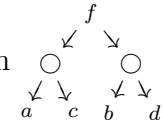
and  $\forall p \in pos(S), \forall c \in \mathcal{CS}(S)$ :

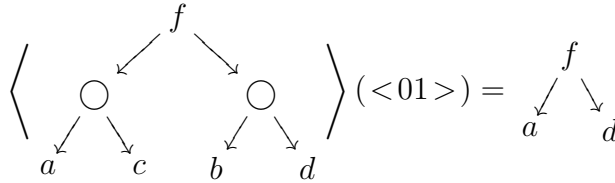
$$\begin{aligned} \forall f \in F, \left\langle \widetilde{\underset{t_0 \dots t_{n-1}}{\vee} \overset{f}{\vee}} \right\rangle(p)(c) &\stackrel{\text{def}}{=} \underset{u_0 \dots u_{n-1}}{\vee} \overset{f}{\vee} \text{ such that } \forall i < n, u_i = \widetilde{\langle t_i \rangle}(p.i)(c) \\ \left\langle \widetilde{\underset{t_0 \dots t_{n-1}}{\vee} \overset{\circ}{\vee}} \right\rangle(p)(c) &\stackrel{\text{def}}{=} \left\langle \widetilde{\langle t_{c(p)} \rangle} \right\rangle(p.c(p))(c) \end{aligned}$$

That gives a new possible definition for  $Ens$ :

$$Ens(S) = \{ \langle S \rangle (c) \mid c \in \mathcal{CS}(S) \}$$

It is easy then to reduce the size of the set represented by a skeleton, by reducing its choice space.

**Example:** Let  $S$  be the skeleton . Then  $cps(S) = \{0, 1\}$ ,  $choice(0) = [2]$ ,  $choice(1) = [2]$ . So the choice space of  $S$  is  $[2]_0 \times [2]_1$ .



◇

### 5.3.2 Links as Relations

Reducing the choice space of a skeleton is choosing a subset of this space. A subset of a cartesian product is a relation. A relation may be decomposed in independent smaller relations. When it is the case, the smaller relations take less space than the original one, and moreover they carry a more local information. The links of a tree schema are based on these independent relations over subsets of the set of choice nodes of the skeleton. The idea of decomposing the global relations into smaller ones leads to a more compact representation even if the decomposition is not independent (see [BCL91, HD93, GB94] for a study in the case of relations represented by BDDs).

The different ways of representing a relation are discussed in chapter 4. These different ways of representing relations might have different expressive power, thus leading to tree schemata with different expressive power. In order to keep things simple we represent in this chapter every link through the use of basic multiple decision diagrams. A very common relation is the equality relation. We will write it =.

### 5.3.3 Links are Local

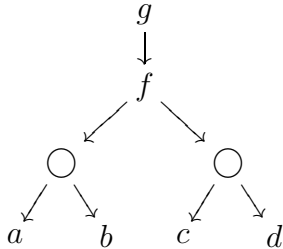
As a first description, a tree schema can be seen as a skeleton together with a global relation on the choice nodes of the skeleton represented by a set of links.

But if these links are just the independent relations, then their indexes will be the global paths of the skeleton. Then, they must be changed each time a single part of the skeleton is changed, and they cannot be shared.

In order to give them a better autonomy, links are represented by relations with set of indexes an ordinal, which is always possible according to chapter 4. Each choice which should be restrained by a particular relation is just associated with the corresponding link and to the entry in the link. In this way, links are local to the particular choice nodes they put in relation and they don't have to be modified as long as these nodes are left unchanged.

Now we can give a new way of representing a set of tree: a tree schema  $T$  on  $F$  is a tree on  $F \cup \{(\bigcirc, e, l) \mid l \text{ is a link, and } e \text{ is an entry of } l\}$ . Of course we will have to restrict this definition to characterize tree schemata. We can still give some basic definitions that do not depend on these restrictions. The skeleton of  $T$ ,  $skel(T)$ , is the tree obtained from  $T$  by changing every node  $(\bigcirc, e, l)$  into  $\bigcirc$ . The link associated with a choice node  $C$  of  $T$  is denoted  $link(C)$ .

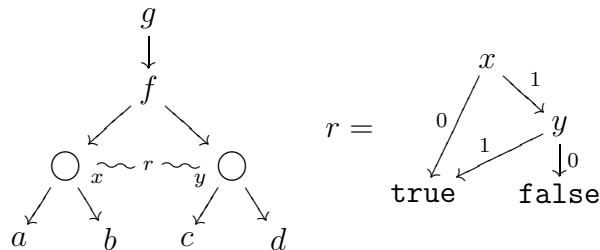
**Example:** Consider the following skeleton:



Its choice space is  $[2]_{00} \times [2]_{01}$ . A possible restriction would be to consider the set

$$\left\{ \begin{array}{c} g \\ \downarrow \\ f \\ \swarrow \searrow \\ a \quad c \end{array}, \begin{array}{c} g \\ \downarrow \\ f \\ \swarrow \searrow \\ a \quad d \end{array}, \begin{array}{c} g \\ \downarrow \\ f \\ \swarrow \searrow \\ b \quad d \end{array} \right\}.$$

The associated global relation would be  $\{0_{00}0_{01}, 0_{00}1_{01}, 1_{00}1_{01}\}$ . The local link  $l$  would be based on the relation  $r = \{0_x0_y, 0_x1_y, 1_x1_y\}$  and in the tree schema, the first choice node would be associated with  $(x, l)$  and the second one with  $(y, l)$ . It would be represented this way:



◇

### 5.3.4 The Names of the Links

In order to manipulate links, to associate them with choice nodes, and in particular to share some of the links, we must give them a name. As we want to avoid as much as possible the use of arbitrary (variable) names, we use a part of the tree schema itself. The only way to distinguish between any global relation decomposed independently as much as possible is through the use of global paths. As we want to avoid the use of such a global information that would require a too costly maintenance, we can't distinguish between any set of partial relation. This means that in some cases, links will be based on relations that could be furtherly independently decomposed, and we can't divide them in smaller links. This will be illustrated in the next paragraph.

The minimum amount of information we need to know about a link is the relation it is based on. As we already pointed out, we would not have enough information in the tree schema for some configurations of links if we just used the relation as the name of the link. Suppose for example that we have two different sets of choice nodes linked by the same relation: we could not infer from the tree schema which sets are linked.

**Example:** Suppose we have four choice nodes of global paths  $p_1$ ,  $p_2$ ,  $p_3$  and  $p_4$  with the same choice space. If the relation binding these nodes is  $p_1 = p_2$  and  $p_3 = p_4$ , then we can't divide it in two smaller relations, because these relations being the same, we would not know whether  $p_1$  is linked to  $p_2$  or  $p_4$ .  $\diamond$

The most local information we can gather about a link is the relation it is based on and the nodes or subtrees of the tree schema it binds. It is this information we use as the name of the links. There are still some cases where we cannot decompose the relation as much as possible, but in the next section on sharing, we will discuss some of these cases where we can distinguish between shared link names. In the graphic representation of tree schemata, a link is represented by a wavy line between the subtrees it binds, labeled by the relation.

### 5.3.5 Sharing

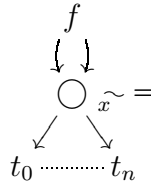
With the links, the names of the choice nodes are augmented with a link name and an entry name. In this way, some choice nodes which were shared in the skeleton are distinguished. In some cases, this distinction is not necessary. More sharing can be achieved by the entry names and by the link names.

#### Entry names

In order to share some entries which are in a sense equivalent, the entries the relations the links are based on are named. The case whenever the entries of a

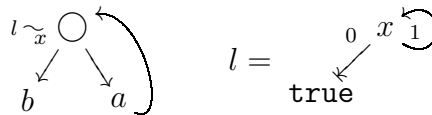
relation can be merged is discussed in detail in the section on the representation of relations. The most common case is the equality relation where all entries are equivalent. This sharing is necessary to represent some infinite relations.

**Example:** The link  $l$  is based on the identity relation, and the choice entry  $x$  is associated with the entries  $\{0, 1\}$ .



◇

**Example:** The link  $l$  is an infinite relation with all entries equivalent. This tree schema represents the set of finite words  $a^*b$ .



◇

### Link Names

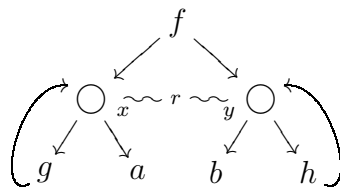
Considering two sets of paths of choice nodes linked by the same relations, and such that the two links binding these sets would have the same name, the problem is to decide whether we must combine the two sets or we can share them. Two sets of paths bound by the same link can be shared if they can be distinguished by the way the skeleton is covered. They can be distinguished if one of them appears after the other. In that case, it is easy to consider that each time a choice node associated with an entirely instantiated link is encountered, then this choice node is in the second set, and it is associated with a new fresh version of this link. It is even possible, following the same idea, to use a stack mechanism to distinguish between more intertwined sets of paths. The covering of the skeletons defines an order  $<_{\text{cover}}$  on the paths of the skeletons. The standard covering of the skeletons described in this chapter induces the alphabetic ordering.

Let  $S_1$  and  $S_2$  be two sets of paths, linked by the same relation  $r$ , such that the two links binding these sets would have the same name. We write  $T$  the tree schema obtained if we share the two sets.  $S_1$  is said to be before  $S_2$ , if

and only if,  $\forall e$  entry of  $r, \forall p_1 \in S_1$  with  $entry(T(p_1)) = e$ , and  $\forall p_1 \in S_1$  with  $entry(T(p_1)) = e, p_1 <_{cover} p_2$ . A set  $L$  of sets of paths bound by the same link does not have to be combined if it is totally ordered for the ordering “is before”.

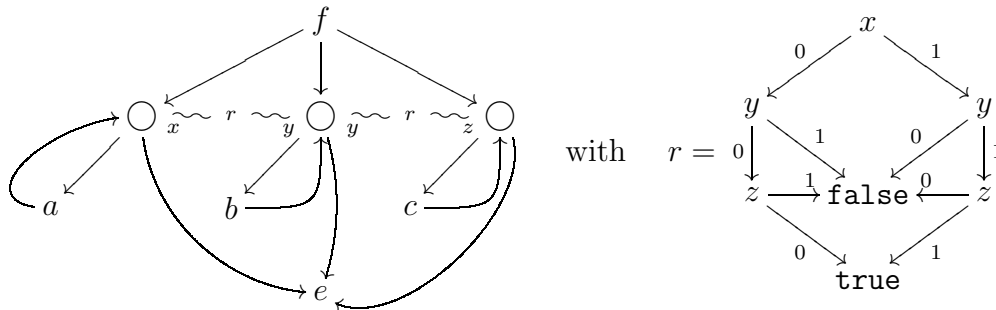
The problem with those two possible sharing is that they may be incompatible: a set of paths  $S_1$  can be before  $S_2$  if the entries of the relation are not shared, and  $S_1$  and  $S_2$  are not comparable after the sharing of some entries. In this case, the sharing of link names takes precedence.

**Examples:** An infinite tree schema that can take advantage of this sharing:



In this example, if  $r$  is the equality and if we share the equivalent entries, we must combine the links, and so the tree schema becomes infinite.

Using the same principles, we can easily represent the set  $f(a^ne, b^ne, c^ne)$ ,  $n \in \mathbb{N}$ :



Note that  $r$  is the equality relation. ◇

### 5.3.6 Set Represented by a Tree Schema

We give two definitions of the set represented by a tree schema. The first one is in line with what was presented so far and hopefully easier to understand than the second one, which is a constructive definition.

#### First Characterization

The main problem in characterizing the set represented by a tree schema is the reconstruction of the global relation defined by the links of the tree schema.

Let  $T$  be a tree schema, and  $l$  a link of  $T$ , based on the relation  $R$ . The link defines relations on any subset  $P$  of  $\text{cps}(T)$  such that there is an injective function  $\sigma$  from  $P$  to the set of entries of  $R$ , and  $\forall p \in P, T(p) = (\bigcirc, \text{name}_R(\sigma(p)), l)$ . These relations, denoted  $\text{GlobRel}(l, P)$  are defined by:

$$\forall x \in \bigotimes_{p \in P} \text{choice}(p), x \in \text{GlobRel}(l, P) \stackrel{\text{def}}{\Leftrightarrow} \exists y \in R, \forall p \in P, x_{(p)} = y_{(\sigma(p))}$$

The next step is to define the correct subsets of  $\text{cps}(T)$  that are bound by a given link, according to the rules of link sharing. The size of an entry name  $x$  of  $l$ ,  $|(l, x)|$  is defined as the number of entries of  $R$  named  $x$ . We write  $\text{least}(E, n, <)$  the  $n$  least elements of the totally ordered set  $E, <$ . If there is less than  $n$  elements in  $E$ , then  $\text{least}(E, n, <) = E$ . The ordered partition of  $E$ ,  $\text{OrdPart}(E, n, <)$  is the transfinite sequence defined by  $\text{OrdPart}(E, n, <) \stackrel{\text{def}}{=} \text{least}\left(E \setminus \bigcup_{j < i} \text{OrdPart}(E, n, <)_{j, n, <}\right)$ . The set of subsets of  $\text{cps}(T)$  that are bound by  $l$  is defined by:

$$\text{PathPart}(T, l) \stackrel{\text{def}}{=} \left\{ \bigcup_x \text{OrdPart}(\{p \mid T(p) = (\bigcirc, x, l)\}, |(l, x)|, <_{\text{cover}})_i \mid i \in \mathbb{N} \right\}$$

Now, we can define the relation associated with a tree schema  $T$ ,  $\text{rel}(T)$  as:

$$\text{rel}(T) \stackrel{\text{def}}{=} \bigwedge_{l \text{ link of } T} \bigwedge_{P \in \text{PathPart}(T, l)} \text{GlobRel}(l, P)$$

The set represented by a tree schema  $T$  can be seen as:

$$\text{Set}(T) = \bigcup_{c \in \text{rel}(T)} \{\langle \text{skel}(T) \rangle(c)\}$$

### Constructive Definition

This construction of the set represented by a tree schema illustrates the way an algorithm can work on tree schemata. A temporary global information is used to store the partial evaluations of the links. As a link may be shared, many different version of a given link can be partially evaluated. So the global information will be a function which associates with each link a list of partial evaluations of the relation it is based on. These lists will be denoted  $(R_1 : \dots : R_n)$ . The function that associates with every link of a tree schema  $T$  the empty list will be denoted  $T_\emptyset$ .

Let  $G$  be a temporary global information for a given covering of a tree schema  $T$ . Let  $l$  be a link of  $T$ ,  $R$  the relation of the link, and  $x$  an entry name of  $l$ . Let  $G(l) = (R_1 : \dots : R_n)$ . The set of possible choices for  $l$  on entry name  $x$  according to  $G$  is written  $G(l)_{(x)}$ . By definition, if no relation of  $G(l)$  contains

an entry named  $x$  in its domain, then  $G(l)_{(x)} \stackrel{\text{def}}{=} R_{(e)}$  where  $e$  is an entry named  $x$ , and else if  $R_i$  is the first relation of  $G(l)$  containing an entry  $e$  named  $x$  in its domain,  $G(l)_{(x)} \stackrel{\text{def}}{=} R_{i(e)}$ . The new global information after the choice  $v$  for the entry name  $x$  of the link  $l$  is written  $G_{x=v}^l$ . Following the same cases as in the definition of  $G(l)_{(x)}$ , if no relation of  $G(l)$  contains an entry named  $x$  in its domain, then  $G_{x=v}^l(l) \stackrel{\text{def}}{=} (R_1 : \dots : R_n : R_{i:e=v})$  where  $e$  is the first entry of  $R$  named  $x$ , and if  $R_i$  is the first relation of  $G(l)$  containing an entry named  $x$  in its domain,  $G_{x=v}^l(l) \stackrel{\text{def}}{=} (R_1 : \dots : R_{i-1} : R_{i:e=v} : R_{i+1} : \dots : R_n)$  where  $e$  is the first entry of  $R_i$  named  $x$ . In any case,  $\forall l' \neq l, G_{x=v}^l(l') \stackrel{\text{def}}{=} G(l')$ .

In order to define the behavior of relations on infinite instantiations, we have to define a progression order on the global informations. A global information  $G_1$  comes after a global information  $G_2$  on the same tree schema  $T$ , written  $G_2 \xrightarrow{T} G_1$  if and only if for all  $l$  link of  $T$ , let  $G_1(l) = (R_1^1 : \dots : R_n^1)$  and  $G_2(l) = (R_1^2 : \dots : R_m^2)$ , then  $m \leq n$  and  $\forall i \leq m, l_i^1$  is a positive instantiation of  $l_i^2$ . A relation  $R$  is a positive instantiation of the relation  $S$  if and only if there is a set of entries  $I$  of  $S$  and an element  $v \in S$  such that  $R = S_{:I=v(I)}$ .

The function translating the tree schemata into sets of trees,  $Set$ , is defined by:

$$Set(T) = \left\{ t \in \mathcal{H}(F) \mid \exists G, (t, G) \in \widetilde{Set}(T, T_\emptyset) \right\}$$

and  $\forall f \in F$ ,

$$\begin{aligned} \widetilde{Set} \left( \begin{array}{c} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{array}, G_0 \right) &\stackrel{\text{def}}{=} \left\{ \left( \begin{array}{c} f \\ \swarrow \searrow \\ u_0 \dots u_{n-1} \end{array}, G_n \right) \mid \forall i < n, (u_i, G_{i+1}) \in \widetilde{Set}(t_i, G_i) \right\} \\ \widetilde{Set} \left( \begin{array}{c} \circ \\ \swarrow \searrow \\ t_0 \dots t_n \end{array} \overset{\sim}{x} l, G \right) &\stackrel{\text{def}}{=} \bigcup_{i \in G(l)_{(x)}} \left\{ (u, H) \mid (u, H) \in \widetilde{Set}(t_i, G_{x=i}^l) \wedge G_{x=i}^l \xrightarrow{T} H \right\} \end{aligned}$$

The function  $\widetilde{Set}$  is defined as usual as the greatest fixpoint of this fixpoint equation. The ordering is the pointwise ordering of inclusion of the images. It is easy to see that this fixpoint equation is monotone for this ordering.

Obviously, any element of  $Set(T)$  is an element of  $Ens(skel(T))$ . Whatever the finite tree in  $Set(T)$ , it is associated with a unique global information by  $\widetilde{Set}(T)$ . In the case of infinite trees, there is at least one global information associated that contains no empty relation.

**Example:** Let  $T$  be  $\left( \begin{array}{c} \overset{l \sim x}{\circ} \\ \swarrow \searrow \\ b \quad a \end{array} \right)$  and  $l = \left( \begin{array}{c} 0 \quad x \\ \swarrow \searrow \\ \text{true} \quad 1 \end{array} \right)$ . We define  $\mathcal{G}_T$  the set of all possible global informations on  $T$ .  $G_n$  is the relation  $l$  is based on, partially evaluated on its  $n-1$  first entries, with value 1.  $G_n^0$  is the same relation partially



evaluated on its  $n^{\text{th}}$  entry with value 0. Then:

$$\widetilde{Set}(T, G_n) = \{(b, G_n^0)\} \cup \left\{ \left( \begin{array}{c} a \\ \downarrow \\ u \end{array}, H \right) \mid (u, H) \in \widetilde{Set}(T, G_{n+1}) \wedge G_{n+1} \xrightarrow{T} H \right\}$$

We show the part of the iteration sequence [CC79] of  $\widetilde{Set}$  on  $T$ :

$$\begin{aligned} \widetilde{Set}_0(T, T_\emptyset) &= \mathcal{H}(F) \times \mathcal{G}_T \\ \widetilde{Set}_1(T, T_\emptyset) &= \{(b, G_0^0)\} \cup \left\{ \left( \begin{array}{c} a \\ \downarrow \\ u \end{array}, H \right) \mid u \in \mathcal{H}(F) \wedge G_1 \xrightarrow{T} H \right\} \\ \widetilde{Set}_2(T, T_\emptyset) &= \left\{ (b, G_0^0), \left( \begin{array}{c} a \\ \downarrow \\ b \end{array}, G_1^0 \right) \right\} \cup \left\{ \left( \begin{array}{c} a \\ \downarrow \\ a \\ \downarrow \\ u \end{array}, H \right) \mid u \in \mathcal{H}(F) \wedge G_2 \xrightarrow{T} H \right\} \\ &\vdots \\ \widetilde{Set}_i(T, T_\emptyset) &= \bigcup_{n < i} \{(a^n b, G_n^0)\} \cup \left\{ (a^i u, H) \mid u \in \mathcal{H}(F) \wedge G_i \xrightarrow{T} H \right\} \\ &\vdots \\ \widetilde{Set}_\omega(T, T_\emptyset) &= \bigcap_{i < \omega} \widetilde{Set}_i(T, T_\emptyset) \end{aligned}$$

Thus  $\widetilde{Set}(T, T_\emptyset)$  contains every couple  $\{(a^n b, G_n^0)\}$ . If it contained a couple  $(a^\omega, H)$ , then  $\forall i, G_i \xrightarrow{T} H$ . This means that  $H$  would contain a version of the relation of  $l$  entirely instantiated on  $1^\omega$ , which is impossible, because it does not belong to the relation.  $\diamond$

### 5.3.7 Restrictions on the Links

If a skeleton could be associated with any relation, then the tree schemata would be able to represent any set of trees. The problem is that tree schemata must be kept finite, so we must restrict the links that can be associated with a particular skeleton.

#### Finiteness

In order to have finite tree schemata, we require that a tree schema is a regular tree, and that each link is based on a regular relation. For a practical use, we will actually restrict the relations even more, to  $\omega$ -deterministic relations.

#### Only Choice Nodes that Depend on Another One are Linked

Finding out whether a given relation can be independently decomposed is a hard problem, which we don't solve efficiently in this thesis. There is an easy case,

though, where we know the choice node  $(\bigcirc, x, l)$  does not depend on the other choice nodes of the link  $l$ : it is the case if and only if the entry does not appear in the decision diagram representing the relation of  $l$  (it happens because of redundant nodes elimination). Thus we require that for any choice node  $(\bigcirc, x, l)$ , either  $l$  is the **true** relation (and we denote the choice node by just  $\bigcirc$ ), or the entry name  $x$  appears in the decision diagram of the relation of  $l$ .

### Keeping the Skeleton a Good Approximation

The skeleton is supposed to be a good upper approximation of the set represented by the tree schema. If the relation associated with the skeleton is too restrictive, then it may be better to build the tree schema on a smaller skeleton. In order to avoid some redundancy, we forbid some of these relations. The problem is that for some sets of trees, there is no best skeleton approximating the set. The only property we can enforce on tree schemata is:

**Property 5.1** *For any tree schema  $T$  such that  $Set(T)$  admits a best skeleton approximation,  $skel(T)$  is this best approximation.*

To enforce this property (even on independent subtree schemata), we restrict the possible links. First, every relation on which a link is based must be *full*, that is, for all entries and all possible value of the entry, there is a vector of the relation with this value on this entry. The formal definition is given bellow. Second, we forbid any link between a choice node and one of its subtree except in two cases. The first case allows to link a choice node and one of its subtrees to another node, provided there is no real relation between the choice node and its subtree. This case allows the elimination of some infinite trees from the set represented by the skeleton, as in this case there is no real prefix relation. The second case allows the representation of some sets with no best skeleton approximations.

**Definition:** Let  $R$  be a relation on  $\bigotimes_{i \in I} [n_i]$ .  $R$  is *full* if and only if  $\bigotimes_{i \in I} [n_i] = Support(R)$ .

We first prove that with such restriction on the links, assuming there is no exception, the skeleton of the tree schema is the best approximation of the set represented by the tree schema. We will show later that the two cases where we can have a link between a node and its subtree don't make the skeleton greater than the best possible if there is a best one.

**Proof:** Let  $T$  be a tree schema such that each link of  $T$  is based on a full relation and there is no link between a choice node and one of its subtrees. Suppose there is a skeleton  $S$  such that  $Set(T) \subset Ens(S) \subset Ens(skel(T))$ . If  $S \neq skel(T)$ , then there is a path  $p$  such that  $S_{[p]}$  and  $skel(T)_{[p]}$  are choice nodes and there is a value  $i$  of  $skel(T)_{[p]}$  that is not a value of  $S_{[p]}$ .  $T_{[p]}$  is labeled by  $(\bigcirc, x, l)$ , and  $l$

is based on  $R$ , which is full. It means that there is a  $v \in R$  such that  $v_{(e)} = i$  ( $e$  is an entry of name  $x$ ). As there is no link between parent nodes, the fact that a value  $c \in rel(T)$  validates the path  $p$  ( $\forall q \in cps(T), q \prec p \Rightarrow qc_{(q)} \prec p$ ) does not depend on the projection of  $c$  on the entries of the link. Thus there is a  $c$  in  $rel(T)$  such that the projection of  $c$  on the entries of  $l$  is  $v$ , and so the choice  $i$  is taken at  $p$ . This means that  $Set(T) \not\subseteq Ens(S)$ .  $\square$

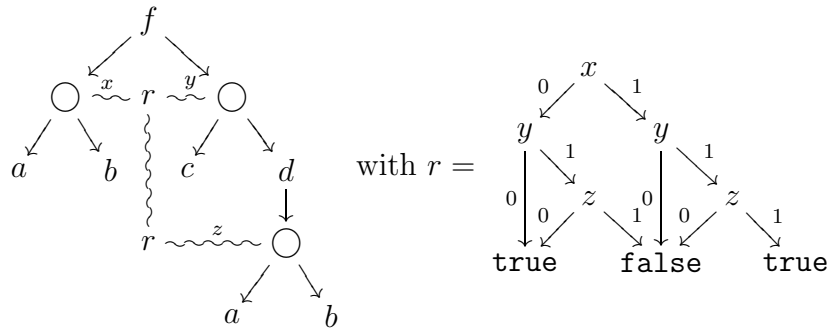
**First Case: Links with no Real Relation between Parent Nodes** Let  $N_1$  be a node labeled by  $(\bigcirc, x_1, l)$  and  $N_2$  a node labeled by  $(\bigcirc, x_2, l)$ <sup>7</sup> such that there is a path from  $N_1$  to  $N_2$ .  $l$  is based on the relation  $R$ . Let  $B = \{b \in R_{(x_1)} \mid \exists p, N_1.bp = N_2\}$  be the set of output values of the choice node  $N_1$  that can lead to  $N_2$ . Then the link does not impose any real relation between  $N_1$  and  $N_2$  (and so is valid) if:

$$\forall b \in B, \forall i \in R_{(x_2)}, \exists v \in R \text{ such that } v_{(x_1)} = b \text{ and } v_{(x_2)} = i$$

Tree schemata with links between first case parent nodes still represent sets with best skeleton approximation the skeletons of the tree schemata<sup>8</sup>.

**Proof:** We start again with the same objects as in the proof for the general case. The difference with the general case is that there may be a link between  $T_{[p]}$  and a parent node. But if there is a  $q \prec p$  such that  $T_{[q]}$  is linked to  $T_{[p]}$ , whatever  $j$  such that  $qj \preceq p$ , there is a  $v$  such that  $v_{(e)} = i$  and the projection of  $v$  on the entry of  $T_{[q]}$  is  $j$ . So a  $c$  in  $rel(T)$  such that its projection on the entries of the link is  $v$  exists.  $\square$

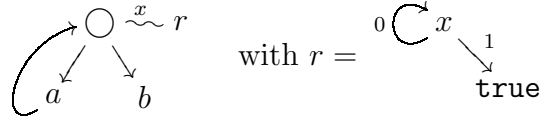
**Examples:** This tree schema with a link between parent nodes is valid.



<sup>7</sup>It is possible that  $x_1 = x_2$  or even  $N_1 = N_2$ .

<sup>8</sup>We must be careful, though, in the presence of infinite relations, as the first case does not impose anything on the infinite behavior of the relation and the skeleton of the tree schema could be greater than the best skeleton if we don't respect the rules of infinite behaviors of tree schemata.

The set represented by the tree schema is  $\left\{ \begin{array}{ccc} f & f & f \\ \swarrow \searrow & \swarrow \searrow & \swarrow \searrow \\ a & c & d, b \\ & & \downarrow \\ & & a \\ & & \downarrow \\ & & b \end{array} \right\}$  The tree schema representing the set of finite trees of the form  $a^*b$  also falls in this case:

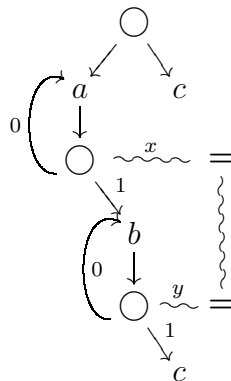


◇

**Second Case: Sets with no Best Skeleton Approximation** In the case when the set represented by the tree schema does not have any best skeleton approximation, we cannot refine the links until as much as possible is represented by the skeleton. Thus we allow any link between two parent nodes if with this link, the sets they represent does not have any best skeleton approximation.

In practice, it is a difficult problem to characterize tree schemata with best skeleton approximation and links with real relations between parent nodes (although we know that such tree schemata are equivalent to tree schemata without such links, and as such are forbidden). Thus, for practical use, we can restrict the second case to allow only a subclass of tree schemata representing sets with no best skeleton approximations which is easier to identify. For example, there is no best skeleton for  $Set(T)$  if there is a link  $l$ , a node  $N$  labeled by  $(\bigcirc, x, l)$  reachable, a path  $p$  that can be taken and such that  $N.p = N$  without going through another node linked to  $l$  (the independent loop), and another node  $M$  labeled by  $(\bigcirc, y, l)$  reachable from  $N$  whatever the number of loops and such that each choice in  $N$  leading to  $p$  allows a choice of  $M$  leading to a path coming back to  $M$ .

**Example:** The following tree schema is valid, because there is no best skeleton for this set of trees. We can even identify an independent loop.

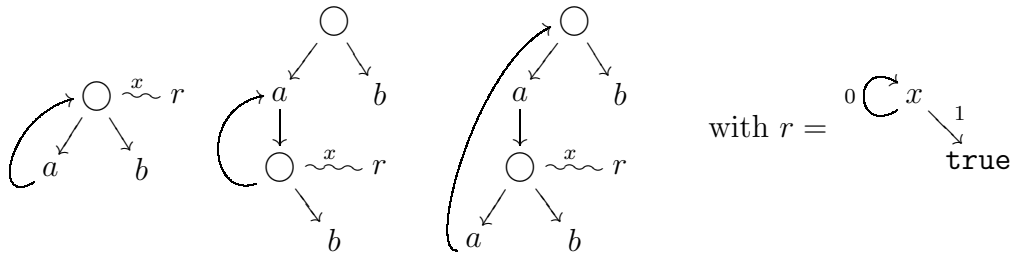


The set of trees represented by this tree schema is  $\{a^n b^n c \mid n \in \mathbb{N}\}$ . ◇

**Infinite Behavior**

Because skeletons approximate sets of possibly infinite trees from above, their infinite behavior is a greatest fixpoint. In order to discard some infinite trees from those sets, we use links. The problem is that these links will relate some nodes and their subtrees, and thus the structure of the skeleton (or even the tree schema) may interfere with them.

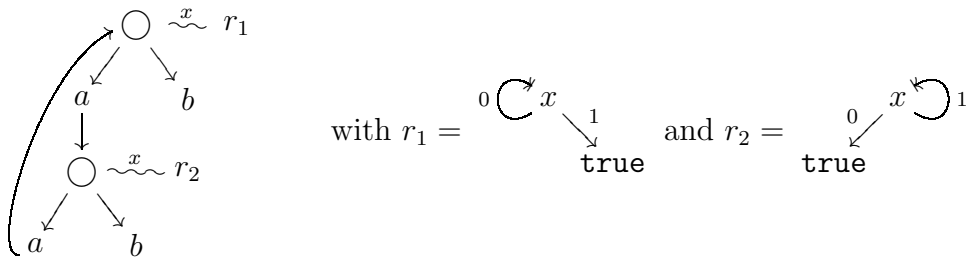
**Example:** The following three tree schemata are equivalent:



◇

It can even be worse, as if we don't link every node in a given cycle, it is possible that the infinite behaviors of some nodes are not compatible.

**Example:**



◇

In order to avoid this case, and more generally to have a uniform representation (and in particular no strictly monotone sequence of tree schemata representing the same set), we just have to link all these nodes. In fact, it is the opposite of the general policy with links: instead of decomposing independent parts as much as possible, we try to link independent nodes.

These cases are easily characterized: we must try to incorporate the current choice node if it is either in a cycle with another choice node that is linked or if one of the children of the choice nodes is a cycle with another choice node that is linked. The latter case is easily handled by the dictionary of keys of cycles. In the former case, we merely don't allow the relation between the different nodes of a given cycle (which may be the `true` relation) to be decomposed in independent relations.

## 5.4 The Set of Tree Schemata, $\mathcal{TS}(F)$

### 5.4.1 Definition

A tree schema  $T$  on  $F$  is a regular tree on  $F \cup \{(\bigcirc, x, l) \mid l \text{ is a link, and } x \text{ is an name entry of the relation of } l\}$  such that:

- a link  $l$  is a regular relation  $R$  and a map  $f : \text{ename}(R) \rightarrow \wp(\text{Subtrees}(T))$  such that  $\forall x \in \text{ename}(R)$ ,
  - $f(x)$  is the set of subtrees of  $T$  labeled by  $(\bigcirc, x, l)$ ;
  - $f(x)$  is not empty;
  - $x$  appears in the decision diagram of  $R$ , if this decision diagram is not `true`.
- if  $t$  is a subtree of  $T$  labeled by  $\begin{array}{c} \bigcirc \overset{x}{\sim} R \\ \swarrow \quad \searrow \\ t_0 \quad \dots \quad t_{n-1} \end{array}$  then:
  - $n = 0$  and  $t = T$ , or  $n > 1$  and the  $t_i$  are labeled by elements of  $F$  and ordered according to these elements (that is  $\text{skel}(T)$  is a skeleton);
  - $R_{(x)} = [n]$  (that is  $R$  is full);
- Whatever the cycle in  $T$ , every choice node in the cycle is associated with the same link.
- for all subtree  $S$  of  $T$  that is a tree schema, either  $\text{Set}(S)$  does not have any best skeleton approximation or for any  $u$  and  $t$  subtrees of  $S$  labeled by  $(\bigcirc, x, l)$  and  $(\bigcirc, y, l)$ , if there is a path from  $u$  to  $t$ , the choice on  $u$  does not restrict the choice on  $t$  (see previous section).

The set of all such tree schemata on  $F$  is denoted  $\mathcal{TS}(F)$ .

### 5.4.2 Description of the Sets of Trees Representable by Tree Schemata

In order to give a formal definition of the sets of trees representable by tree schemata and a flavor of the expressive power (and limitations) of tree schemata, we use formal language transformers as defined in [CC95]. In order to have the necessary expressive power, in particular concerning the infinite behavior, we enrich the formal language transformers with regular relations. Then, we restrict the set of formal language transformers with regular relations so that their semantics is exactly the set of tree schemata.

#### Formal Language Transformers

First, let us recall the notations of [CC95]. A non-ground term is a tree<sup>9</sup> on  $F \cup v$ , where  $v$  is a set of term variables. The semantics of a non-ground term  $T$  is the map  $\llbracket T \rrbracket \in (v \rightarrow \mathcal{H}(F)) \rightarrow \mathcal{H}(F)$  defined as the function mapping any function  $\kappa : v \rightarrow \mathcal{H}(F)$  to the tree substitution of every label  $x \in v$  of  $T$  by  $\kappa(x)$ . Because we deal with infinite trees, we need a little restriction on some terms: we say that a term  $T'$  is of *finite non groundness* if the set of paths  $p$  such that  $T'(p)$  is a variable is finite.

A meta expression  $e$  denotes a language transformer. Meta expressions can be defined as:

$$e ::= \mathcal{X} \quad | \quad \{T' : T_1 \in e_1, \dots, T_n \in e_n\} \quad | \quad e_1 \cup e_2$$

where  $T'$  is a term of finite groundness, the  $T_i$  are terms, and  $\mathcal{X}$  is a set variable in  $\mathcal{V}$ . These set variables are free in meta expressions, but term variables are local to meta expressions of the form  $\{T' : T_1 \in e_1, \dots, T_n \in e_n\}$  which are used to describe intersection ( $\{x : x \in e_1, x \in e_2\}$ ), projection  $\left( \left\{ x : \begin{array}{c} f \\ \swarrow \searrow \\ x \quad y \end{array} \in e \right\} \right)$ , sets denoted by terms of finite groundness ( $\{T'\}$ ) and many others. The semantics of a meta expression  $e$  is the map  $\llbracket e \rrbracket \in (\mathcal{V} \rightarrow \wp(\mathcal{H}(F))) \rightarrow \wp(\mathcal{H}(F))$  inductively defined by:

$$\begin{aligned} \llbracket \mathcal{X} \rrbracket \rho &\stackrel{\text{def}}{=} \rho(\mathcal{X}) \\ \llbracket \{T' : T_1 \in e_1, \dots, T_n \in e_n\} \rrbracket \rho &\stackrel{\text{def}}{=} \left\{ \llbracket T' \rrbracket \kappa \mid \kappa \in v \rightarrow \mathcal{H}(F) \bigwedge_{1 \leq i \leq n} \llbracket T_i \rrbracket \kappa \in \llbracket e_i \rrbracket \rho \right\} \\ \llbracket e_1 \cup e_2 \rrbracket \rho &\stackrel{\text{def}}{=} \llbracket e_1 \rrbracket \rho \cup \llbracket e_2 \rrbracket \rho \end{aligned}$$

With this definition,  $\llbracket e \rrbracket$  is monotonic. So it can be used to describe sets of languages by means of systems of equations of the form  $\mathcal{X} = e_{\mathcal{X}}$ . The semantics of such systems of equations would be the greatest fixpoint of  $\rho(\mathcal{X}) = \llbracket e_{\mathcal{X}} \rrbracket \rho$ .

<sup>9</sup>We extend the languages of [CC95] to infinite trees. Thus a term can be any regular tree.

We will show in the next chapter that if  $\rho$  maps every set variable to a set representable by a tree schema, then  $\rho'$  defined as  $\rho'(\mathcal{X}) = \{\{e_{\mathcal{X}}\}\}$   $\rho$  maps every set variable to a tree schema too.

### Formal Language Transformers with Relations

Tree schemata can be transformed by even more powerful operations, as we can add relations to the system of equations. These relations can be described by boolean formulae  $f$  over guards of the form  $\mathcal{X} = a$  where  $a \in F$ .

$$f ::= (\mathcal{X} = a) : f \mid \text{true} \mid f_1 \vee f_2 \mid \text{iter}(f) \mid \mathbf{R}$$

$\mathbf{R}$  is a relation variable in  $\mathbf{V}$ . The semantics of a formula  $f$  is the map  $\llbracket f \rrbracket : (\mathbf{V} \rightarrow \mathcal{R}el) \rightarrow \mathcal{R}el$  defined inductively by:

$$\begin{aligned} \llbracket (\mathcal{X} = a) : f \rrbracket \tau &\stackrel{\text{def}}{=} \{a\alpha \mid \alpha \in \llbracket f \rrbracket \tau \text{ and the name of the first entry is } \mathcal{X}\} \\ \llbracket \text{true} \rrbracket \tau &\stackrel{\text{def}}{=} \bigotimes_{i \in \mathbb{N}^*} F \\ \llbracket f_1 \vee f_2 \rrbracket \tau &\stackrel{\text{def}}{=} \llbracket f_1 \rrbracket \tau \vee \llbracket f_2 \rrbracket \tau \\ \llbracket \text{iter}(f) \rrbracket \tau &\stackrel{\text{def}}{=} \Omega(\llbracket f \rrbracket \tau) \\ \llbracket \mathbf{R} \rrbracket \tau &\stackrel{\text{def}}{=} \tau(\mathbf{R}) \end{aligned}$$

With this semantics, formulae are monotonic, except for the `iter` definition. It means that we can define the relation imposed on the system of equations as one of the relation variable, whose value is defined by a system of relation equations of the form  $\mathbf{R} = f_{\mathbf{R}}$  such that for every  $\mathbf{R}$  that appears inside an `iter` definition in any  $f_{\mathbf{S}}$ , there is no `iter` in  $f_{\mathbf{R}}$ . The semantics of this system of relation equations is the least fixpoint of  $\tau(\mathbf{R}) = \llbracket f_{\mathbf{R}} \rrbracket \tau$

The semantics of a set of formal languages equations  $(\mathcal{X} = e_{\mathcal{X}})$  constrained by a relation  $(\mathbf{R})$  is the greatest fixpoint over the set  $\mathcal{V} \times \text{AllProj}(\mathbf{R}) \rightarrow \wp(\mathcal{H}(F) \times \text{AllProj}(\mathbf{R}))$  of the set of equations:

$$\varrho(\mathcal{X}, S) = \{(t, Q :_{\mathcal{X}=t(\varepsilon)}) \mid (t, Q) \in \{\{e_{\mathcal{X}}\}\}_{\varrho_S}\}$$

In order to give sense to these equations, we must extend the semantics of meta expressions to deal with relations:

$$\begin{aligned} \{\{\mathcal{X}\}\}_{\varrho_S} &\stackrel{\text{def}}{=} \varrho(\mathcal{X}, S) \\ \{\{T' : T_1 \in e_1, \dots, T_n \in e_n\}\}_{\varrho_{S_0}} &\stackrel{\text{def}}{=} \left\{ (\llbracket T' \rrbracket \kappa, S_n) \mid \bigwedge_{1 \leq i \leq n} (\llbracket T_i \rrbracket \kappa, \varrho_{S_i}) \in \{\{e_i\}\}_{\varrho_{S_{i-1}}} \right\} \\ \{\{e_1 \cup e_2\}\}_{\varrho_S} &\stackrel{\text{def}}{=} \{\{e_1\}\}_{\varrho_S} \cup \{\{e_2\}\}_{\varrho_S} \end{aligned}$$

The set associated with  $\mathcal{X}$  by  $\varrho$  is  $\{t \in \mathcal{H}(F) \mid (t, S) \in \varrho(\mathcal{X}, \mathbf{R}) \text{ and } S \neq \emptyset\}$ .



### Restrictions on Meta Expressions

It is easy to see that any set represented by a tree schema is a solution of such system of equations constrained by a relation. Moreover, meta expressions with relations applied to tree schemata can be represented by tree schemata, and the set of all trees, which is the starting point of the iteration sequence leading to the greatest fixpoint can be represented by a tree schema. The problem is that the set of tree schemata is not closed under infinite union or intersection. Thus, for some meta expressions, the greatest fixpoint cannot be represented by a tree schema.

The minimal generating meta expressions are easy to extract from the very structure of the tree schemata: we keep the union, for the choice node, the set variables, for the loops, and expressions of the form  $\left\{ \underset{x_0 \dots x_{n-1}}{\overset{f}{\downarrow}} : (x_i \in e_i)_{i < n} \right\}$ , for the root construction of sets. With these meta expressions and the regular relations, we can define any tree schema as representing a greatest fixpoint of a meta expression equation as defined above.

## 5.5 Discussion

The representation presented in this chapter can be viewed as a special case or an extension of the notion of automaton. But it really brings new elements in the representation of sets of trees. The first element is the strong expressive power of the representation, which is a concern in abstract interpretation. The second element is a strong notion of determinism with good consequences on the algorithmic of the representation. The third element is an efficiency concern in the design of the representation which led to the sharing of many elements of the set of trees.

### 5.5.1 Expressive Power

Skeletons are related to deterministic top-down finite tree automata. If we remove the infinite trees from their interpretation, then they are as expressive as deterministic top-down tree automata. If we add finite relations and still consider finite trees, we get non-deterministic tree automata. With infinite relations, we encompass tree automata with equality constraints between brothers.

Although Tree schemata can represent most sets of trees representable by tree automata of many kind, not every useful set of trees is representable by a tree schema, of course.

**Example:** Because of the regularity of relations in the links, the following set of trees cannot be represented by a tree schema:

$$\left\{ \begin{array}{c} \textcircled{C}^f \\ \searrow \\ t \end{array} \middle| t \in \mathcal{H}(F) \right\}$$

This set corresponds to the meta-expression semantics of a term of infinite non-groundness:  $\left\{ \begin{array}{c} \textcircled{C}^f \\ \searrow \\ x \end{array} \right\}$ . Note the difference with the following set of trees which can be represented by a tree schema:

$$\mathcal{X} = \left\{ \begin{array}{c} f \\ \swarrow \quad \searrow \\ x \quad y \end{array} \middle| x \in \mathcal{X} \text{ and } y \in \mathcal{H}(F) \right\} \text{ is represented by } \begin{array}{c} \textcircled{C}^f \\ \searrow \\ \mathbf{1}_F \end{array}$$

◇

### 5.5.2 Strong Determinism

The empty set admits a best skeleton representation, and so any tree schema representing the empty set is based on this skeleton. But as the skeleton of a tree schema represents a supset of the set represented by the tree schema, testing emptiness is just testing whether the tree schema is labeled by some  $(\textcircled{O}, x, l)$  of arity 0. So emptiness is decidable in constant time.

This good algorithmic result is a consequence of a strong notion of determinism in the procedure deciding membership in tree schemata. This procedure is deterministic in the usual sense, that we don't have to backtrack on a decision in the process. Moreover, in the course of the decision process, we know at every point that there is a tree in the set represented by the tree schema that would lead the decision process to that point.

### 5.5.3 Sources of Non-Uniqueness

Tree schemata use the algorithms to maintain a kind of minimalization properties that come from the representation of regular trees. Yet, sets of trees may be non-uniquely represented by tree schemata. This is a bad point for tree schemata, as because of this non-uniqueness, the representation does not get the full benefit from the minimalization. Mostly, non-uniqueness stems from the links. These sources of non-uniqueness have not been treated in this presentation of tree schemata, because the cost to treat them was deemed to high. In some cases, though, we can present heuristics to limit non-uniqueness.

### Decomposition in Independent Relations

Allowing decomposition of the links in independent relations is one of the big algorithmic issues of tree schemata. This decomposition may come naturally, for free, when building the tree schemata. But it does not mean that we get the most decomposed relations, which would lead to more efficient algorithms. The problem is that, except in some rare cases, deciding whether a given relation can be decomposed is difficult.

### Entry Names

Entry names are the only place where we introduced variables. They raise two problems: the choice of the names, as only the fact that they are different matters, and finding out equivalent entries.

Concerning the choice of the names, we can give two possible heuristics. The first one consists in giving to a set of equivalent entries the index of the first entry as the entry name for these entries. Note that there is no problem of interference between variable names of different relations. The problem of this choice is that we must then maintain this property when modifying the relation. In particular, we need the knowledge of which entry comes first. The second heuristic consists in giving to the entries the node to which they are related in the tree schema. Because for some equivalent entries, the nodes will be different, this will differentiate some equivalent entries. If two nodes that are equal are associated with a relation through non equivalent entries, we can use the first heuristic.

Concerning the discovery of the equivalence sets of entries, the problem is quite similar to the decomposition in independent relations. For the most common relation, equality, we know that the different entries are equivalent, but the problem is to hard in general.

### Never Instantiated Relations

When defining the possible links imposed on a tree schemata, we were mainly concerned by the best skeleton approximation and the necessity that links don't restrict too much the set of trees. But sometimes, the links simply don't add (or rather remove) anything, or could be smaller and have the same effect. Finding out whether it is the case might be too costly, but we can come with this knowledge when performing some necessary algorithms. So it can be useful to identify those cases.

The first case is when a given entry in a relation is never reached. Although we always check that each entry name is associated with one node at least, it is more difficult to check that the number of different paths leading to nodes with that entry name is the same as the number of entries with that name. The main

reason is that a given node may be accessed from more than one path. Note that it is a non local information.

The second case involves the relation between two nodes that cannot be both visited during the decision procedure of a given tree. This cannot concern parent nodes. But it is the case if every closest common ancestor of the two nodes is a choice node. This property is not always checked, because it is not local.

# Chapter 6

## Algorithms on Tree Schemata

In this chapter, we show some algorithms on tree schemata that may be useful in abstract interpretation. These algorithms prove constructively some properties of the tree schemata, such as the decidability of the inclusion testing. They may also serve as guidelines for the development of other algorithms on tree schemata.

The essence of the representation by tree schemata is the decomposition of the set of trees in a tree structure (the skeleton) and relations (the links). Algorithms manipulating relations have already been presented in chapter 4. In order to introduce progressively the different concepts and difficulties of the algorithms, we first present the algorithms on skeletons, and then the definitive algorithms combining the difficulties of both skeletons and relations.

### 6.1 Auxiliary Informations on the Links

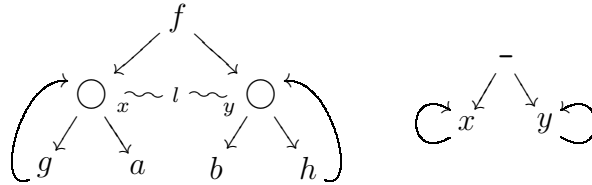
Links are local in the way they are accessed and shared, but they require a global information structure when building constructively the set represented by a tree schema (page 99). When we are just interested by the finite behaviors of the links, we can use these global informations when going through the tree schemata to compare different links with different entry names and even different sets of entries. But sometime, we may need a more global information on the links to perform operations on the entire relations. In this case, it is useful to know the scope of the links, and to be able to compare the entries of two links on different tree schemata. Such informations can be computed, but if they are needed often enough, they should be kept once for all while building the tree schemata. That is, if the information can be made local and does not add any new source of non-uniqueness.

The scope of the link can be defined as follows: when going through the tree schema, the scope of a link  $l$  starts at the first choice node linked to  $l$  we can encounter and stops at the last choice node linked to  $l$  that we can encounter. This intuitive definition has two drawbacks: it depends on the particular way of

going through the tree schema (the covering ordering), and if we follow the loops indefinitely, there is no reason why we should have a “last” choice node. But we can give another notion of scope that is still local and does not depend on the particular covering of the trees: the *scope* of a link is the smallest (for  $\leq$ ) subtree of the tree schema that contains every choice node linked to  $l$ .

The scope of a link is indeed a good place (or key to the place) to store information about the link: we know that the link does not have to be taken into account until we first encounter the scope, and we don’t have to care anymore once every subtree of the scope have been treated. Moreover, as soon as we don’t modify any subtree of the scope, the scope does not change, so the information is still local and can be shared, without maintenance. In order to be able to compare links coming from different tree schemata, we need to make a correspondence between the different entries of the links, that is, we need to now the entry index as a path in the tree schema. This information can be stored relative to the scope by a tree, whose root corresponds to the scope, and which is labeled by entry names plus a new label,  $-$ .

**Example:** The following schema is the scope of  $l$ . The second tree is the entry tree associated with  $l$ :



◇

Another useful information that we can store is whether the link corresponds to a best skeleton or not. This information can allow the use of optimizations based on the fact that the skeleton are best upper approximations.

## 6.2 Inclusion

Inclusion testing is a basic operation that is performed in the iteration sequences, at each iteration, to see if we have reached a fixpoint or a post fixpoint.

### 6.2.1 Inclusion of Skeletons

The idea of the algorithms deciding whether a skeleton  $T$  represents a subset of the skeleton  $U$  is to find a  $t \in Ens(T)$  that is not in  $Ens(U)$ . It is performed by going through  $T$  and  $U$  at the same time. We don’t need to enumerate all the trees in  $Ens(T)$  because of the regularity properties of  $T$  and  $U$ .

inclusionSkel( $T, U$ ):

```

if  $T = \bigcirc$  then return(true)
if  $U = \bigcirc$  then return(false)
return(inclusionSkelRec( $T, U, \emptyset$ ))

```

inclusionSkelRec( $T, U, A$ ):

```

if  $(T, U) \in A$  then return(true)
 $A' \leftarrow A \cup \{(T, U)\}$ 

if  $T = \begin{array}{c} \bigcirc \\ \swarrow \searrow \\ T_0 \dots T_n \end{array}$  then

  if  $U = \begin{array}{c} f \\ \swarrow \searrow \\ U_0 \dots U_{m-1} \end{array}$  then return(false)

   $U$  is  $\begin{array}{c} \bigcirc \\ \swarrow \searrow \\ U_0 \dots U_m \end{array}$ 

  for  $i \leftarrow 0$  to  $n$  do
  |
  |  $T_i$  is  $\begin{array}{c} f_i \\ \swarrow \searrow \\ T_{i0} \dots T_{ip-1} \end{array}$ 
  |
  | if  $\exists j$  such that  $U_j$  is  $\begin{array}{c} f_i \\ \swarrow \searrow \\ U_{j0} \dots U_{jp-1} \end{array}$  then
  |
  |    $A' \leftarrow A' \cup \{(T_i, U_j)\}$ 
  |   for  $k \leftarrow 0$  to  $p-1$  do
  |     if inclusionSkelRec( $T_{ik}, U_{jk}, A'$ ) = false then return(false)
  |   else return(false)
  |
  | return(true)
  |
else  $T$  is  $\begin{array}{c} f \\ \swarrow \searrow \\ T_0 \dots T_{n-1} \end{array}$ 

  if  $U = \begin{array}{c} g \\ \swarrow \searrow \\ U_0 \dots U_{m-1} \end{array}$  then

    if  $g \neq f$  then return(false)
    for  $i \leftarrow 0$  to  $n-1$  do
      if inclusionSkelRec( $T_i, U_i, A'$ ) = false then return(false)
    return(true)

   $U$  is  $\begin{array}{c} \bigcirc \\ \swarrow \searrow \\ U_0 \dots U_m \end{array}$ 

  if  $\exists U_i$  such that  $U_i = \begin{array}{c} f \\ \swarrow \searrow \\ V_0 \dots V_{n-1} \end{array}$  then

     $A' \leftarrow A' \cup \{(T, U_i)\}$ 
    for  $j \leftarrow 0$  to  $n-1$  do
      if inclusionSkelRec( $T_j, U_{ij}, A'$ ) = false then return(false)
    return(true)
  else return(false)

```

## 6.2.2 Inclusion of Tree Schemata

In the case of tree schemata, we must also check the inclusion of the relations represented by the links. But it can be performed in the way described by the constructive definition of the set represented by a tree schema (page 99) while going through the skeleton. This traversal of the tree schemata is necessary anyway to find the links and compare them. It is used to test the inclusion of the finite parts of the relations. For the infinite parts of the relations, we must compare the links globally, using the auxiliary informations.

### The Algorithm

`inclusion( $T, U$ ):`

```

if  $T = \bigcirc$  then return(true)
if  $U = \bigcirc$  then return(false)
Links  $\leftarrow \emptyset^G$ 
return(inclusionRec( $T, U, \{ \langle T_\emptyset, U_\emptyset \rangle \}, \emptyset \neq (\emptyset, \emptyset)$ ))

```

`inclusionRec( $T, U, \Gamma, A$ ):`

```

if  $(T, U) \in A$  then
   $l_1$  is the link in the loop of  $T$ , and  $l_2$  is the link in the loop of  $U$ 
  add  $(l_1, l_2)$  to LinksE (and  $l_1$  and  $l_2$  to LinksN if necessary)
  return( $\Gamma, \emptyset$ )
if  $T = \begin{array}{c} \bigcirc \overset{x}{\sim} l \\ \swarrow \quad \searrow \\ T_0 \dots T_n \end{array}$  then
  if  $U$  is labeled by a  $f \in F$  then
    if  $l$  is a link that preserves the best skeleton approximation then
      return( $\emptyset, \emptyset$ )
    else
       $\forall T_i$  not labeled by  $f$  do
         $(\Gamma', b) \leftarrow \text{instGlobRels}(\Gamma, l, x, i, 0)$ 
        if  $\Gamma' \neq \emptyset$  then return( $\emptyset, \emptyset$ )
      Let  $T_i$  be the only possible choice. ( $T_i$  is labeled by  $f$ )
       $(\Gamma', b) \leftarrow \text{instGlobRels}(\Gamma, l, x, i, 0)$ 
       $A' \leftarrow A \cup \{(T, U)\}$ 
       $(\Gamma'', L) \leftarrow \text{inclusionChildren}(T_i, U, \Gamma', A')$ 
      if  $T$  is the scope of some links in LinksN then
        add these links to  $L$ 
        if a subset of  $L$  is a connected part of Links then
           $L' \leftarrow$  the biggest such subset of  $L$ 
          remove  $L'$  from LinksN

```





$$U \text{ is } \begin{array}{c} \circlearrowleft^y h \\ \swarrow \searrow \\ U_0 \dots U_m \end{array}$$

```

if  $\exists U_i$  labeled by  $f$  then
   $(\Gamma', b) \leftarrow \text{instGlobRels}(\Gamma, l, x, i, 1)$ 
  if  $b$  then return  $(\emptyset, \emptyset)$ 
   $A' \leftarrow A \cup \{(T, U)\}$ 
   $\Gamma'', L \leftarrow \text{inclusionChildren}(T, U_i, \Gamma', A')$ 
  if  $U$  is the scope of some links in  $\text{Links}^N$  then
    add these links to  $L$ 
    if a subset of  $L$  is a connected part of  $\text{Links}$  then
       $L' \leftarrow$  the biggest such subset of  $L$ 
      remove  $L'$  from  $\text{Links}^N$ 
      if the links in  $L'$  of  $T$  are included in the links in  $L'$  of  $U$  then
        return  $(\Gamma'', L \setminus L')$ 
      else return  $(\emptyset, \emptyset)$ 
    return  $(\Gamma'', L)$ 
  else return  $(\emptyset, \emptyset)$ 

```

$\text{inclusionChildren}(T, U, \Gamma_0, A)$ :

$$T \text{ is } \begin{array}{c} f \\ \swarrow \searrow \\ T_0 \dots T_{n-1} \end{array}, \text{ and } U \text{ is } \begin{array}{c} f \\ \swarrow \searrow \\ U_0 \dots U_{n-1} \end{array}$$

```

 $A' \leftarrow A \cup \{(T, U)\}$ 
for  $i \leftarrow 0$  to  $n - 1$  do
   $(\Gamma_{i+1}, L_i) \leftarrow \text{inclusionRec}(T_i, U_i, \Gamma_i, A')$ 
   $L'_i \leftarrow L_i$  with the relative paths updated by  $i$ 
  if  $\Gamma_{i+1} = \emptyset$  then return  $(\emptyset, \emptyset)$ 
 $L \leftarrow \bigcup_{i < n} L'_i$ 
if  $T$  or  $U$  are scopes of some links in  $\text{Links}^N$  then
  add these links to  $L$ 
  if a subset of  $L$  is a connected part of  $\text{Links}$  then
     $L' \leftarrow$  the biggest such subset of  $L$ 
    remove  $L'$  from  $\text{Links}^N$ 
    if the links in  $L'$  of  $T$  are included in the links in  $L'$  of  $U$  then
      return  $(\Gamma_n, L \setminus L')$ 
    else return  $(\emptyset, \emptyset)$ 
  return  $(\Gamma_n, L)$ 

```

$\text{instGlobRels}(\Gamma, l, x, v, i)$ :

```

 $\Gamma' \leftarrow \emptyset$ 
 $b \leftarrow \text{false}$ 
 $\forall \mathcal{G} \in \Gamma$  do
   $G \leftarrow \mathcal{G}_{(i)}$ 
  if  $v \in G(l)_{(x)}$  then

```

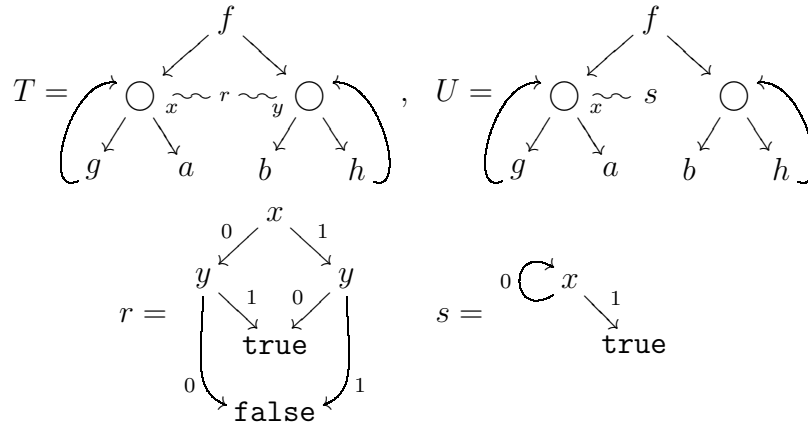
```

 $\mathcal{G}' \leftarrow \mathcal{G}$  where the  $i^{\text{th}}$  component has been replaced by  $G_{x=v}^l$ 
 $\Gamma' \leftarrow \Gamma \cup \{\mathcal{G}'\}$ 
else  $b \leftarrow \mathbf{true}$ 
return( $\Gamma', b$ )
    
```

### Inclusion of the Links

To decide the inclusion of the links of a tree schema in the links of another tree schema, we use an extended version of `inclusionRel` (or another basic inclusion algorithm for relations if they are represented by another mean than  $\omega$ -deterministic relation). The only difference is that we use sets of independent relations instead of relations. We just have to use the couples of sets instead of the couples of relations. As a guideline, we evaluate in sequence the entries in a vector of equivalent entries.

**Example:** We try to decide whether  $T$  is included in  $U$ :



The result of the computation on the finite part is that  $T$  can be included in  $U$ . But when we compare the links,  $r$  for  $T$ , and  $s$  and  $\mathbf{true}$  for  $U$ , we see that the evaluation 01 on entries 0 and 1 (relative paths to the root) give the same relations, but going through  $\mathbf{true}$  for  $r$ , and not for  $s$ . Thus,  $T$  is not included in  $U$ :  $f(g^\omega, h^\omega)$  is in  $T$  but not in  $U$ .  $\diamond$

## 6.3 Intersection

### 6.3.1 Intersection of Skeletons

Skeletons are closed by intersection, most of the work in the intersection of skeletons is performed by the algorithms on regular trees. The only thing that must

be performed to ensure the uniqueness of the representation is the propagation of the empty set.

**interSkel**( $T, U$ ):

**if**  $T = \bigcirc$  **or**  $U = \bigcirc$  **then return**( $\bigcirc$ )  
**return**(**interSkelRec**( $T, U, \emptyset$ ))

**interSkelRec**( $T, U, A$ ):

**if**  $T = U$  **then return**( $T$ )  
**if**  $\{T, U\} \in \text{dom}(A)$  **then return**( $A(\{T, U\})$ )  
 $X$  is a new label of arity 0  
 $A' \leftarrow A$  with  $\{T, U\} \rightarrow X$

**if**  $T = \begin{array}{c} \bigcirc \\ \swarrow \quad \searrow \\ T_0 \quad \dots \quad T_n \end{array}$  **then**

**if**  $U = \begin{array}{c} \bigcirc \\ \swarrow \quad \searrow \\ U_0 \quad \dots \quad U_m \end{array}$  **then**

$i \leftarrow 0$

**for**  $j \leftarrow 0$  **to**  $n$  **do**

**if**  $\exists U_k$  with the same label as  $T_j$  **then**

$S_i \leftarrow \text{interSkelRec}(T_j, U_k, A')$

**if**  $S_i \neq \bigcirc$  **then**  $i \leftarrow i + 1$

**if**  $i = 0$  **then return**( $\bigcirc$ )

**if**  $i = 1$  **then return**(**recCons**( $S_0, X$ ))

**return**(**recCons**( $\begin{array}{c} \bigcirc \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_i \end{array}, X$ ))

**else**  $U$  is  $\begin{array}{c} f \\ \swarrow \quad \searrow \\ U_0 \quad \dots \quad U_{m-1} \end{array}$

**if**  $\exists T_i$  labeled by  $f$  **then return**(**recCons**(**interSkelRec**( $T_i, U, A'$ ),  $X$ ))

**return**( $\bigcirc$ )

**else if**  $U = \begin{array}{c} \bigcirc \\ \swarrow \quad \searrow \\ U_0 \quad \dots \quad U_n \end{array}$  **then**

**if**  $\exists U_i$  with the same label as  $T$  **then return**(**recCons**(**interSkelRec**( $T, U_i, A'$ ),  $X$ ))

**else return**( $\bigcirc$ )

$T$  is labeled by  $\begin{array}{c} f \\ \swarrow \quad \searrow \\ T_0 \quad \dots \quad T_{n-1} \end{array}$ , and  $U$  is labeled by  $\begin{array}{c} g \\ \swarrow \quad \searrow \\ U_0 \quad \dots \quad U_{m-1} \end{array}$

**if**  $f \neq g$  **then return**( $\bigcirc$ )

**for**  $i \leftarrow 0$  **to**  $n - 1$  **do**

$S_i \leftarrow \text{interSkelRec}(T_i, U_i, A')$

**if**  $S_i = \bigcirc$  **then return**( $\bigcirc$ )

**return**(**recCons**( $\begin{array}{c} f \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_{n-1} \end{array}, X$ ))

### 6.3.2 Intersection of Tree Schemata

In the intersection of tree schemata, we can use global informations to take advantage of the necessary traversal of the tree schemata to deal with the finite parts of the relations at the same time, as for inclusion. But it leads to a quite intricate algorithm. Instead, we just partially deal with the finite part of the relations. The property that we use is the following:  $Set(T) \cap Set(U) \subset Ens(\text{interSkel}(\text{skel}(T), \text{skel}(U)))$ . This property holds even if there is no best skeleton approximation. So the algorithm proceeds in three steps: we go through the schemata to compute the intersection of their skeletons and the least common scopes. When we reach an independent least common scope, we compute the intersection of the relations represented by the links on the new skeleton. Then, we may update the skeleton so that we still have the best skeleton approximation, if possible.

#### Main Algorithm: Traversal of the Tree Schemata

The main algorithm consists in going through the tree schemata to compute, as a first approximation, the intersection of their skeletons. Meanwhile, we collect some informations on the links to help the computation of their intersection when we reach a common scope. This information consists in a sequence associated with every entry name of every link. This sequences are made of subsets of the range of the entries, corresponding to the entries that are kept in the intersection of the skeletons.

```

intersection( $T, U$ ):
  if  $T = \bigcirc$  or  $U = \bigcirc$  then return( $\bigcirc$ )
  Links  $\leftarrow \emptyset^G$ 
  Restrictions is a function that associates the empty sequence with each entry
  name of each link
  ( $S, L$ )  $\leftarrow$  interRec( $T, U, \emptyset$ )
  return( $S$ )

interRec( $T, U, A$ ):
  if  $T = U$  then return( $T, \emptyset$ )
  if  $\{T, U\} \in \text{dom}(A)$  then return( $A(\{T, U\}), \emptyset$ )
   $X$  is a new label of arity 0
   $A' \leftarrow A$  with  $\{T, U\} \rightarrow X$ 
  if  $T = \begin{array}{c} \bigcirc \overset{x}{\sim} l \\ \swarrow \quad \searrow \\ T_0 \quad \dots \quad T_n \end{array}$  then
    if  $U = \begin{array}{c} \bigcirc \overset{y}{\sim} h \\ \swarrow \quad \searrow \\ U_0 \quad \dots \quad U_m \end{array}$  then
      |  $i \leftarrow 0$ 

```

```

|  $I \leftarrow \emptyset, J \leftarrow \emptyset$ 
| for  $j \leftarrow 0$  to  $n$  do
|   if  $\exists U_k$  with the same label as  $T_j$  then
|      $(S_i, L_i) \leftarrow \text{interRec}(T_j, U_k, A')$ 
|     if  $S_i \neq \bigcirc$  then
|       |  $I \leftarrow I \cup \{j\}$ 
|       |  $J \leftarrow J \cup \{k\}$ 
|       |  $i \leftarrow i + 1$ 
|   if  $i = 0$  then return $(\bigcirc, \emptyset)$ 
|   add  $I$  to Restrictions( $l, x$ ), and  $J$  to Restrictions( $h, y$ )
|   add  $(l, h)$  to LinksE
|    $L \leftarrow \bigcup_{j < i} L_j$ 
|   if  $i = 1$  then  $S \leftarrow \text{recCons}(S_0, X)$ 
|   else  $S \leftarrow \text{recCons} \left( \begin{array}{c} \bigcirc \\ \swarrow \searrow \\ S_0 \dots S_i \end{array}, X \right)$ 
|
| else  $U$  is  $\begin{array}{c} f \\ \swarrow \searrow \\ U_0 \dots U_{m-1} \end{array}$ 
|   if  $\exists T_i$  labeled by  $f$  then
|     add  $\{i\}$  to Restrictions( $l, x$ )
|      $(S, L) \leftarrow \text{recCons}(\text{interRec}(T_i, U, A'), X)$ 
|   else return $(\bigcirc, \emptyset)$ 
|
| else if  $U = \begin{array}{c} \bigcirc \overset{x}{\sim} l \\ \swarrow \searrow \\ U_0 \dots U_n \end{array}$  then
|   if  $\exists U_i$  with the same label as  $T$  then
|     | add  $\{i\}$  to Restrictions( $l, x$ )
|     |  $(S, L) \leftarrow \text{recCons}(\text{interRec}(T, U_i, A'), X)$ 
|   else return $(\bigcirc, \emptyset)$ 
|
| else  $T$  is labeled by  $\begin{array}{c} f \\ \swarrow \searrow \\ T_0 \dots T_{n-1} \end{array}$ , and  $U$  is labeled by  $\begin{array}{c} g \\ \swarrow \searrow \\ U_0 \dots U_{m-1} \end{array}$ 
|   if  $f \neq g$  then return $(\bigcirc, \emptyset)$ 
|   for  $i \leftarrow 0$  to  $n - 1$  do
|     |  $(S_i, L_i) \leftarrow \text{interRec}(T_i, U_i, A')$ 
|     | if  $S_i = \bigcirc$  then return $(\bigcirc, \emptyset)$ 
|     |  $L'_i \leftarrow L_i$  with the relative paths updated by  $i$ 
|    $L \leftarrow \bigcup_{i < n} L'_i$ 
|    $S \leftarrow \text{recCons} \left( \begin{array}{c} f \\ \swarrow \searrow \\ S_0 \dots S_{n-1} \end{array}, X \right)$ 
|   return $(\text{interLink}(T, U, S, L))$ 
|
interlink( $T, U, S, L$ ):
| if  $T$  or  $U$  is the scope of a link then
|    $L' \leftarrow L$  plus those links

```

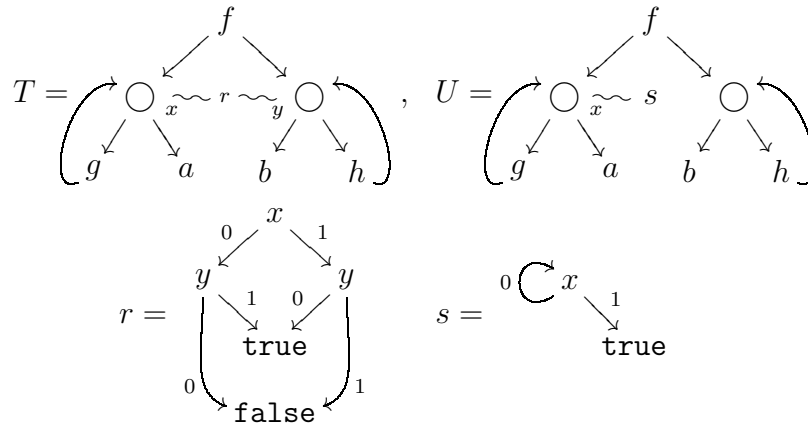
```

if a subset of  $L'$  is a connected part of Links then
   $L'' \leftarrow$  the biggest such subset of  $L'$ 
  remove  $L''$  from Links $N$  and  $L'$ 
  compute the intersection of the links in  $L''$  and add this link in  $S$ 
   $S' \leftarrow S$  updated so that it satisfies the best skeleton approximation
  return( $S', L'$ )
return( $S, L'$ )
return( $S, L$ )
  
```

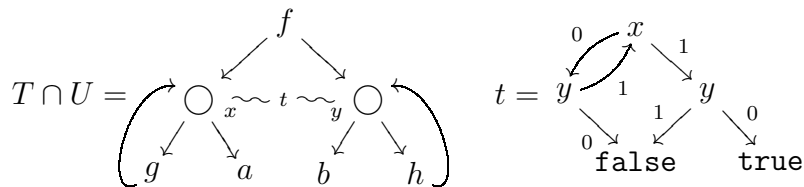
### Intersection of the Links

To compute the new link that is the intersection of the links of the two tree schemata, we use an extension of the intersection algorithm on relations (**inter-Rel**, for  $\omega$ -deterministic relations), just as for inclusion. We use the information computed in **Restrictions**, and the relative paths of the links to harmonize the two sets of links of the two tree schemata. Then, the only difference with the basic algorithm is the use of vectors of relations for loop testing instead of mere relations. This new link is then added to the skeleton that was computed so far.

**Example:** We compute the intersection of  $T$  and  $U$  defined below:



The two tree schemata have the same skeleton, so the first approximation of the skeleton of their intersection gives the same skeleton. The intersection of  $r$  and  $(s, \mathbf{true})$  gives the relation  $t$  defined below:



◇

### Updating the Skeleton According to the New Link

The skeleton of the new tree schema must respect the rules defined section 5.4.1 (page 106) so that as much information as possible is put in the skeleton. To update correctly the skeleton, we must analyze the relation  $R$  of the link. To insure the fullness of the relation, we first compute the different  $R_{(x)}$  for all  $x$  entry name.

$\forall x$  entry name of  $R$  **do**

**if**  $R_{(x)} = \emptyset$  **then**

replace every node  $\begin{array}{c} \circ \overset{x}{\sim} R \\ \swarrow \searrow \\ T_0 \dots T_n \end{array}$  by  $\circ$ .

propagate the information until no  $\circ$  appears in  $S$  or  $S = \circ$

**else if**  $R_{(x)} = \{i\}$  **then**

replace every node  $\begin{array}{c} \circ \overset{x}{\sim} R \\ \swarrow \searrow \\ T_0 \dots T_n \end{array}$  by  $T_i$

**else if**  $|R_{(x)}| \leq n$  **then**

remove every  $T_i$  such that  $i \notin R_{(x)}$  from  $\begin{array}{c} \circ \overset{x}{\sim} R \\ \swarrow \searrow \\ T_0 \dots T_n \end{array}$

change the values of  $x$  such that  $x$  ranges from 0 to  $|R_{(x)}| - 1$

The last step to enforce the rules is to remove the link from every node linked by an  $x$  such that  $x$  does not appear in the decision diagram of  $R$ .

All these operations give a new tree schema  $S'$  valid with respect to the definition of section 5.4.1, except possibly for the last rule. This rule asks that if there is a best skeleton approximation,  $skel(S')$  is this skeleton.

In order to compute the tree schema with best skeleton, we can follow the instructions described below:

**while**  $skel(S')$  is not the best skeleton for  $Ens(S')$  **do**

there is  $T \triangleleft U \triangleleft S'$  such that:

$T$  is labeled by  $(\circ, x, l)$ ,  $U$  is labeled by  $(\circ, y, l)$  and

there is a path  $p$  from  $U$  to  $T$  that does not allow every choice of  $T$

replace  $U_{[p]}$  by the new restrained choice node

If  $Ens(S')$  admits a best skeleton approximation, then the process terminates. Sometimes, we can recognize whether the set admits a best skeleton approximation. It is the case if both the tree schemata of which  $S'$  is the intersection admit best skeletons. We can also recognize cases of no best skeleton approximation as described in the second case page 104. But in some other cases, we don't have any efficient algorithm to decide whether the process of best skeleton construction will terminate. In these cases, we can stop after an arbitrary number of iterations and let  $skel(S')$  not be the best skeleton of  $Ens(S')$ . The only effect of this approximation is a reduction of the efficiency of further algorithms on  $S'$ .



## 6.4 Union

### 6.4.1 Union of Skeletons

Union is interesting in that it introduces new links, even when we start from mere skeletons with no links. Note that if we don't add links, we get the best skeleton containing the union of  $T$  and  $U$ .

$\text{unionSkel}(T, U)$ :

```

if  $T = \bigcirc$  then return( $U$ )
if  $U = \bigcirc$  then return( $T$ )
return( $\text{unionSkelRec}(T, U, \emptyset)$ )

```

$\text{unionSkelRec}(T, U, A)$ :

```

if  $T = U$  then return( $T$ )
if  $\{T, U\} \in A$  then return( $A(\{T, U\})$ )
 $X$  is a new label of arity 0
 $A' \leftarrow A$  with  $\{T, U\} \rightarrow X$ 

```

```

if  $T = \begin{array}{c} \bigcirc \\ \swarrow \quad \searrow \\ t_0 \quad \dots \quad t_n \end{array}$  then

```

```

if  $U = \begin{array}{c} \bigcirc \\ \swarrow \quad \searrow \\ u_0 \quad \dots \quad u_m \end{array}$  then

```

```

 $i \leftarrow 0, j \leftarrow 0, k \leftarrow 0$ 

```

```

while  $j \leq n$  and  $k \leq m$  do

```

```

 $f$  is the label of  $T_j$  and  $g$  is the label of  $U_k$ 

```

```

if  $f <_F g$  then

```

```

 $S_i \leftarrow T_j$ 

```

```

 $j \leftarrow j + 1$ 

```

```

else if  $g <_F f$  then

```

```

 $S_i \leftarrow U_k$ 

```

```

 $k \leftarrow k + 1$ 

```

```

else ( $f = g$ )

```

```

 $S_i \leftarrow \text{unionSkelRec}(T_j, U_k, A)$ 

```

```

 $j \leftarrow j + 1, k \leftarrow k + 1$ 

```

```

 $i \leftarrow i + 1$ 

```

```

while  $j \leq n$  do

```

```

 $S_i \leftarrow T_j$ 

```

```

 $i \leftarrow i + 1, j \leftarrow j + 1$ 

```

```

while  $k \leq m$  do

```

```

 $S_i \leftarrow U_k$ 

```

```

 $i \leftarrow i + 1, k \leftarrow k + 1$ 

```

```

return  $\left( \text{recCons} \left( \begin{array}{c} \bigcirc \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_{i-1} \end{array}, X \right) \right)$ 

```

$U$  is labeled by  $g$

**if**  $\exists T_i$  labeled by  $g$  **then**

$S_i \leftarrow \text{unionSkelRec}(T_i, U, A)$

$\forall j \leq n, j \neq i$  **do**  $S_j \leftarrow T_j$

**return**  $\left( \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_n \end{array}, X \right) \right)$

**else**

$i$  is the index of insertion of  $g$  in the sequence of labels of  $T_j$

$\forall j < i$  **do**  $S_j \leftarrow T_j$

$S_i \leftarrow U$

$\forall j \in [i, n + 1[$  **do**  $S_{j+1} \leftarrow T_j$

**return**  $\left( \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_{n+1} \end{array}, X \right) \right)$

**else if**  $U = \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ U_0 \quad \dots \quad U_n \end{array}$  **then**

$T$  is labeled by  $f$

**if**  $\exists U_i$  labeled by  $f$  **then**

$S_i \leftarrow \text{unionSkelRec}(T, U_i, A)$

$\forall j \leq n, j \neq i$  **do**  $S_j \leftarrow U_j$

**return**  $\left( \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_n \end{array}, X \right) \right)$

**else**

$i$  is the index of insertion of  $f$  in the sequence of labels of  $U_j$

$\forall j < i$  **do**  $S_j \leftarrow U_j$

$S_i \leftarrow T$

$\forall j \in [i, n + 1[$  **do**  $S_{j+1} \leftarrow U_j$

**return**  $\left( \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_{n+1} \end{array}, X \right) \right)$

**else**  $T = \begin{array}{c} f \\ \swarrow \quad \searrow \\ T_0 \quad \dots \quad T_{n-1} \end{array}$  and  $U = \begin{array}{c} g \\ \swarrow \quad \searrow \\ U_0 \quad \dots \quad U_{n-1} \end{array}$

**if**  $f < g$  **then return**  $\left( \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ T \quad \quad U \end{array}, X \right) \right)$

**if**  $f > g$  **then return**  $\left( \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ U \quad \quad T \end{array}, X \right) \right)$

**for**  $i \leftarrow 0$  **to**  $n - 2$  **do**

$(S_i, L_i) \leftarrow \text{unionWithLinks}(T_i, U_i, A)$

**if** all  $L_i$  are empty **then**

$S_{n-1} \leftarrow \text{unionSkelRec}(T_{n-1}, U_{n-1}, A)$

**return**  $\left( \text{recCons} \left( \begin{array}{c} f \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_{n-1} \end{array}, X \right) \right)$

```

 $S_{n-1} \leftarrow \text{unionWithLinks}(T_{n-1}, U_{n-1}, A)$ 
 $S \leftarrow \text{recCons} \left( \begin{array}{c} f \\ \swarrow \searrow \\ S_0 \dots S_{n-1} \end{array}, X \right)$ 
if more than one  $L_i$  is not empty then
   $L \leftarrow S \rightarrow \bigcup_{i < n} \text{im}(L_i)$ 
else  $L \leftarrow \bigcup_{i < n} L_i$ 
makeUnionLink( $L, S$ )
return( $S$ )

```

$\text{unionWithLinks}(T, U, A)$ :

```

if  $T = U$  then return( $T$ )
if  $\{T, U\} \in A$  then return( $A(\{T, U\})$ )
 $X$  is a new label of arity 0
 $A' \leftarrow A$  with  $\{T, U\} \rightarrow X$ 
if  $T = \begin{array}{c} \circ \\ \swarrow \searrow \\ t_0 \dots t_n \end{array}$  then
  if  $U = \begin{array}{c} \circ \\ \swarrow \searrow \\ u_0 \dots u_m \end{array}$  then
     $i \leftarrow 0, j \leftarrow 0, k \leftarrow 0$ 
     $I \leftarrow \emptyset, J \leftarrow \emptyset$ 
     $L \leftarrow \emptyset$ 
    while  $j \leq n$  and  $k \leq m$  do
       $f$  is the label of  $T_j$  and  $g$  is the label of  $U_k$ 
      if  $f <_F g$  then
         $S_i \leftarrow T_j$ 
         $I \leftarrow I \cup \{i\}$ 
         $j \leftarrow j + 1$ 
      else if  $g <_F f$  then
         $S_i \leftarrow U_k$ 
         $J \leftarrow J \cup \{i\}$ 
         $k \leftarrow k + 1$ 
      else ( $f = g$ )
         $(S_i, L') \leftarrow \text{unionWithLinks}(T_j, U_k, A)$ 
         $L \leftarrow L \cup L'$ 
         $I \leftarrow I \cup \{i\}, J \leftarrow J \cup \{i\}$ 
         $j \leftarrow j + 1, k \leftarrow k + 1$ 
       $i \leftarrow i + 1$ 
    while  $j \leq n$  do
       $S_i \leftarrow T_j$ 
       $I \leftarrow I \cup \{i\}$ 
       $i \leftarrow i + 1, j \leftarrow j + 1$ 
    while  $k \leq m$  do

```

```

     $S_i \leftarrow U_k$ 
     $J \leftarrow J \cup \{i\}$ 
     $i \leftarrow i + 1, k \leftarrow k + 1$ 
     $S \leftarrow \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_{i-1} \end{array}, X \right)$ 
    if  $I \neq J$  then  $L \leftarrow L \cup \circ \rightarrow (S, I, J)$ 
    return( $S, L$ )
   $U$  is labeled by  $g$ 
  if  $\exists T_i$  labeled by  $g$  then
     $(S_i, L) \leftarrow \text{unionWithLinks}(T_i, U, A)$ 
     $\forall j \leq n, j \neq i$  do  $S_j \leftarrow T_j$ 
     $I \leftarrow [n + 1]$ 
     $J \leftarrow \{i\}$ 

     $S \leftarrow \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_n \end{array}, X \right)$ 
    return( $S, L \cup \circ \rightarrow (S, I, J)$ )
  else
     $i$  is the index of insertion of  $g$  in the sequence of labels of  $T_j$ 
     $\forall j < i$  do  $S_j \leftarrow T_j$ 
     $S_i \leftarrow U$ 
     $\forall j \in [i, n + 1[$  do  $S_{j+1} \leftarrow T_j$ 
     $J \leftarrow \{i\}$ 
     $I \leftarrow [n + 2] \setminus \{i\}$ 

     $S \leftarrow \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_{n+1} \end{array}, X \right)$ 
    return( $S, \circ \rightarrow (S, I, J)$ )
  else if  $U = \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ U_0 \quad \dots \quad U_n \end{array}$  then
     $T$  is labeled by  $f$ 
    if  $\exists U_i$  labeled by  $f$  then
       $(S_i, L) \leftarrow \text{unionWithLinks}(T, U_i, A)$ 
       $\forall j \leq n, j \neq i$  do  $S_j \leftarrow U_j$ 
       $I \leftarrow \{i\}$ 
       $J \leftarrow [n + 1]$ 

       $S \leftarrow \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \quad \searrow \\ S_0 \quad \dots \quad S_n \end{array}, X \right)$ 
      return( $S, L \cup \circ \rightarrow (S, I, J)$ )
    else
       $i$  is the index of insertion of  $f$  in the sequence of labels of  $U_j$ 
       $\forall j < i$  do  $S_j \leftarrow U_j$ 
       $S_i \leftarrow T$ 

```

```

|  $\forall j \in [i, n + 1[$  do  $S_{j+1} \leftarrow U_j$ 
|  $I \leftarrow \{i\}$ 
|  $J \leftarrow [n + 2] \setminus \{i\}$ 
|  $S \leftarrow \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \searrow \\ S_0 \dots S_{n+1} \end{array}, X \right)$ 
| return $(S, \circ \rightarrow (S, I, J))$ 
else  $T = \begin{array}{c} f \\ \swarrow \searrow \\ T_0 \dots T_{n-1} \end{array}$  and  $U = \begin{array}{c} g \\ \swarrow \searrow \\ U_0 \dots U_{n-1} \end{array}$ 
if  $f < g$  then
|  $S \leftarrow \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \searrow \\ T \quad U \end{array}, X \right)$ 
| return $(S, S \rightarrow (S, \{0\}, \{1\}))$ 
else if  $f > g$  then
|  $S \leftarrow \text{recCons} \left( \begin{array}{c} \circ \\ \swarrow \searrow \\ U \quad T \end{array}, X \right)$ 
| return $(S, S \rightarrow (S, \{1\}, \{0\}))$ 
for  $i \leftarrow 0$  to  $n - 1$  do
   $(S_i, L_i) \leftarrow \text{unionWithLinks}(T_i, U_i, A)$ 
 $S \leftarrow \text{recCons} \left( \begin{array}{c} f \\ \swarrow \searrow \\ S_0 \dots S_{n-1} \end{array}, X \right)$ 
if more than one  $L_i$  is not empty then
   $L \leftarrow S \rightarrow \bigcup_{i < n} \text{im}(L_i)$ 
else  $L \leftarrow \bigcup_{i < n} L_i$ 
return $(S, L)$ 

```

```

makeUnionLink( $L, T$ ):
 $\forall U \in \text{dom}(L) \setminus \{\circ\}$  do
   $\{(C_0, I_0, J_0), \dots, (C_n, I_n, J_n)\}$  is  $L(U)$ 
  Let  $x_0, \dots, x_n$  be a set of entry names
   $R \leftarrow \bigotimes_{i \leq n} I_i \cup \bigotimes_{i \leq n} J_i$ 
  if  $R \neq \bigotimes_{i \leq n} I_i \cup J_i$  then
     $l \leftarrow$  the link defined by  $R$ , and  $x_i \rightarrow C_i$ 
    replace the label  $\circ$  of  $C_i$  by  $\circ_{x_i \rightsquigarrow l}$ 
    register the scope of  $l$  to be  $U$ 

```

A consequence of the algorithm is that if the labels in  $F$  are of arity 0 or 1 only, no link is created during the computation of the union. As union is the only basic algorithm introducing links, it means that as long as we manipulate sets of vectors with basic algorithms, we will not have any link in the tree schemata. This property justifies the use of tree schemata to represent relations (section 4.4.1).

## 6.4.2 Union of Tree Schemata

The extension of union to tree schemata is the same as the extension of intersection to tree schemata: while going through the two tree schemata, we build the union of their skeletons, and gather informations about the links. When we attain a least common scope, we compute the union of the links, in the same way as for intersection of links.

The only difference is the introduction of new links by the sharing of common prefixes. It means that we must also compute the intersection of the links with the additional links generated by the union of the skeletons. A consequence is that we don't have to detect least common scope when we are in the phase `unionWithLinks` of the algorithms. We can wait until we reach the scope of the link generated by union, that is until we are in `unionRec` or `makeUnionLink`.

Remember also another difference with intersection if we use  $\omega$ -deterministic relations: in that case, we might sometimes approximate the union of the links. In this case, the algorithm will just compute an approximation of the union.

## 6.5 Projection

Projection is an important algorithm because it is —with union— one of the constructive algorithm used in section 5.4.2 to describe the set of tree schemata.

### 6.5.1 Definition

Projection concerns only some paths in the tree schema. Suppose for example that  $g \in F$  of arity 3. One possible projection would be the set of trees that appear at first position. Then we don't care about the second or third position. To describe this fact, we introduce a new label,  $\Omega$ , which can stand for any subtree. Let  $t$  be a tree on  $F \cup \{\Omega\}$ . We say that  $t$  is a possible prefix in the set of trees  $E$ , and we write  $t \in_{\Omega} E$  if there is a function  $f$  from paths  $p$  such that  $t(p) = \Omega$  to  $\mathcal{H}(F)$  such that the tree  $u$  defined below is in  $E$ :  $v(q) = t(q)$  if  $t(q) \neq \Omega$  and  $v_{[p]} = f(p)$  if  $t(p) = \Omega$ .

Let  $t$  be a tree on  $F \cup \{\Omega, x\}$  and  $T$  a tree schema. The projection of  $T$  on  $t$  according to  $x$  is defined as  $t_x^{-1}(T) \stackrel{\text{def}}{=} \{u \in \mathcal{H}(F) \mid t[x \setminus u] \in_{\Omega} \text{Set}(T)\}$ . Note that projection subsumes intersection: Let  $T$  and  $U$  be two tree schemata, and  $\cap$  a new label of arity two then:  $T \cap U = \underset{x}{\underset{x}{\underset{x}{\downarrow}} \cap}^{-1} \left( \underset{T}{\underset{U}{\downarrow}} \right)$

We define tree projection with respect to one variable only, but it is possible to define it with respect to a finite vector of variables (the result is a set of vectors of trees). The algorithms below can easily be extended to that case, but for the simplicity of the presentation, we will only consider one variable.

### 6.5.2 Projection of Skeletons

We progressively refine the set  $t_x^{-1}(T)$  as we go through  $T$  and  $t$ . At the beginning, this set is  $\mathcal{H}(F)$  (which is in  $\mathcal{Sk}(F)$ , the set of sets of trees that can be represented by a skeleton, because  $F$  is finite). When we discover that  $t[x \setminus \Omega] \notin_{\Omega} \text{Ens}(T)$ , this set becomes empty and we return. When we come to a new path  $p$  such that  $t(p) = x$ , we intersect the set with the skeleton at this point. So we just need to visit the distinct  $T_{[p]}$  such that  $t(p) = x$ .

`projSkel( $t, x, T$ ):`

```

Result  $\leftarrow \mathbf{1}_F$ 
Mem  $\leftarrow \emptyset$ 
projSkelRec( $t, x, T$ )
return(Result)

```

`projSkelRec( $t, x, T$ ):`

```

if  $t \neq \Omega$  and  $(t, T) \notin \text{Mem}$  then
  add  $(t, T)$  to Mem
  if  $t = x$  then
    | Result  $\leftarrow \text{interSkel}(\text{Result}, T)$ 
  else  $t$  is  $\begin{matrix} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{matrix}$ 
    | if  $T = \begin{matrix} \circ \\ \swarrow \searrow \\ T_0 \dots T_m \end{matrix}$  then
      | if  $\exists T_i$  labeled by  $f$  then projSkelRec( $t, x, T_i$ )
      | else Result  $\leftarrow \circ$ 
    | else  $T$  is  $\begin{matrix} g \\ \swarrow \searrow \\ T_0 \dots T_{m-1} \end{matrix}$ 
      | if  $f = g$  then
        | for  $i \leftarrow 0$  to  $n - 1$  do projSkelRec( $t_i, x, T_i$ )
      | else Result  $\leftarrow \circ$ 

```

### 6.5.3 Projection of Tree Schemata

The computation of the projection of the skeleton corresponds to a kind of intersection algorithm between the skeleton and  $t[\Omega \setminus \mathbf{1}_F][x \setminus \mathbf{1}_F]$ , plus an intersection of all subtrees corresponding to  $x$ . We can use the same idea in the case of tree schemata, but this will give links linking nodes in the results to nodes outside of the result. It is a desirable feature during the computation, as it allows the different prefix requirements to affect the result, but at the end of the computation, we must forget these links.

As was the case for intersection, we can use a global information to deal with the finite part of the relations, taking advantage of the necessary traversal of the

tree schema. But in order to simplify the presentation, we will collect the least common scope of every relation and use the algorithms on relations for every link.

projection( $t, x, T$ ):

```

Result  $\leftarrow$   $1_F$ 
Mem  $\leftarrow$   $\emptyset$ 
Links  $\leftarrow$   $\emptyset^G$ 
projRec( $t, x, T$ )
forget in the links every entry which is linked outside of Result
return(Result)

```

projRec( $t, x, T$ ):

```

L  $\leftarrow$   $\emptyset$ 
if  $t \neq \Omega$  and  $(t, T) \notin$  Mem then
  add  $(t, T)$  to Mem
  if  $t = x$  then
    | (Result, L)  $\leftarrow$  interRec(Result, T,  $\emptyset$ )
  else  $t$  is  $\begin{matrix} f \\ \swarrow \searrow \\ t_0 \dots t_{n-1} \end{matrix}$ 
    | if  $T = \begin{matrix} \circ \overset{x}{\sim} l \\ \swarrow \searrow \\ T_0 \dots T_m \end{matrix}$  then
      | if  $\exists T_i$  labeled by  $f$  then  $L \leftarrow$  projRec( $t, x, T_i$ )
      | else Result  $\leftarrow$   $\circ$ 
    else  $T$  is  $\begin{matrix} g \\ \swarrow \searrow \\ T_0 \dots T_{m-1} \end{matrix}$ 
      | if  $f = g$  then
        | for  $i \leftarrow 0$  to  $n - 1$  do  $L_i \leftarrow$  projRec( $t_i, x, T_i$ )
        |  $L \leftarrow \bigcup_{i < n} L_i$ 
        | else Result  $\leftarrow$   $\circ$ 
    if  $T$  is the scope of a link then
      |  $L' \leftarrow$   $L$  plus those links
      | if a subset of  $L'$  is a connected part of Links then
        |  $L'' \leftarrow$  the biggest such subset of  $L'$ 
        | remove  $L''$  from LinksN and  $L'$ 
        | compute the intersection of the links in  $L''$  and add this link in Result
      | return( $L'$ )
    return(L)

```



## 6.6 Meta Expressions

We overview how we can use the algorithms described above to compute the meta expressions of section 5.4.2 as language transformers. We suppose we have a finite set of set variables,  $\mathcal{V}$ , and that each set variable  $\mathcal{X}$  is associated with the tree schema  $T_{\mathcal{X}}$ . We try to compute the value  $T_{e_{\mathcal{X}}}$  which is the tree schema representing the semantics of the meta expression  $e_{\mathcal{X}}$  in the context of the current values of the set variables.

Meta expressions are finite and can be computed inductively. If the meta expression is a set variable  $\mathcal{X}$ , then its new value is  $T_{\mathcal{X}}$ . If it is a union  $e_1 \cup e_2$ , then its new value is the union of  $T_{e_1}$  and  $T_{e_2}$ .

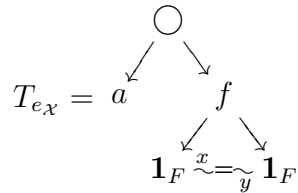
The last case is  $e = \{t' : t_1 \in e_1 \dots t_n \in e_n\}$ , with  $t'$  a regular tree over  $F \cup v$  of finite groundness and each  $t_i$  a regular tree over  $F \cup v$  ( $v$  is a set of term variables). We start by computing the set of variable values,  $t_{1v}^{-1}(T_{e_1})$ . It is performed with the **projection** algorithm. Then we modify slightly the **projection** algorithm, so that instead of starting from variables associated with the set of all possible trees, we start from the set of variable values that we just computed. And we keep going on until  $t_{nv}^{-1}(T_{e_n})$ .

Then, starting from a set of variable values, we compute the set of trees represented by  $t'$  in this context.  $t'$  is of finite groundness, so each variable  $x$  appears finitely many times, say  $p$  times. If  $p = 1$ , then we just have to make a tree substitution. If  $p > 1$ , we first make a tree substitution, and we intersect the links of the tree schemata corresponding to each occurrence of  $x$  with the link which links every choice node of the  $p^{\text{th}}$  occurrence of  $x$  with the corresponding choice node of the other occurrence. The entry name of each occurrence must be different (e.g.  $p$  for the  $p^{\text{th}}$  occurrence) and the relation of the link is the equality relation.

**Example:**

$$e_{\mathcal{X}} = \left\{ \begin{array}{c} f \\ \swarrow \quad \searrow \\ x \quad \quad x \end{array} : \begin{array}{c} g \\ \downarrow \\ x \end{array} \in \mathcal{Y} \right\} \cup \{a\} \quad \text{and } T_{\mathcal{X}} = \mathbf{1}_F \quad \text{and } T_{\mathcal{Y}} = \begin{array}{c} g \\ \downarrow \\ \mathbf{1}_F \end{array}$$

The result of the projection gives  $x \rightarrow \mathbf{1}_F$ . As a result, we get the following tree schema:



◇

## 6.7 Discussion

The algorithms described in this chapter are basic algorithms. They show how algorithms can be built to manipulate tree schemata. They depend highly on the algorithms described in chapters 3 and 4. As such, they may introduce approximations, for example in efficient union.

We saw in this chapter that the set of tree schemata is closed under finite basic operations such as union or projection. But still, in the domain of tree schemata, there are infinite increasing chains. Because of the regularity requirement for tree schemata, some of them don't even have a limit in the set of tree schemata. So we might need some further approximations.

The main source of complexity in tree schemata is the links. The fewer the number of links, the more sharing we can achieve, and the more efficient the algorithms. So, a first approximation strategy can attack the links. By removing a link, or maybe just a part of it, we obtain an upper approximation (for inclusion) of the tree schema. Because the links that induce sets with no best skeletons are the most difficult to handle, they may be the first target of this kind of approximation.

The second possibility for approximation is the skeleton of the tree schema itself. The easiest one consists in replacing a subtree of the tree schema by  $\mathbf{1}_F$ . The choice of the subtree might be given by the observation of the growth of a part of the tree schemata during an iteration. A little more precise approximation, in the spirit of [Sch97], can force the equality of two subtrees by an approximation by their union.

All these approximation strategies can be computed by algorithms in the same spirit as those described in this chapter. The problem is their lack of precision in the choice of which part of the tree schema to approximate. In the next chapter, we introduce an extension to tree schemata that allow a more precise analysis.

# Chapter 7

## Tree Schemata with Counters

Although already quite powerful, tree schemata may not be entirely satisfactory in abstract interpretation. In addition to the basic classical operations on sets of trees, abstract interpretation may use more complex operations in relation with the abstract values manipulated. In particular, during the computation of fixpoints, abstract interpretation may approximate the iteration through the use of widenings, which are based on abstraction of some iterates. In this case, we can use the approximations defined in chapter 6, but they are quite crude and may be inadequate. In fact we need to have a kind of direction of approximation in order to approximate more smartly.

One way of doing this is by counting the number of time a given pattern is repeated. If for example this number is increasing during many iterates, then it can be a good idea to directly replace the sequence of the pattern by an infinite (looping) pattern. To achieve this kind of analysis, we can add counters to the representation.

This chapter will develop on this extension. It will appear that this extension is easily integrated in tree schemata and comes with a wider acceptance of the notion of counters.

### 7.1 New Entries for the Links

When performing a decision process to decide the membership of a tree in a tree schema, we keep an environment, the partially evaluated relations, that can restrict the foregoing choices. In the same way, we can add entries to the links that are associated with other variables. There is no conceptual difference between entries associated with choice nodes outside of the current subtree of the tree schema and entries associated with some other variables outside of the choice nodes of the tree schema.

### 7.1.1 Tree Schemata on Labels and Variables

In order to enforce the strong determinism of tree schemata even in the presence of these new entries, we must still have full relations. To stay in the same frame, these variables must range over a finite domain which we can map to a natural number. We call  $\Upsilon$  the set of such variables. We can now speak of tree schemata over set of labels  $F$  and set of variables  $\Upsilon$ , members of  $\mathcal{TS}(F, \Upsilon)$ . Such tree schemata  $T$  are augmented with a finite set  $var(T) \subset \Upsilon$ . In a tree schema, each  $c \in var(T)$  is associated with an entry name and a link. Note that it means that a given variable cannot be associated with two different links, and thus cannot be the bond between two otherwise independent choice nodes. Variables of  $var(T)$  can now be members of the mapping of links.

Variables of tree schemata on labels and variables must still respect the rules of section 5.4.1: they must appear in the decision diagram of the link with which they are associated, and the relation of such links must be full. This last requirement means that, whatever the value of a variable, there is a vector in the relation corresponding to this variable. It means that if the variable is in  $var(T)$ , then whatever the value of the variable, the tree schema is not empty. Although this allows a fast emptiness testing procedure, it may be more useful to allow some values to be associated with the empty set. For the use of the variable in the links, we need a mapping from the domain of the variable to a  $[n]$ . We can use this mapping to associate the empty tree schema with some value of the variable: the mapping is not defined on such variables. In this way, we still have a very fast emptiness testing mechanism (depending on the values of the variables), but a more flexible use of the variables.

### 7.1.2 Many Variables in a Link

Because the set of variables can contain any reference, it is very possible that some of them interfere with each other out of the links of a given tree schema. The situation is quite similar to the relation between parent nodes, as these nodes are indeed related by the structure of the skeleton—a relation which is added and not related to the link between them—. The problem in this case is to keep the strong determinism in the representation (and the fast emptiness testing). In the former case, we just had to care about the choices that could lead from one node to the other, and in this case require no real relation. But in general, if we don't have any knowledge about two variables in a given link, the relation of the link must have a solution, *whatever* the combination of values of the variables. The same holds for more than two variables, of course. If in some cases, we know that some combinations of variables are not possible, we can add this information in the tree schema (and maybe have more precise skeletons) by replacing the two variables (or more) by a new one representing their combination, but whose domain excludes the impossible combinations.

### 7.1.3 Variables on Infinite Domains

The finitary condition for the variables domains is necessary to maintain some properties of the tree schemata, and in particular for an easy uniform representation of relations. But many useful properties may depend on variables ranging over an infinite domain. Suppose for example that a choice node depends on the number of loops in another tree schema. This number of loops can be as big as we want, but if this number is below a given limit, we take one choice, and another one if it is above. Although the domain of the variable is  $\mathbb{N}$ , what we need in fact is an abstraction of the variable, which is an element of the finite set  $\{x \leq l, x > l\}$ .

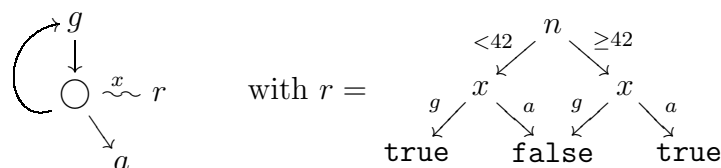
So, by means of abstract interpretation, it is possible to depend on infinite domains while admitting variables on finite domains only. In fact, the only thing we need is a many to one function going from the domain of a variable (or the domains of many variables) to a finite domain associated with another variable. Although not necessary, a full Galois connection might be useful, when, for example, we use the set of possible values of the variables as a result after the run of a tree on the tree schema. The point is, that from the point of view of the tree schema, as long as the variable is not closely related to the tree schema, we don't care. As long as we have variables on finite domains, we cannot in general take their origin or computation into account.

## 7.2 Interaction between Counters and Skeletons

The reason why we extended relations to other variables was the possibility they offer to use the value of counters in the decision process. But it is not the only possibility of interaction between counters and tree schemata. Tree schemata themselves can modify the counters during their evaluation. We present three ways of interacting in addition to affecting the decision process. The first one can be useful in a wide variety of contexts, and it is presented in details.

### 7.2.1 Choice Node Counters

With counters, we could use an alternative representation for the tree with 42  $g$  followed by a  $a$ . Instead of having indeed 42 nodes labeled by  $g$ , we could have:



where  $n$  is a variable representing the number of times the loop have been completed. Not only does it save space, but it can give an idea of what is growing

in a sequence  $ga, gga, ggga, \dots, g^{42}a, \dots$ . In the final version,  $r$  will as usual be labeled by natural numbers. The problem is how to relate  $n$  to a loop and how this will interact with operations on tree schemata.

### Counter Names

What we want to count is the number of times the decision process of a given tree goes through a given loop. One way of doing so is to concentrate on a choice where we can leave the loop and count the number of times we take the choices that don't leave the loop. The basic units to examine are described by a choice node and a choice number such that there is a path from this choice to the choice node.

### Counter Identification in the Tree Schema

As with any variable that ranges over an infinite domain, we must use a function that maps the value of the variable to a finite domain. We must also note in the tree schema which choices of which choice nodes are used as counters, so that we can increase the counter each time we go through the associated choice. To do this, in addition to an entry name and a link, we associate with choice nodes a counter list: a choice node is now a tuple  $(\bigcirc, x, l, cl)$ . The new element of the tuple,  $cl$ , is a partial map from the choices of the choice node to counter functions. If a given choice  $i$  is not used as a counter,  $cl(i)$  is not defined. A counter function is a kind of link between some counters and a given link of the tree schema.

Formally, a counter function is a many to one function of  $\mathbb{N}^k \rightarrow [n]$ , a link  $l$ , an entry name of range  $n$  of the link  $l$ , and a mapping  $f$  from each element of  $[k]$  to a choice node and the number of a choice of this choice node. We have  $cl(f(i))$  is the counter function. For example, with  $k = 1$ , the function can be a discrimination: `if  $n < 42$  then 0 else 1`. With  $k = 2$ , we can discriminate the sum of the counters, if both of them lead to the same loop, or compare the values of the two nodes, for example.

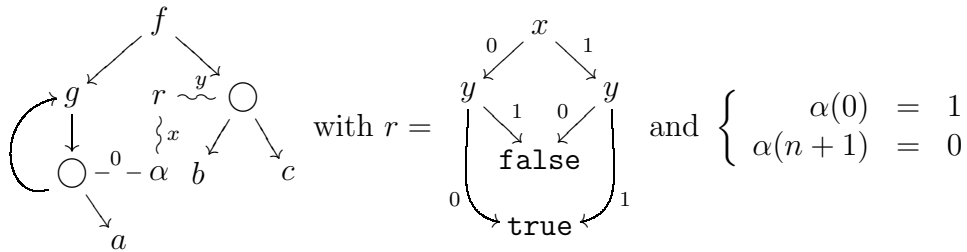
Because of this definition, a given counter can only be associated with one finite domain variable, the result of the counter function. It forces the link between two choice nodes that would be related through a given counter. When we want to use two different functions, we can always combine them to a function to a finer domain.

### Legible Links for Counter Functions

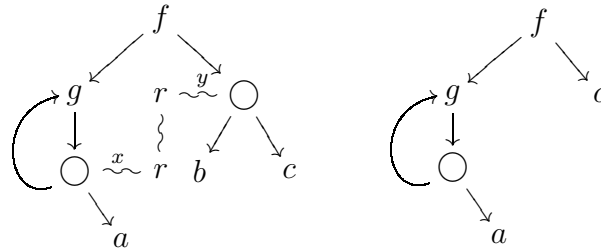
When using counters in a tree schema, we don't try anymore to use a best skeleton to support the tree schema. In the case of  $g^{42}a$ , it is not even a good idea. But with counters, we can have bigger problems: if we allow any link with counter, we can have tree schemata whose set of trees depend on the order of evaluation

in the schemata. This is due to a fundamental difference between entries linked to choice nodes and entries linked to counter functions: every time we need to evaluate a relation, the counter function has a value, and this value may change during the evaluation of the tree schema, whereas the choice may be evaluated or not, and can take one value only.

**Example:** Consider the following tree schema with counters:



This tree schema can be interpreted as any of the following tree schemata depending on whether we first explore the left child of  $f$  or its right child:



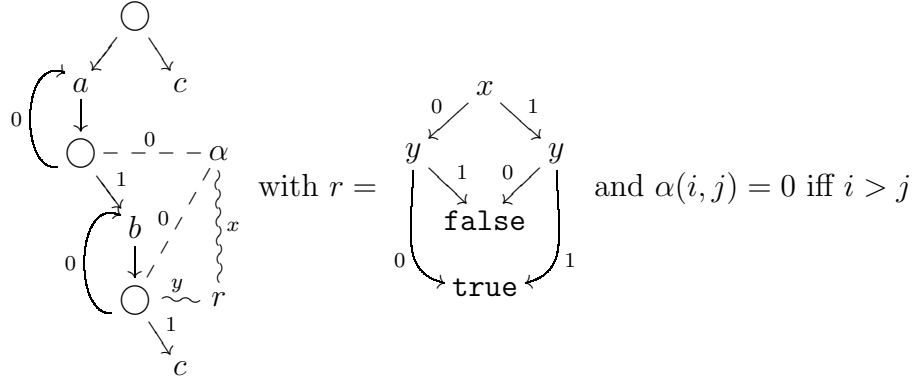
If we first explore the left child, then, when we come to the right child, the choice we can make depend on whether we have been in the loop or not. But if we first explore the right child, when we evaluate the relation, necessarily, the number of loops is 0, so the value of  $x$  is 1, so the only choice is  $c$ .  $\diamond$

To avoid this kind of ambiguity, we require that the choice nodes in the link appear after the counters, and that in the evaluation, there is no opportunity of ambiguity. Formally, whatever the choice node  $N$  and choice  $i$  of  $N$  linked to  $l$ , whatever the choice node  $M$  labeled by  $(\bigcirc, x, l, cl)$ , there is a  $p$  such that  $N.p = M$  and whatever  $q \prec p$ , if  $N.q$  is not a choice node, then no extension of  $q$  can lead to  $N$  without going through  $M$ .

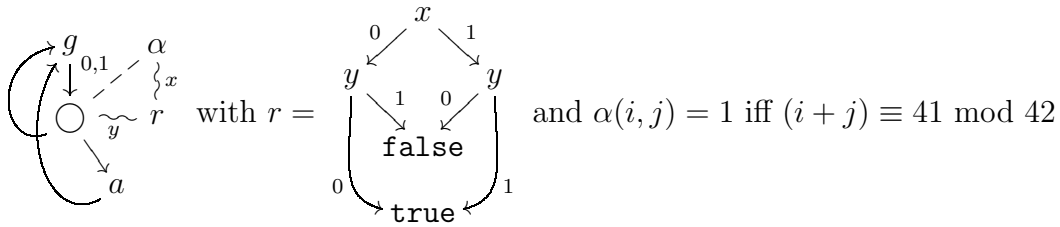
The second requirement just limits counters to choices of interest, which are the choices in a loop. Whatever the choice node  $N$  labeled by  $(\bigcirc, x, l, cl)$ , whatever  $i \in \text{dom}(cl)$ , there is a path  $p$  starting by  $i$  such that  $N.p = N$ .

**Examples:** We have an alternative to classical links with no best skeletons. We can use links based on counters instead. The following tree schema represents

the set  $\{a^n b^n c \mid n \in \mathbb{N}\}$ .



We also show an example of counter function with congruence:



This tree schema represents the infinite tree  $(g^{42}a)^\omega$ . ◇

### 7.2.2 Labels as Abstract Operators

In the last section, we saw that choice nodes could be used as operators on counters. In the same way, it is possible to use the different labels of  $F$  as abstract operators. In fact, each outgoing edge of the labels can be an operator. If we just have labels of arity 0 or 1, then tree schemata become abstract flowcharts, representing sets of traces.

In this case, tree schemata not only represent a set of trees, but can also be used to compute possible values of the counters. Such a use has some similarities with attribute grammars [Knu68, EF80].

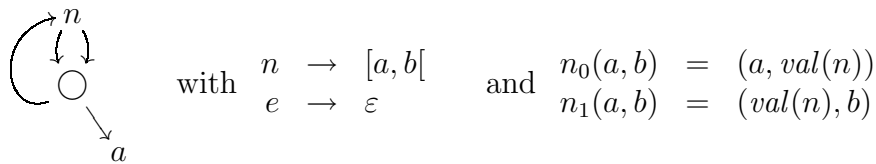
### 7.2.3 Counters as Labels

A limitation of tree schemata, necessary to represent the set of all trees,  $\mathcal{H}(F)$ , is that the set of labels,  $F$ , must be finite. In some cases, though, we might want an infinite set of labels. An easy way of dealing with an infinite set of labels  $F$  is the use of a finite set of labels  $F'$ , each label in  $F'$  representing a set of labels in  $F$ .



With the use of counters, we can have a more precise approximation of the set of labels represented by a given label in  $F'$ . This set of labels can be a function of the counters at the given point in the decision process.

**Example:** With this extension, we can represent the set of AVL trees. We use two counters,  $a$  and  $b$ ,  $a$  starting with the minimum integer value, and  $b$  the maximum integer value. The set is represented by the following tree schema, with the correspondence of the labels  $\{n, e\}$  to the set  $\mathbb{N} \cup \{\varepsilon\}$ , and the label operators,  $n_0$  and  $n_1$  ( $e_0$  does not change the values of the counters):



◇

## 7.3 Finding Cycles

By default, no counter is associated with a tree schema, except if it is expressly demanded by the user. But the user rarely knows which part of the tree schema will grow and can be best represented with counters. We present an algorithm to automatically discover such cycles during an iteration. It is then easy to approximate a tree schema by numerical analysis of the counters, in the way of [CH78] or [Mas92].

There are three problems with those new counted cycles: we must decide when and where to find those cycles, and we must fold those cycles as much as possible.

### 7.3.1 Deciding When to Look for a Cycle

To introduce counters in the tree schema, we look for parts of the tree schema where counters might be beneficial. Moreover, we must not spend the computation over tree schemata looking for such parts. So we propose a heuristic to decide to perform this computation.

The heuristic is based on the following remark: when we have a possibility to build a new cycle with counters, we have along a path a label which is repeated many times. So we can count the number of times each label appears in the tree schema, that is the number of distinct nodes of the tree schema with that label. If no label appears more than a given large number of times, then we know that there is no possibility for a beneficial cycle with counters.

Suppose now that there is a label which appears so many times that there might be a beneficial cycle with counters. Then we can start looking for such a cycle. If there is no such cycle, we can refine our information: when we have a possibility to build a new cycle with counters, we have along a path a label and its direct children repeated many times. So, for the label which appeared many times, we now divide the number of times the label appears between every possibility for the label of its children.

When this refining process becomes too expensive to compute, we can use some approximations (see next paragraph)

**Example:** Suppose  $F = \{a, b, f\}$  of arity 0,0,2. The number of distinct nodes labeled by  $a$  can be 1 at most. When the number of distinct nodes labeled by  $f$  is greater than  $k$ , we look for a beneficial cycle with counters. If there is no such cycle, we count the number of  $\begin{matrix} f \\ \swarrow \searrow \\ a \quad f \end{matrix}$ ,  $\begin{matrix} f \\ \swarrow \searrow \\ b \quad f \end{matrix}$ ,  $\begin{matrix} f \\ \swarrow \searrow \\ f \quad a \end{matrix}$ ,  $\begin{matrix} f \\ \swarrow \searrow \\ f \quad b \end{matrix}$  and  $\begin{matrix} f \\ \swarrow \searrow \\ f \quad f \end{matrix}$ . If one such label appears many times, then we can divide it furtherly.  $\diamond$

### 7.3.2 Finding or Building a New Cycle

When we decide that there might be a beneficial cycle with counters in the tree schema, we have a label (or sequence of labels)  $f$  that appears many times. We need only compare nodes labeled by  $f$ .

The first step is the refinement of the information on the label. This step is necessary if there is no interesting cycle, and it is useful as a first decision procedure: if there is no subdivision of the label that appears often enough, then we do not have to go further, and if there is such a subdivision, then we can restrict the nodes we consider to nodes labeled by this subdivision.

When we have our set of candidate nodes, we can compare them, in the way of chapter 3, but with a little more leniency, because we just look for a finite cycle, not an infinite one. So, when the number of pair of nodes labeled by  $f$  visited during the comparison is big enough, we allow a difference between one node labeled by  $f$  and another node. This node is the end of the cycle, the outgoing path. So, at this point, we add a choice node (or increase the existing one) and link the loop to the choice node, in the way described in the previous section.

During the comparison of the candidate nodes, we can use the information on the number of nodes labeled by a given label  $g$ . If, in the comparison, we come to such a label and the number of nodes labeled by  $g$  is too little, then we can stop the comparison.

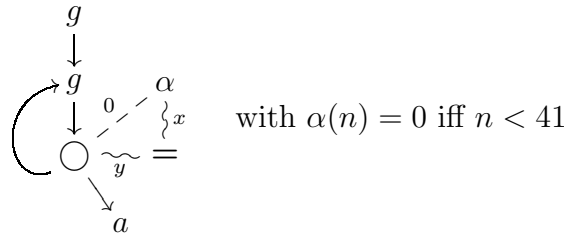
When the size of the sequence of labels becomes too big, the computation of the number of nodes labeled by such labels might become too costly. Then, we can approximate the tree schema so that we force the apparition of a finite cycle. The

advantage of this approximation compared to the approximation by an infinite cycle is that the approximation by finite cycle is more precise. Moreover, we can still have some information on the number of times we go through the cycle, whether it is increasing, to decide whether the approximation was appropriate.

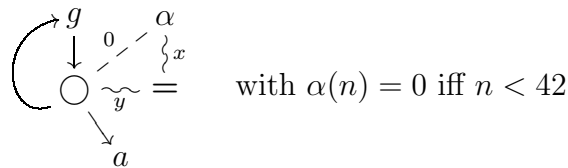
### 7.3.3 Maximal Folding

When we have a cycle linked to a choice node counter, such as the ones that can be automatically discovered, we want to fold it as much as possible. Because of its finitary nature, we cannot use the same algorithm as for infinite cycles. We show an example where there is a folding which is not resolved by the algorithms of chapter 3:

**Example:**



is the same as



◇

This problem is easily solved by modifying slightly the keys associated with the cycle in the sharing algorithms. The normal key for the choice node that is linked to the counter is replaced by its father, and if we build such a node, we increase the value of the number of times the cycle is to be followed (in the counter function).

## 7.4 Modified Algorithms

In order to use tree schemata with counters, we need to modify the algorithms on tree schemata. Thanks to the separation between skeletons and relations, and to the locality of links, the modifications are very light. We will describe them referring to the algorithms of chapter 6. The only interactions we consider in detail in this section are the influence of the counters over the decision process (in the relations, through counter function), and the choice node counters.

### 7.4.1 Global Information and Forgetting

The only information we need to add to the global informations used when going through the tree schemata is the value of the counters. With this simple extension, and by taking into account the value of the counter functions each time we perform a  $G(l)_{(x)}$  or a  $G_{x=v}^l$ , we can keep the constructive definition of section 5.3.6 (page 99), and we have the new definition of the set represented by a tree schema with counters. The initial global information extends to counter values, as the current external values, if any, and for choice node counters, the initial value is 0. Choice node counters are incremented by one each time we go through the appropriate child of the appropriate choice node.

When we want to compute local informations, we need to forget the global information, after taking it into account in the links. It is the case for example at the end of the computation of the projection of tree schemata. It is also a necessary step before computing the inclusion or the intersection of the relations, so that the counter functions of the two tree schemata be in phase.

When we want to get rid of the global information (forget it), we just have to change the counter functions according to the current value of the choice node counters. Because each time we start a new evaluation the value of choice node counters is 0, we just replace every counter function  $\alpha$  depending on the choice node counter  $n$  by  $\alpha'(n) = \alpha(n + k)$ , where  $k$  is the value of the counter in the global information.

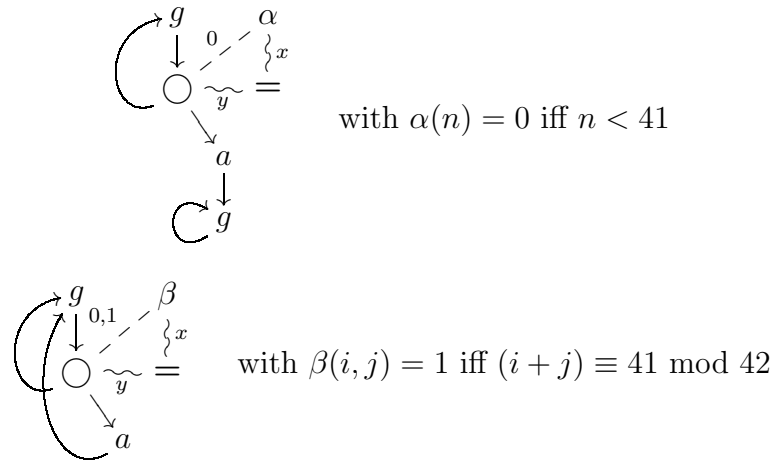
### 7.4.2 Inclusion and Loops

Tree schemata inclusion testing consists in skeletons and finite parts of the relation inclusion testing, then infinite relations testing. With counted loops, we can have very big finite loops, and it would be too costly to unfold entirely those loops. So, the inclusion will go as for the simple tree schemata, but we will also compute the inclusion of finite loops with the infinite relations. To isolate the finite loops, we do as for infinite relations, we stop the inclusion testing of the skeletons as soon as we loop. The scope of the choice node counter is the root of its cycle.

The idea is that instead of following the loops, we jump directly to the values of the counters that change the values of the counter functions. We keep the pairs of values in memory, and we iterate until no other change in the value of the counter functions is possible by increasing the counters, or the only possible pairs of value have already been encountered.

Each time we make a jump with a counter, we compute the new counter functions, as described above, and we compare the choice nodes that are linked to these counter functions with the appropriate node of the other tree schema.

**Example:** We compare the following tree schemata:



On the first comparison of the choice nodes,  $n = 0, i = 0, j = 0$ , so  $\alpha = 0$  and  $\beta = 0$ , so we just compare the left child. This comparison holds true. Then we jump to  $n = 41, i = 41$ . This time, we just go to the right child, and in the loop, we have  $i = 41, j = 1$ , so we compare again  $\overset{\curvearrowright}{C}g$  with the left child of the choice node of the second tree schema. This time again, the comparison holds true. Finally, we jump to  $i = 82$ , and we compare the right child. This time, we can conclude that the first tree schema is not included in the second one.  $\diamond$

Concerning the inclusion testing of the infinite behavior, it can be performed when we have tested the finite loops. It is not modified by the counters.

### 7.4.3 Intersection and Counter Functions Merging

The adaptation of the intersection algorithm on tree schemata with counters shows all the problems that are raised by any other fundamental algorithms on tree schemata (union, projection...). We show how to solve these problems for intersection, but the solutions are the same for the other algorithms.

To compute the intersection of two tree schemata with counters, we need to compute the new relations, between new counter functions, if any. Indeed, before computing the intersection of the relations, we need to merge the counter functions.

#### Adaptation to the New Skeleton

The first step is the adaptation of the counters and counter functions to the new skeleton (the intersection of the skeletons of the tree schemata). In this new skeleton, cycles may be unfolded, either by root unfolding or cycle growth. In case of root unfolding of a counted cycle, we count the number  $k$  of such unfoldings, and we replace  $\alpha$  depending on the counter  $n$  by  $\alpha'(n) = \alpha(n + k)$  (as for global

information forgetting). In case of cycle growth, we count the number of cycle growth and divide the counter between every inflated choice node. If the new counters are named  $n_0, \dots, n_i$ ,  $\alpha'(n_0, \dots, n_i) = \alpha(n_0 + \dots + n_i)$ .

### Counters Comparison

The second step consists in comparing the counters of the two tree schemata. As for regular relation entries, we need to recognize when two counters count the same loop. We can get rid of one counter name if and only if two choice nodes are in the same loop (there is a path from one to the other and back to the first one). Let  $N$  and  $M$  be two such choice nodes. We call  $i_0, \dots, i_k$  the counters of the choices of  $N$  that lead to  $M$ , and  $j_0, \dots, j_n$  the counters of the choices of  $M$  that lead to  $N$ . We have  $\sum_{a=0}^k i_a = \sum_{a=0}^n j_a$ . This is a necessary step to compare the counter functions.

### Counter Functions Comparison

The third step is the comparison of the counter functions. Only counter functions which share some counter names need to be merged. In this case, we compute the new counter function as the cartesian product of the original ones. If  $\alpha$  is the counter function of the first tree schema, and  $\beta$  the counter function of the second tree schema. We call  $p$  the common counters of  $\alpha$  and  $\beta$ ,  $n$  the other counters of  $\alpha$  and  $m$  the other counters of  $\beta$ . The merging of  $\alpha$  and  $\beta$ ,  $\gamma$  is defined as  $\gamma(n, p, m) = \langle \alpha(n, p), \beta(p, m) \rangle$ . The extension of the relations to these new values is obvious, we just consider in the decision diagram the appropriate projection of the value: if  $R$  is the relation of the first schema, the new relation is defined as  $R'(\langle a, b \rangle) = R(a)$ .

### Verifying the New Tree Schema

Then we can compute the intersection of the relations. At this stage, we have a tree schema representing the intersection, but we need to perform some verifications to transform the tree schema into a valid tree schema. This is performed for links (as was described in chapter 6), as well as for counter functions or external counters.

Concerning external counters, we just need to verify that the relation linked to the counter is full. If it is not the case, then we must propagate the information to the least choice node  $N$  such that every choice node in the link is a strict subtree of  $N$ . If there is no such choice node, it means that for the values on which the relation is false, the tree schema is empty. So we remove these values from the mapping function. If there is a least such choice node  $N$ , then we add  $N$  to the link, forbidding the choice of  $N$  leading to the original link for every value on which the relation is false.

Concerning counter functions, we can change the way the counters are computed. First, we simplify the counter functions by going through the new tree schema with global information in the way described in the previous subsection. Such a traversal might suppress some parts of the skeleton. This suppressions must be propagated, as for relations leading to an empty choice. If, after the counter functions simplification we still have some values of the counter functions on which the relation is false, then we add every choice node where this counters are computed in the relation, forbidding those values. If the entry of the counter function does not appear in the decision diagram, then we can get rid of the counter function (and all the associated counters).

## 7.5 Conclusion

The extensions presented in this chapter showed another interest of the separation between the skeleton part and the relational part of the set of trees. Thanks to this separation, the addition of external counters was easy. We also added choice node counters. Such counters can be automatically created during the computation of tree schemata. The use of the counters in the tree schemata increases the complexity of the algorithms on tree schemata. They are intended as a way to perform smart approximations by numerical analysis of the counters. They offer another tradeoff opportunity between precision and complexity.





# Chapter 8

## Tree Based Abstract Interpretations

In order to illustrate the possible usefulness of tree schemata, we briefly describe different examples of abstract interpretations using sets of trees in their domains. We compare the existing solutions to tree schemata and show the advantages of our solution.

### 8.1 Model of Logic Programs

Abstract interpretation have been widely applied to logic programs. A general presentation of the application of abstract interpretation of logic programs and a survey of the different applications can be found in [CC92a].

The analysis we study here is the computation of the Herbrand model of logic programs.

#### 8.1.1 Syntax and Collecting Semantics

Let us first define the syntax of logic programs. We suppose that we have two finite ranked sets of labels,  $F$  and  $G$ , and a set of variables,  $V$ . A term is a tree labeled by  $F \cup V$ . An atom is a tree labeled by  $F \cup G \cup V$ , where the only subtree labeled by  $G$  is the root. We write an atom  $\overset{p}{\vee} \underset{t_0 \dots t_{n-1}}{\simeq}$ , where  $p \in G$ , and each  $t_i$  is a term. A clause is a couple (head, body), such that the head is an atom and the body is a finite sequence of atoms. We write  $a \leftarrow a_0 \dots a_{n-1}$ . A logic program is a finite sequence of clauses.

A ground instance of a term  $t$  is a tree  $u$  such that there is a substitution  $\kappa : V \rightarrow \mathcal{H}(F)$  and  $u$  is the tree substitution of each variable  $X$  by  $\kappa(X)$  in  $t$ . A ground instance of a clause  $a \leftarrow a_0 \dots a_{n-1}$  is a clause  $b \leftarrow b_0 \dots b_{n-1}$  such that there is a substitution  $\kappa$  and  $b$  is the tree substitution defined by  $\kappa$  on  $a$  and each

$b_i$  is the tree substitution defined by the same  $\kappa$  on  $a_i$ . A logic program  $P$  defines a set of ground instances of its clauses. We denote this set  $ground(P)$ .

The standard semantics we chose for logic programs is a small step operational semantics based on SLD resolution. The collecting semantics that we choose to study is the backward semantics based on the  $T_P$  operator of [vEK76]:

$$T_P(E) = \{ a \mid \exists a \leftarrow a_0 \dots a_{n-1} \in ground(P) \text{ such that each } a_i \in E \}$$

A *model* of the program  $P$  is a fixpoint of  $T_P$ . Generally, one uses the least model of the program  $P$ . But the greatest model may be considered too: it is used for example to manipulate infinite trees, as in [Col82, Col84], or to specify infinite state systems in [CP98b].

### 8.1.2 Approximation of the Model

The semantic domain of the collecting semantics is  $\mathcal{P}^b = \wp(\mathcal{H}(F \cup G))$ . The abstract domain we use is the set  $\mathcal{P}^\# = \mathcal{TS}^\cdot(F \cup G)$ . The approximation ordering is the inclusion ordering.

The  $T_P$  operator of a logic program  $P$  can be defined using a meta expression in the way of section 5.4.2. If  $P$  contains only one clause  $a \leftarrow a_1 \dots a_n$ , then  $T_P(E) = \{ \{ a : a_1 \in \mathcal{X}, \dots a_n \in \mathcal{X} \} \mid \rho \text{ where } \rho(\mathcal{X}) = E \}$ . If  $P$  contains more than one clause, then the  $T_P$  operator corresponds to the union of the meta expressions associated with every clause. Thus, if  $E$  is represented by a tree schema, we can compute  $T_P(E)$  and represent it with a tree schema.

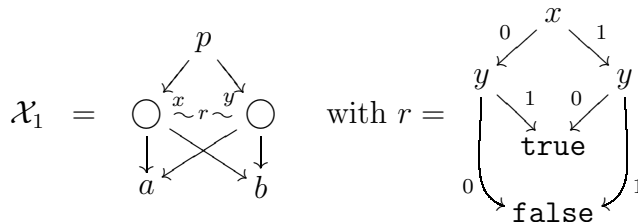
**Example:** Let  $P$  be the following logic program:

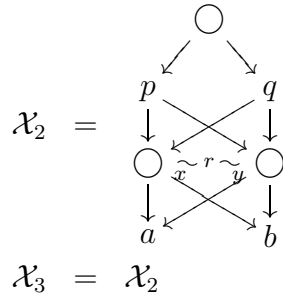
$$\begin{aligned} p(a, b) & \quad . \\ p(b, a) & \quad . \\ q(X, Y) & \leftarrow p(X, Y) \end{aligned}$$

The meta expression representing  $T_P$  is  $\left\{ \begin{smallmatrix} p \\ \swarrow \searrow \\ a \quad b \end{smallmatrix} \right\} \cup \left\{ \begin{smallmatrix} p \\ \swarrow \searrow \\ b \quad a \end{smallmatrix} \right\} \cup \left\{ \begin{smallmatrix} q \\ \swarrow \searrow \\ X \quad Y \end{smallmatrix} : \begin{smallmatrix} p \\ \swarrow \searrow \\ X \quad Y \end{smallmatrix} \in \mathcal{X} \right\}$ .

The iteration sequence computing the least fixpoint of the  $T_P$  operator will be:

$$\mathcal{X}_0 = \bigcirc$$





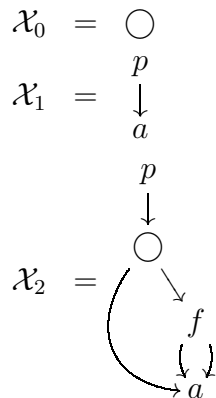
◇

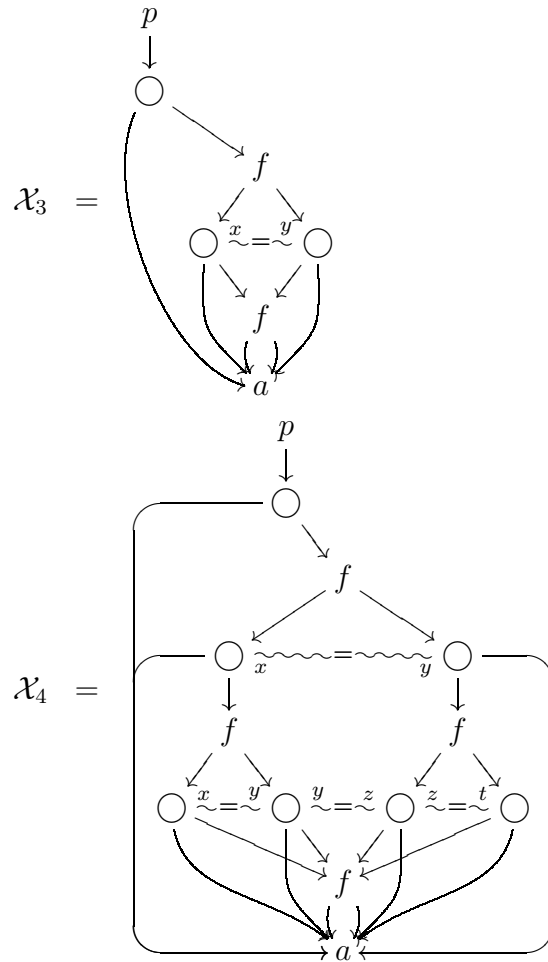
As long as the iteration sequence is finite, the fixpoint is represented by a tree schema. When the iteration sequence is infinite, though, it is possible that the fixpoint cannot be represented by a tree schema. In this case, anyway, we need some way of acceleration of the iteration sequence, and the usual way in abstract interpretation is the use of an approximation through widening. We can use a widening using the choice node counters of section 7 which will fold the finite cycles as soon as the counters are above a given constant. If we stop the label numbering at a given depth, then the total size of the tree schema is bounded. With the bound on the counters, it gives a finite set of tree schemata, thus ensuring termination. Note also that if the meta expression have the limited form of section 5.4.2, we can compute directly the fixpoint without performing the iteration sequence.

**Example:**  $P$  is the following program:

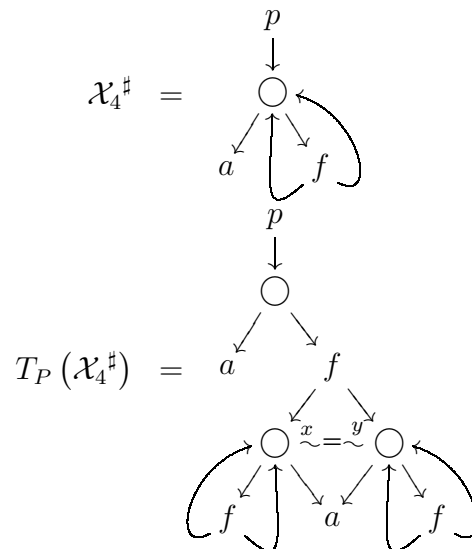
$$\begin{array}{l} p(f(X,X)) \leftarrow p(X) \\ p(a) \quad . \end{array}$$

The beginning of the iteration sequence for the least fixpoint is:





It is obvious that a tree schema representing a fixpoint of  $T_p$  would require an infinite number of entry names. We have the following possible approximation for  $\mathcal{X}_4$ :



We have  $T_P(\mathcal{X}_4^\sharp)$  is included in  $\mathcal{X}_4^\sharp$ , so  $\mathcal{X}_4^\sharp$  is a post fixpoint of  $T_P$ . Indeed, we can take  $T_P(\mathcal{X}_4^\sharp)$ , which is also a post fixpoint and is a more precise approximation of the least fixpoint of  $T_P$ .  $\diamond$

### 8.1.3 Comparison with Other Approaches

Earlier works concerning the approximation of the least Herbrand model of logic programs were known under the name of type inference of logic programs (see for example [Mis84, MR85, YS87]). These algorithms were not complete and the results were very approximate due to the use of the cartesian closure approximation.

In [HJ90], Nevin Heintze and Joxan Jaffar proposed an improvement on type inference using set based analysis: their analysis was more precise and the inferred approximation was recursive. But due to cartesian approximation and the rigid framework they used, they could not get the exact result on the first example. Moreover, their representation uses many variable names, and in particular introduces an exponential number of variable names during the computation of the intersection. Thus, the representation was very inefficient. In fact, their representation, and the use of cartesian approximation, could be as expressive as non deterministic tree automata. It means that by tuning carefully the analysis (for example using abstract interpretation techniques, as suggested in [CC95]), one could get the exact meaning of the first example with the same approximation.

Abstract interpretation have been used to compute an approximation of the least Herbrand model (see for example [HK87, BJ88, FS91]). These approaches have been limited by the available abstract domains. Thus their answer on the second example could not be as precise as with tree schemata.

## 8.2 Sets of Traces

The operational semantics of programs associates with a given program a set of states and a transition relation over this set of states. For a given set of inputs, the interpretation is a trace, starting from an initial state defined by the inputs and following the relation from state to state until no other transition is possible (or until we loop on a final state, if we require that the relation is total). Thus, an alternative representation for the operational semantics of the program is a set of traces. The exact shape of the traces depend on the particular semantics, but we will see that they can always be represented by (possibly infinite) trees. So, they are a good field of application for abstract interpretation based on trees.

Abstract interpretation of the execution traces was one of the initial applications of abstract interpretation. David Schmidt gave a recent presentation on the

abstraction of a given trace by an abstract computation tree in [Sch98b]. Many examples in this section are taken from his article.

### 8.2.1 Flowchart Semantics

#### Presentation

Flowcharts are one of the oldest representations of the semantics of imperative languages. A flowchart is a graph labeled by program instructions. Each node in the graph corresponds to a program point. There are two special nodes named entry and exit. The nodes have one outgoing edge, except for the exit node which has none and test nodes which have two outgoing nodes, one labeled by **true**, and the other one labeled by **false**.

A state of the program is a couple  $\langle pp, \rho \rangle$ , where  $pp$  is a program point and  $\rho$  is an environment binding the different variables of the program to their current values. The flowchart is a representation of the transition relation of the program: each states  $\langle pp, \rho \rangle \rightarrow \langle pp', \rho' \rangle$  if there is an edge from  $pp$  to  $pp'$  in the flowchart, and if  $pp$  is a test point, then the edge is labeled by the evaluation of the test in the environment  $\rho$ . The new environment  $\rho'$  is computed by the instruction in the node of  $pp$ .

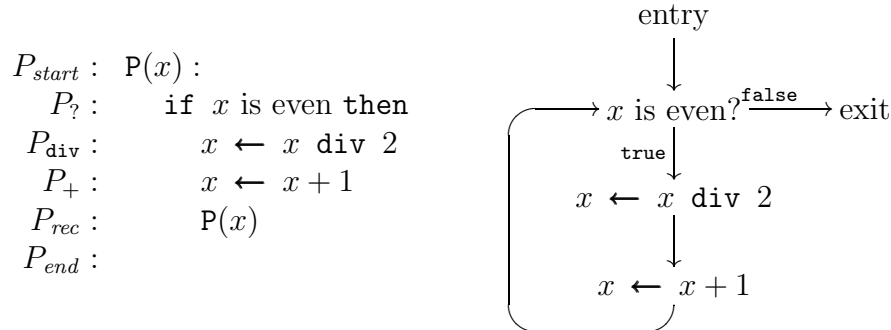
#### Flowchart Semantics as Sets of Trees

It is obvious from the semantics that any state can be followed by at most one state. It means that the execution trace of a program based on a flowchart semantics is a (possibly infinite) word of states. Thus, the semantics of the program can be equivalently represented by a set of words, or trees with labels of arity at most one. This representation can be understood as a first computation on the flowchart that loses no information, and that is convenient to express properties of the program, such as dataflow properties or path properties.

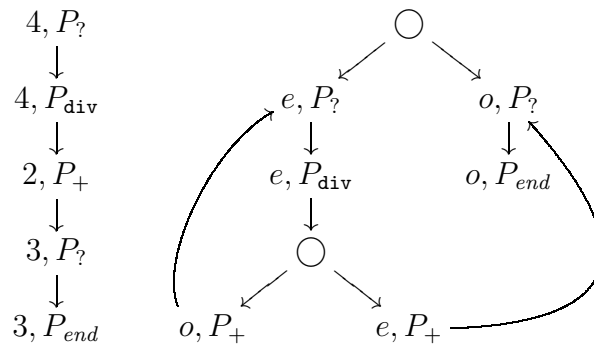
The problem is that the set of states of a program is generally infinite, and the set of traces is very difficult to represent in an explicit way (as opposed to the flowchart, which is an implicit representation). One way of representing sets of trees labeled by an infinite set of labels  $F$  with tree schemata is the use of a finite set of labels  $F^\sharp$ , each new label abstracting elements of  $\wp(F)$ , such that  $F^\sharp$  represents a partition of  $F$ .

In this way it is possible to have a tree schema representing a supset of the set of traces. When the properties collected from the representation of the standard semantics are dataflow properties, this approach corresponds to direct abstract interpretation with finite abstract domain.

**Example:** Consider the following (labeled) imperative program and its flow-chart semantics:



We show a possible trace with entry  $x = 4$  and a supset of the set of traces represented by a tree schema using the abstract domain  $\{o, e\}$ , where  $o$  represents the set of odd natural numbers and  $e$  the set of even natural number.



◇

In [Sch98b], a given trace is approximated by a tree, which represents a set of traces. The advantage of our point of view is that there is no need for a complex safety relation: the tree schema is a safe approximation if and only if it represents a supset of the set of traces. The use of choice nodes allows the possibility of specifying the infinite behavior of the program in the tree schema: if the program were known to terminate, then we can add a relation on the second choice node that excludes infinite loops.

Although the approximation by tree schemata allows better flexibility than the classical behavior tree, the traces of flowchart semantics are mere words. It means that the full complexity of tree schemata is not exploited and that we could use a more efficient more specialized representation. So we present also a semantics where the traces themselves are trees.

## 8.2.2 Big-Step Semantics

Big-step semantics represent the transition relation of programs in terms of inference rule schemes. Big-step semantics are used to describe the semantics of

higher order languages, or imperative languages with recursive interprocedural calls. As above, a state of the program is a couple  $\langle pp, \rho \rangle$  of program points and environments. The inference rule schemes represent possible transitions and compute return values. An inference rule is composed of an inferred sequent  $s$  and a finite sequence of required sequents  $s_1, \dots, s_n$ . We write  $\frac{s_1, \dots, s_n}{s}$ . A sequent is a couple state-value. The transition relation derived from the set of inference rule schemes is: a state  $S$  is derived to  $S'$  if there is a rule  $\frac{s_1, \dots, s_n}{S, v}$  and there is an  $i$  such that  $s_i = S', v'$ , the transitions before  $i$  return a value compatible with the  $s_j$ , and  $S'$  returns  $v'$  and then the transition after  $i$  return a value compatible with the  $s_j$ . Then the value computed by  $S$  is  $v$ . The execution trace of a program with initial value is the tree corresponding to all the necessary steps used to compute the return value.

We can still approximate the set of traces by a tree schema representing a supset of the set of traces. Again, we use an approximation of the set of states.

**Example:** We present a small program written in a small functional language containing conditional, recursive definitions and primitive operators on expressions. The primitive operators will be denoted  $op$ , the expressions  $e$  and the identifiers  $x, y, f, g$ . Environments map identifiers to values, which can be closures. Program points are identified to subexpressions. The inference rules are:

$$\frac{e, \rho, v}{op(e), \rho, op(v)} \quad \frac{e_1, \rho, \mathbf{true} \quad e_2, \rho, v}{\mathbf{if} \ e_1 \mathbf{then} \ e_2 \mathbf{else} \ e_3, \rho, v} \quad \frac{e_1, \rho, \mathbf{false} \quad e_3, \rho, v}{\mathbf{if} \ e_1 \mathbf{then} \ e_2 \mathbf{else} \ e_3, \rho, v}$$

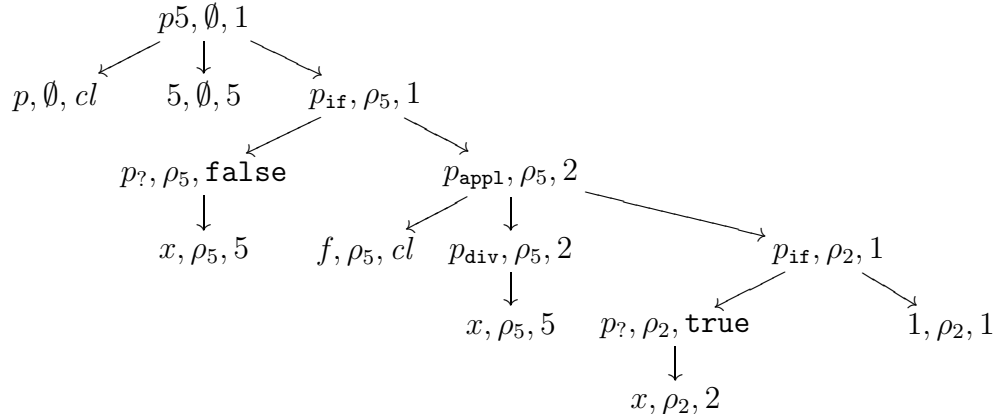
$$\frac{\mathbf{rec} f(x).e, \rho, \langle \rho, f, x, e \rangle \quad x, \rho, \rho(x)}{e_1, \rho, \langle \rho', f', x', e' \rangle \quad e_2, \rho, v' \quad e', \rho' \cup \{f' \rightarrow \langle \rho', f', x', e' \rangle, x' \rightarrow v' \}, v}$$

$$e_1 e_2, \rho, v$$

Here is the small program we analyze, with the labels corresponding to its subexpressions:

$p = \mathbf{rec} f(x). \mathbf{if} \ \mathbf{even}(x) \ \mathbf{then} \ 1 \ \mathbf{else} \ f(x \ \mathbf{div}2)$   
 $p_{\mathbf{if}} = \mathbf{if} \ p? \ \mathbf{then} \ 1 \ \mathbf{else} \ p_{\mathbf{appl}} \quad \mathbf{and} \ p_{\mathbf{appl}} = f(p_{\mathbf{div}})$

The following tree is the trace of  $p5$  (i.e.  $p$  applied to 5):

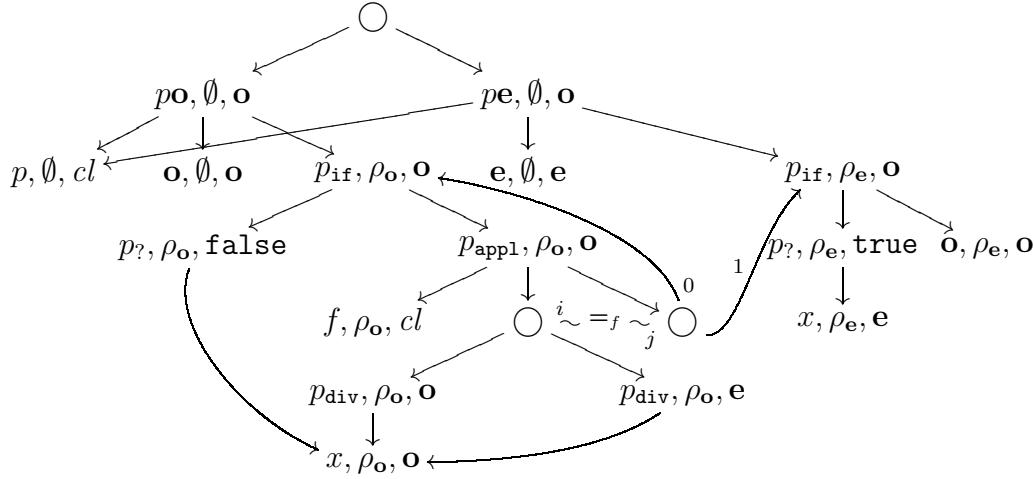




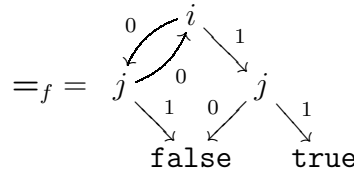
$$\text{where } cl = \langle \emptyset, f, x, p_{if} \rangle$$

$$\rho_i = \{f \rightarrow cl, x \rightarrow i\}$$

The abstraction of the set of traces of  $p$  applied to an integer is:



Where the relation  $=_f$  allows only finite loops:



◇

### What is Gained with Tree Schemata

The approximation of the set of traces by tree schemata present many advantages, compared with the behavior tree: there is a clear distinction between necessary derivations and non-deterministic choices represented by  $\bigcirc$ . This distinction allows better property extraction. The expressive power is better with tree schemata, even compared with the so-called set-based abstraction presented in [Sch98b]. The expressiveness is increased concerning the finite behavior when dealing with trees and not only words. Concerning the infinite behavior, the expressiveness is increased, even compared to the use of positive and negative inference rules [IS98] in the behavior tree.

### 8.2.3 Parallel Programs: Fairness

Parallel programs semantics can be expressed using any of the previous semantics, with the additional complexity that for a given program and a given initial

state, we can derive many traces. Each trace corresponds to a possible order of execution of the different parallel parts of the program. Such a semantics is called the interleaving semantics [SNW94, vGG89] of the program.

We show a very simple example of parallel program and its interleaving semantics to illustrate the expressive power of tree schemata with  $\omega$ -deterministic relations.

**Example:** The following program sets the variable  $x$  to **true**, then executes in parallel the loop **while**  $x$  **do** **skip** and sets the variable  $x$  to **false**.

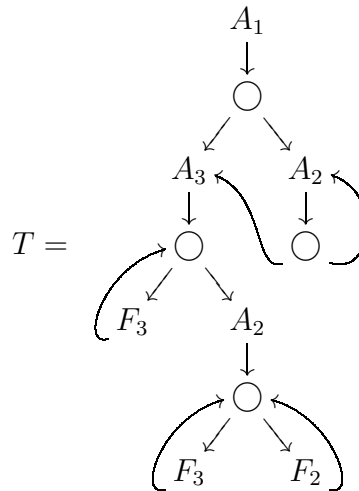
$$X = \text{true}; \parallel \text{while } X \text{ do skip} \parallel X = \text{false} \parallel$$

The positions in the program are:

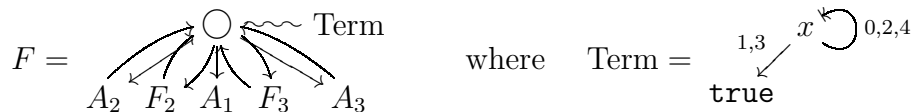
$$A_1 \parallel A_2 \parallel A_3 \parallel$$

In order to take into account the possibility that a parallel part of the program can be terminated and the other parallel parts are stalled, we introduce the positions  $F_2$  and  $F_3$  and we allow  $F_2$  to be followed by  $F_2$ . In this way the traces of the program are necessarily infinite. The program is said to terminate if after some point, we just have the states  $F_2$  and  $F_3$ .

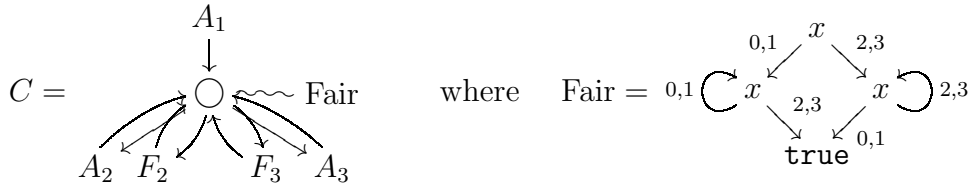
Under these assumptions, the set of traces of the program is:



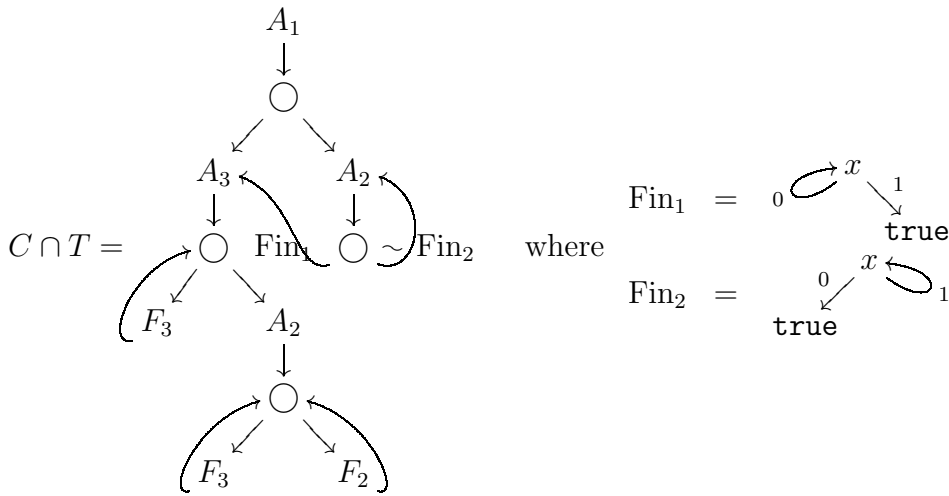
We can express the termination property with a tree schema. The following tree schema contains all terminating traces of programs with positions  $A_1, A_2, A_3, F_2$  and  $F_3$ :



So, to prove termination, we have to prove  $T \subset F$ . We see that here, with no additional assumption, it is not the case<sup>1</sup>. We will show that under a weak fairness assumption, the program does always terminate. The proof is very easy. The following tree schema expresses the weak fairness property:



Then the semantics of the program under the fairness assumption is:



And we have  $C \cap T \subset F$ . ◇

This example shows how easy it is to manipulate non-trivial infinite behaviors with tree schemata.

## 8.3 Other Applications

Many other abstract interpretations or analyzes may benefit from the use of tree schemata. We present some of them very briefly.

### 8.3.1 Model Checking

Model checking [CES83] is a technique used to verify properties over systems, such as hardware or software systems. When the system is finite, model checking

<sup>1</sup>In general,  $T$  is a supset of the set of traces, so we cannot conclude that the program does not terminate. In order to do this, we can use subsets of the set of traces.

uses compact symbolic representations, such as BDDs. When the system is too big or infinite, model checking may benefit from abstract interpretation to reduce the size of the systems to be explored. This seems to be a growing direction of investigation in model checking (see for example [Bru93, CGL94, Dam96, Lev97, Sch98a]).

Properties of systems, especially temporal properties, tend to be expressed over sets of trees, using logics such as CTL\* [Tho88] or the modal  $\mu$ -calculus. Models of systems using such tree representation would greatly benefit from the symbolic representation using tree schemata. Because of their ability to express complex infinite behavior, such as fairness, they would be well suited for these logics. When analyzing programs, one could use the set of traces as defined in the previous section, instead of the computation tree. In [ES88], it is noted that the representation of the model as a set of traces instead of a set of states and a relation is more general. In the more recent [KMM<sup>+</sup>97], the adequacy of the use of sets of trees to model parameterized systems is demonstrated.

### 8.3.2 Closure Analysis

In the set of states corresponding to a higher order functional program, we have closures representing higher order functions. Abstracting the set of states so that we can represent these closures during an analysis is a difficult problem. In a unified presentation of the problem, [JW95] presented a representation based on flow graphs. Their presentation defines a hierarchy of instantiations of their framework of growing accuracy and complexity (the simplest one, OCFA is equivalent to set-based analysis). In this framework, the tree schemata can be used as an alternative representation for the flow graph, as in the set of traces case.

# Chapter 9

## Conclusion

### 9.1 Main Results

The main results of this thesis are a new representation for sets of trees and a new representation for infinite relations.

The new representation for infinite relations is based on decision diagrams and is an extension of BDDs. It defines a new class of infinite relations,  $\omega$ -deterministic relations, which can be used as efficient approximations of regular relations. Their expressiveness allows the representation of safety, liveness and fairness properties (and many more).

The new representation for sets of trees is based on the decomposition of this sets between a tree-shaped structural information and a relational information. The new class of sets of trees it defines encompasses infinite tree automata and tree automata with equality constraints between brothers. The representation enforces maximal sharing of prefixes and the sharing of isomorphic sets of subtrees.

It allows the emptiness decision in constant time. Inclusion testing is decidable. If the relations are closed under intersection, the intersection of two tree schemata is a tree schema. If the relations are moreover closed under union, then the union of two tree schemata is also a tree schema. If we have approximation strategies over relations, they carry over to tree schemata to compute approximations of intersection or union for example. With the same restriction, we can compute projections and tree constructions on tree schemata.

Because of the decomposition at the heart of tree schemata, useful extensions such as loop counting can easily be added. Thanks to their expressive power and to the way they can handle approximations, tree schemata seem very well suited for many tree based abstract interpretations.

## 9.2 Future Work

### 9.2.1 Practical Usefulness

The main goal of the tree schemata is to allow practical and precise abstract interpretations based on sets of trees. To this effect, we presented many algorithms. Some of them were tested on prototype implementations in Java, but they depend on many parameters that need to be tested with many experiments in realistic conditions. Among this parameters, we can choose the representation of relations, the best ways of approximating iteration sequences and the representation of choice nodes. Moreover, the algorithms presented in the thesis give a flavor of the way we can manage tree schemata, but some of them can certainly be improved.

### 9.2.2 Negation

In this thesis, we did not mention negation, although this operation might be very useful, in particular to be used in model checking.  $\omega$ -regular relations are closed by negation, but  $\omega$ -deterministic relations are not. It is possible to approximate the negation of  $\omega$ -deterministic relations, but it means that we need to carry over a supset and a subset approximation.

Negation is a regular process, that is, if its input is regular, the process will compute only a finite number of states. So we believe that tree schemata based on  $\omega$ -regular relations can be closed by negation. Again, with less expressive (and more practical) relations, we will need to carry over two approximations. It could be feasible, and it would be interesting to have an idea of the complexity of such an operation. It is possible that the addition of a specific negation node reduces the complexity, but we need to be very careful that such a node does not tamper too much with the sharing.

### 9.2.3 Extensions

In chapter 7, we presented some quite general extensions that increased considerably the expressive power of tree schemata. These extension need to be furtherly explored before we can state their usefulness.

# Appendix A

## List of Symbols and Conventions

### A.1 Widely Used Mathematical Symbols

$\mathbb{N}$	the set of natural numbers
$\wp(E)$	the set of subsets of $E$
$ E $	the cardinal of the set $E$
$\Leftrightarrow$	if and only if
$\Rightarrow$	implies
$dom(f)$	the domain of the partial function $f$
$im(f)$	the image of the function $f$
$f : E \rightarrow F$	the domain of the function $f$ is $E$ and its codomain is in $F$
$\bigotimes_{i \in I} E_i$	the cartesian product of the family of sets $(E_i)_{i \in I}$
$\bigcup_{i \in I} E_i$	the union of the family of sets $(E_i)_{i \in I}$
$\bigcap_{i \in I} E_i$	the intersection of the family of sets $(E_i)_{i \in I}$
$E \setminus F$	the set of elements of $E$ without the elements of $F$

### A.2 Conventions for Other Mathematical Symbols

$\text{lub}_{\leq}(E)$	the least upper bound of $E$ for the partial ordering $\leq$
$\text{glb}_{\leq}(E)$	the greatest lower bound of $E$ for the partial ordering $\leq$
$[n]$	the set of natural numbers smaller than $n$
$[n, m[$	the set of natural numbers between $n$ and $m$ , $m$ excluded
$ u $	the size of the word $u$
$\varepsilon$	the empty word

$\bigodot_{i \in I}^{<_I} u_i$	the concatenation of the finite words $u_i$ in the total ordering defined by $<_I$ (page 6)
$A^*$	the set of finite words over $A$
$A^\omega$	the set of infinite words over $A$
$A^\infty$	the set of finite and infinite words over $A$
$\prec$	the prefix relation on words
$\prec_A^*$	the alphabetic ordering on $A^*$
$\overrightarrow{L}$	the closure in the set of infinite words of the set of finite words $L$ , with respect to the prefix ordering (page 7)
$L^\omega(\mathcal{A})$	the set of infinite words recognized by the Büchi automaton $\mathcal{A}$
$L^*(\mathcal{A})$	the set of finite words recognized by the automaton $\mathcal{A}$
$pos(t)$	the domain of the tree $t$
$Ar_L$	the arity function associated with a set of labels $L$
$\mathcal{H}(L)$	the set of trees labeled by $L$ and respecting its arity
$t_{[p]}$	the subtree of $t$ in $p$ , where $p$ is a path of $t$
$s \prec t$	$s$ is a subtree of $t$
$t[u \setminus v]$	substitution of the subtree $u$ of $t$ by $v$
$Subtrees(t)$	the set of subtrees of $t$
$G^N$	the set of nodes of the graph $G$
$G^E$	the set of edges of the graph $G$
$\emptyset^G$	the empty graph
$\langle \rangle$	the element of $\bigotimes_{i \in \emptyset}$
$Support(R)$	the support set of the relation $R$ (page 13)
$x_{(j)}$	the projection of $x$ (an element or a set) according to $j$ (an index, a set of indexes or an index name)
$R_{:J=e}$	the projection of $R$ according to $e$ (page 14)

### A.3 Sets and Operators Defined in the Thesis

$\equiv_t$	The equivalence between two graph nodes, when they represent the same tree.
------------	---

#### A.3.1 Relations

$R = R_1 \bar{\wedge} R_2$	the decomposition of the relation $R$ in two independent relations (page 14)
----------------------------	--



$\text{AllProj}(R)$	the set of all possible projections of the relation $R$ according to any set of values (page 14)
$e_{i \leftrightarrow j}$	the vector $e$ with its $i^{\text{th}}$ and its $j^{\text{th}}$ components exchanged
$\mathcal{R}el$	The set of all possible relations with entry names and sets of indexes subset of $\mathbb{N}^*$ (page 44)
$\text{word}(e)$	the word equivalent to the vector $e$
$\langle u \rangle$	the vector equivalent to the word $u$
$\text{name}_R(e)$	the name associated with the entry $e$ of the relation $R$
$\text{ename}(R)$	the set of entry names of the relation $R$
$R_1 \simeq R_2$	the relations $R_1$ and $R_2$ are isomorphic (page 46)
$\Omega(R)$	the recursive relation defined by the open relation $R$ . If $R$ is not open, then the empty relation (page 48)
$\square(R, J, w)$	is the property stating whether the relation $R$ projected according to $w$ on the set of indexes $J$ is “square”. That is, if $R$ is a relation on $\bigotimes_{i \in I} E_i$ , $\square(R, J, w)$ is true if $R_{:J=w} = \bigotimes_{i \in I \setminus J} E_i$ .
$\text{nameiter}(R, k)$	is the property stating whether $k$ is a period of the entry names of $R$ . It is true if $\forall i \in \mathbb{N}$ , $\text{name}_R(i + k) = \text{name}_R(i)$ .
$\text{iter}(R)$	the family of recursive relations associated with the regular relation $R$ (page 48)
$\text{infbehave}_R(p)$	the set of indexes in the family of recursive relations $(R_i)_{i \in C}$ associated with the regular relation $R$ such that whatever $i$ in this set of indexes, whatever $\langle f \rangle \in R_i$ , $\langle pf \rangle \in R$ (page 59)
$R_\omega(t)$	the $\omega$ -deterministic relation represented by the decision diagram $t$ (page 62)

### A.3.2 Tree Schemata

$\langle_F$	ordering on the set of labels $F$
$\bigcirc$	the choice node
$\mathcal{H}_{\bigcirc}(F)$	the set of trees on $F \cup \{\bigcirc\}$
$\text{Ens}(S)$	the set of trees represented by a skeleton $S$ (page 89)
$\text{Sk}(F)$	the set of skeletons on $F$
$\text{Sk}^\cdot(F)$	the set of sets of trees labeled by $F$ that can be represented by a skeleton
$\mathbf{1}_F$	the skeleton representing the set $\mathcal{H}(F)$ (page 92)
$\text{cps}(S)$	the set of paths of $S$ labeled by a choice

$choice(p)$	$p$ can be a path or a choice node. In both cases, it is the number of possible choices at this point
$\mathcal{CS}(S)$	the choice space of a skeleton $S$ (page 93)
$\langle S \rangle$	the function defined by the skeleton $S$ on its choice space (page 93)
$skel(T)$	the skeleton of the tree schema $T$
$link(C)$	the link associated with the choice node $C$
$entry(C)$	the entry associated with the choice node $C$
$rel(T)$	the relation on the tree schema $T$ (page 99)
$Set(T)$	the set of trees represented by the tree schema $T$
$T_\emptyset$	the function that associates with every link of a tree schema $T$ the empty list (page 99)
$G(l)_{(x)}$	the set of possible choices for $l$ on entry name $x$ , according to the global information $G$ (page 99)
$G_{x=v}^l$	the new global information after taking the choice $v$ for the entry $x$ of the link $l$ while the global information is $G$ (page 100)
$\mathcal{TS}(F)$	the set of tree schemata on $F$
$\mathcal{TS}^\cdot(F)$	the set of sets of trees labeled by $F$ that can be represented by a tree schema
$t_x^{-1}(T)$	the projection of the tree schema $T$ according to the node $x$ of the tree $t$ (page 130)
$var(T)$	the set of variables of the tree schema $T$

# Appendix B

## Algorithms

### B.1 Notations

Algorithms are always presented in a pseudo language for more generality. This pseudo language is based on imperative languages. It uses mathematical notations and English sentences, but also some key words and an indentation to outline the control flow of the algorithms.

Each algorithm description starts with the name of the algorithm with a list of arguments in brackets separated by a coma. The description of the algorithm is then indented to show the extend of this description. This description consists in a sequence of expressions to be executed in order.

- $\leftarrow$  is used to affect values to variables. The left hand side of the arrow is the variable to be affected, and the right hand side is the new value for the variable.
- $=$  is used for comparison in tests. Its value, then, is either **true** or **false**. It can be used as a statement, to remind the reader of some equalities.
- $\forall$  followed by a membership expression (of the form  $x \in E$ ) and a **do**. It loops on the following expression(s) for each elements of  $E$ . If there is more than one expression, they are indented to show the extend of the loop.
- do** is used to start a sequence of expressions in a loop (such as  $\forall$ , **for**, **while**). It can start a loop by itself. The sequence of expressions in the loop can then be followed by a **while** followed by a test. The **while** is on the same indentation as the **do**. The loop is performed until the test is false.
- else** see **if**.

- for** followed by an affectation, **to**, a value, then **do** and one or more expressions. The affectation is on a variable  $x$  containing an integer. It loops on the expression, increasing the value of  $x$  by one at the end of each loop until it is greater than the value following **to**. If there is more than one expression, they are indented to show the extend of the loop.
- if** followed by a test which is delimited by a **then**, followed by one or more expressions to be executed if the test is true, and an optional **else** followed by one or more expressions to be executed if the test is false. The range of **then** or **else** is expressed by an indentation. The **else** of a given **if** is always on the same indentation as the **if**.
- return** with an argument, is the end of the current execution of the algorithm. The argument, if any, is the value computed by the algorithm. Note, that any loop or any test in the current algorithm is ended.
- then** see **if**.
- to** see **for**.
- while** followed by a test which is delimited by a **do**. If the test is true, it loops on the following expression(s) while the test is true. The expressions that are in this loop are indented. An alternative syntax describes the same loop with at least one execution of the expressions. See **do**.

## B.2 List of Algorithms

- addKeys(dt)** adds the keys corresponding to the recursive relation represented by **dt** in the dictionary. (page 71)
- bestOpen( $R$ )** returns the decision tree of the greatest open relation representing the recursive relation  $R$ . (page 66)
- compareRel( $t_1, t_2, S, dt$ )** is called with an initial function  $S$  empty. Then returns **true** if and only if  $t_1$  and  $t_2$ , subtrees of **dt**, represent the same subrelation of  $\Omega(dt)$ . (page 69)
- dt $_{\omega}$ ( $R$ )** returns the decision tree representing the  $\omega$ -deterministic relation  $R$ . (semi-algorithm)
- dt $_{op}$ ( $R$ )** returns a decision tree representing  $R$ , where  $R$  is an open relation. This decision tree is used to represent  $\Omega(R)$  if  $R$  is the greatest open relation representing  $\Omega(R)$ . In the final representation, we use **dt $_{reg}$ ( $R$ )** to represent the open relation. (semi-algorithm)

- $\text{dt}_{reg}(R)$  returns the decision tree representing the regular relation  $R$  (before elimination of redundant nodes). (semi-algorithm)
- $\text{dt}_{set}(R)$  returns the decision tree representing  $R$ , where  $R$  is a finite union of recursive relations. (semi-algorithm)
- $\text{eqTree}(M, N)$  returns **true** if and only if the two nodes  $M$  and  $N$  of a graph represent the same tree (page 19)
- $\text{inclusion}(T, U)$  returns **true** if  $\text{Set}(T) \subset \text{Set}(U)$  (page 116)
- $\text{inclusionChildren}(T, U, \Gamma_0, A)$  computes the inclusion of the tree schemata  $T$  and  $U$  labeled by the same label in  $F$ .
- $\text{instGlobReIs}(\Gamma, l, x, v, i)$  instantiates the global informations in  $\Gamma$  with the entry  $x$  of the link  $l$  taking the value  $v$ . The link  $l$  is a link of the  $i^{\text{th}}$  tree schema.
- $\text{inclusionRel}(\text{dt}_1, \text{dt}_2)$  returns **true** if every vector in the  $\omega$ -deterministic relation represented by decision tree  $\text{dt}_1$  is in the  $\omega$ -deterministic relation represented by the decision tree  $\text{dt}_2$ . (page 75)
- $\text{inclusionSkel}(T, U)$  returns **true** if  $\text{Ens}(T) \subset \text{Ens}(U)$  (page 115)
- $\text{instantiate}(\text{dt}, \text{nt}_R, x, v)$  returns the decision tree and the name tree of the relation  $R_\omega(\text{dt})_{:x=v}$  (page 76)
- $\text{instNameTree}(\text{nt}_R, x)$  returns the name tree  $\text{nt}_R$  where the first occurrence of  $x$  have been removed.
- $\text{interRel}(\text{dt}_1, \text{dt}_2)$  returns the decision tree representing the  $\omega$ -deterministic relation  $R_\omega(\text{dt}_1) \wedge R_\omega(\text{dt}_2)$  (page 77)
- $\text{intersection}(T, U)$  returns the tree schema representing the intersection  $\text{Set}(T)$  and  $\text{Set}(U)$  (page 121)
- $\text{interSkel}(T, U)$  returns the skeleton representing the intersection of  $\text{Ens}(T)$  and  $\text{Ens}(U)$  (page 120)
- $\text{leastDeterministic}(R)$  returns the MDD of the least  $\omega$ -deterministic relation containing the regular relation  $R$  (page 80)
- $\text{mdd}(R)$  returns the MDD of the finite relation  $R$  (page 53)
- $\text{mergeEntries}(t, s, n, S)$  is first called with  $n = 0$  and  $S$  empty.  $t$  and  $s$  are the trees representing the entry names of the regular relations  $R$  and  $S$ . It returns a tree representing a set of entry names which is compatible with both  $R$  and  $S$ . (page 74)

`projection( $t, x, T$ )` returns the projection of the tree schema  $T$  on the tree  $t$  according to the variable  $x$  (page 132)

`projSkel( $t, x, T$ )` returns the projection of the skeleton  $T$  on the tree  $t$  according to the variable  $x$  (page 131)

`recCons( $t, x$ )` returns the tree  $t$  where every subtree labeled by  $x$  have been replaced recursively by  $t$ . (page 39)

`representation( $t$ )` returns a node (in a graph) representing the tree  $t$ . This representation of  $t$  is unique, as long as we always use the two dictionaries  $D$  and  $D_G$ . (page 25)

`represent( $t, T$ )` is the recursive algorithm for `representation`. It finds out the cycles with the help of the set of trees above the current tree,  $T$ . When a cycle is found, calls for `representCycle`.

`representCycle( $N$ )` returns the uniquely determined node representing the same tree as  $N$  which is in a cycle.

`share( $G$ )` modifies the graph  $G$  (by merging some of its nodes) so that any couple of different nodes of  $G$  represent different trees (page 20)

`initiateBlocks( $N$ )` creates the partition `Blocks` of the graph according to the labels of the graphs

`shareWithDone( $N$ )`  $N$  is a node of a graph labeled by partial keys. If this node represent the same tree as one of the nodes that can be reached through the partial keys, returns `true`. Must be called with empty dictionaries  $D_D$  and  $D_S$ . If the algorithm returns `true`, then the node equivalent to  $N$  is stored in  $D_S(N)$ . (page 26)

`tryShare( $M, N, i, O$ )` is used to determine whether the node  $M$  is equivalent to the already treated node  $N$  or any node reachable from  $N$ . We have  $M.i = O$  which is already treated and is used to restrain the number of calls to `compareWithDone`.

`compareWithDone( $M, N$ )` determines whether the node  $M$  represents the same tree as the node  $N$  which is already in  $D$  (and so a representation with maximal sharing).

`treeKey( $N$ )` returns the tree that is the key of the node  $N$  of a strongly connected graph (page 22)

- `treeMap`( $N, \mathcal{F}$ ) recursively applies the function  $\mathcal{F}$  to the tree represented by the node  $N$ , and returns the result of this operation. We suppose that a fixpoint strategy is available for  $\mathcal{F}$ . (page 36)
- `treeSubst`( $t_1, x, t_2$ ) The substitution of the label  $x$  by the tree  $t_2$  in the tree  $t_1$  (page 38)
- `unavoidableSet`(`dt`) returns the set of unavoidable relations defined by the decision tree `dt`. (page 71)
- `unfoldTrue`( $t, r$ ) returns the decision tree  $t$  where every `true` node reachable without going through a `iter` is replaced by  $r$ . (page 77)
- `unionRel`(`dt1`, `dt2`) returns the decision tree of the least  $\omega$ -deterministic relations containing the union of the two  $\omega$ -deterministic relations represented by the decision trees `dt1` and `dt2`. (page 81)
- `unionSkel`( $T, U$ ) returns the tree schema representing the union of the two skeletons  $T$  and  $U$  (page 125)
- `unionWithLinks`( $T, U, A$ ) returns the skeleton of the union of  $T$  and  $U$  and the set of union links informations collected in the process.
- `makeUnionLink`( $L, T$ ) adds to  $T$  the links described in the union link information  $L$ .





# Bibliography

- [AH84] Dana Angluin and Douglas N. Hoover. Regular prefix relations. *Mathematical Systems Theory*, 17:167–191, 1984.
- [AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [Aik94] Alexander Aiken. Set constraints: Results, applications and future directions. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 326–335. Springer-Verlag, May 1994.
- [AM91] Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 427–446. Springer-Verlag, 1991.
- [AN80] André Arnold and Maurice Nivat. Formal computations of non deterministic recursive program schemes. *Mathematical Systems Theory*, 13:219–236, 1980.
- [AN92] André Arnold and Damian Niwiński. Fixed point characterization of weak monadic logic definable sets of trees. In M. Nivat and A. Podelski, editors, *Tree Automata and Languages*, volume 10 of *Studies in Computer Science and Artificial Intelligence*, pages 159–188. Elsevier Science Publishers, 1992.
- [And86] Nils Andersen. Approximating term rewriting systems with tree grammars. Technical Report 86/16, Institute of Datalogy, University of Copenhagen, 1986.
- [AW92] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints. In *IEEE - LICS '92*, pages 329–340, 1992.

- [BCL91] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, Edinburgh, Scotland, August 1991.
- [BCM<sup>+</sup>90] Jerry R. Burch, Edmund M. Clarke, K. L. McMillan, David L. Dill, and J. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. In *Fifth Annual Symposium on Logic in Computer Science*, June 1990.
- [BHMSV84] R. K. Brayton, G. D. Hachtel, C. T. Mc Mullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.
- [BJ88] M. Bruynooghe and G. Janssens. An instance of abstract interpretation integrating type and mode inferencing. In R. Kowalski and K. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, volume 1, pages 669–683. MIT Press, 1988.
- [BKR97] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata*, volume 1260 of *Lecture Notes in Computer Science*, 1997.
- [BMW91] Jürgen Börstler, Ulrich Möncke, and Reinhard Wilhelm. Table compression for tree automata. *ACM Transactions on Programming Languages and Systems*, 13(3):295–314, July 1991.
- [BN80] Luc Boasson and Maurice Nivat. Adherences of languages. *Journal of Computer and System Sciences*, 20(3):285–309, 1980.
- [BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, June 1990.
- [Bru93] G. Bruns. A practice technique for process abstraction. In *4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 37–49. Springer-Verlag, 1993.
- [Bry86] Randal E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.
- [Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

- [BT92] B. Bogaert and Sophie Tison. Equality and disequality constraints on direct subterms in tree automata. In A. Finkel and M. Jantzen, editors, *9th Annual Symposium on Theoretical Aspects of Computer Science (STACS 92)*, volume 577 of *Lecture Notes in Computer Science*, pages 161–171. Springer-Verlag, February 1992.
- [Büc60] J. R. Büchi. On a decision method in restricted second order arithmetics. In E. Nagel et al., editors, *International Congress on Logic, Methodology and Philosophy of Science*. Stanford University Press, 1960.
- [Büc73] J. R. Büchi. The monadic theory of  $\omega_1$ . In *Decidable Theories II*, volume 328 of *Lecture Notes in Mathematics*. Springer, 1973.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252, 1977.
- [CC79] Patrick Cousot and Radhia Cousot. Constructive version of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.
- [CC82] A. Cardon and M. Crochemore. Partitioning a graph in  $O(|A| \log_2 |V|)$ . *Theoretical Computer Science*, 19:85–98, 1982.
- [CC92a] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.
- [CC92b] Patrick Cousot and Radhia Cousot. Abstract interpretation framework. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
- [CC95] Patrick Cousot and Radhia Cousot. Formal languages, grammar and set-constraint-based program analysis by abstract interpretation. In *Conference on Functional Programming and Computer Architecture (FPCA '95)*, pages 170–181. ACM Press, June 1995.
- [CES83] Edmund M. Clarke, E. Allen Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. In *10th ACM Symposium on Principles of Programming Languages (POPL '83)*, 1983.
- [CGL94] Edmund M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

- [CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear constraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages (POPL '78)*, pages 84–97, 1978.
- [Cha94] Witold Charatonik. Set constraints in some equational theories. In Jean-Pierre Jouanaud, editor, *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 304–319. Springer-Verlag, September 1994.
- [CMR92] M.-M. Corsini, K. Musumbi, and A. Rauzy. The  $\mu$ -calculus over finite domains as an abstract semantics of Prolog. In M. Billaud, P. Castran, M.-M. Corsini, K. Musumbu, and A. Rauzy, editors, *Workshop on Static Analysis*, number 81–82 in Bigre, September 1992.
- [CNP93] Hugues Calbrix, Maurice Nivat, and Andreas Podelski. Ultimately periodic set words of rational  $\omega$ -languages. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *9th International Conference on Mathematical Foundations of Programming Semantics (MFPS '93)*, volume 802 of *Lecture Notes in Computer Science*, pages 554–566. Springer-Verlag, April 1993.
- [Col82] Alain Colmerauer. PROLOG and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, volume 16 of *APIC Studies in Data Processing*, pages 231–251. Academic Press, 1982.
- [Col84] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *International Conference on Fifth Generation Computer Systems (FGCS'84)*, pages 85–99. Elsevier Science Publishers, 1984.
- [Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. PhD thesis, Université de Grenoble, March 1978.
- [Cou96] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.
- [CP95] Jiazhen Cai and Robert Paige. Using multiset discrimination to solve language processing without hashing. *Theoretical Computer Science*, 145:189–228, 1995. URL “ftp://ftp.diku.dk/diku/semantics/papers/D-209.ps.Z”.
- [CP98a] Witold Charatonik and Andreas Podelski. Co-definite set constraints. In T. Nipkow, editor, *9th International Conference on*

- Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 211–225. Springer-Verlag, March–April 1998.
- [CP98b] Witold Charatonik and Andreas Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *First International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384, pages 358–375. Springer-Verlag, March–April 1998.
- [CS91] R. Cleaveland and B. Steffen. A linear-time model-checking for alternation free modal  $\mu$ -calculus. In *Third workshop on Computer Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, July 1991.
- [Dam96] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Technische Universiteit Eindhoven, 1996.
- [Deu92] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE'92 International Conference on Computer Languages*. IEEE Press, 1992.
- [DTT97] P. Devienne, JM. Talbot, and Sophie Tison. Solving classes of set constraints with tree automata. In G. Smolka, editor, *3th International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 62–76. Springer-Verlag, October 1997.
- [EC80] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer-Verlag, 1980.
- [EF80] Joost Engelfriet and Gilberto Filè. Formal properties of one-visit and multi-pass attribute grammars. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 182–194. Springer-Verlag, 1980.
- [EJ91] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *32nd Annual Symposium on Foundations of Computer Science*, pages 368–377. IEEE Computer Society, 1991.

- [EJS93] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of  $\mu$ -calculus. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 385–391, June/July 1993.
- [Eng94] Joost Engelfriet. Graph grammars and tree transducers. In Sophie Tison, editor, *Trees in Algebra and Programming — CAAP '94*, volume 787 of *Lecture Notes in Computer Science*, pages 15–36. Springer-Verlag, April 1994.
- [ES88] E. Allen Emerson and Jai Srinivasan. Branching time temporal logic. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 123–172. Springer-Verlag, May/June 1988.
- [EV94] Joost Engelfriet and Heiko Vogler. The translation power of top-down tree-to-graph transducers. *Journal of Computer and System Sciences*, 49:258–305, 1994.
- [FS91] G. Filé and P. Sottero. Abstract interpretation for type checking. In J. Maluszyński and M. Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*, pages 311–322. Springer-Verlag, August 1991.
- [GB94] Daniel Geist and Ilan Beer. Efficient model checking by automated ordering of transition relation partitions. In David L. Dill, editor, *Sixth International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 299–310. Springer-Verlag, 1994.
- [GF93] Aarti Gupta and Allan L. Fisher. Parametric circuit representation using inductive boolean functions. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 15–28. Springer-Verlag, 1993.
- [GN84] F. Gire and Maurice Nivat. Relations rationnelles infinitaires. *Calcolo*, 21:91–125, 1984.
- [Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.
- [GS84] F. Gécseg and M. Steinby. *Tree Automata*. Akadémia Kiadó, 1984.

- [HD93] Alan J. Hu and David L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 4–14. Springer-Verlag, 1993.
- [Hei92] Nevin Heintze. *Set Based Program Analysis*. PhD thesis, School of Computer Science, Carnegie Mellon University, October 1992.
- [HJ90] Nevin Heintze and Joxan Jaffar. A finite presentation theorem for approximating logic programs. In *ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 197–209, 1990.
- [HJJ<sup>+</sup>96] Jesper G. Henriksen, Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, 1996.
- [HK87] K. Horiuchi and T. Kanamori. Polymorphic type inference in Prolog by abstract interpretation. In K. Furukawa, H. Tanaka, and T. Fujisaka, editors, *6th Conference on Logic Programming*, volume 315 of *Lecture Notes in Computer Science*, pages 195–214. Springer-Verlag, June 1987.
- [Hop71] John Hopcroft. An  $n \log n$  algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of machines and computations*, pages 189–196. Academic Press, 1971.
- [IS98] Husain Ibraheem and David A. Schmidt. Adapting big-step semantics to small-step style: Coinductive interpretations and “higher-order” derivations. In C. Talcott, editor, *2nd Workshop on Higher-Order Techniques in Operational Semantics*. Elsevier Science Publishers, 1998.
- [JM79] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. In *6th POPL*, pages 244–256. ACM Press, January 1979.
- [JW95] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 393–407, 1995.
- [KMM<sup>+</sup>97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Computer*

- Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 424–435. Springer-Verlag, 1997.
- [Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968.
- [Knu73] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.
- [Knu97] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.
- [Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.
- [KR96] Nils Klarlund and Theis Rauhe. BDD algorithms and cache misses. Technical Report RS-96-5, BRICS, Department of Computer Science, University of Aarhus, 1996.
- [Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.
- [Lev97] F. Levi. Abstract model checking of value-passing processes. In A. Bossi, editor, *International Workshop on Verification, Model Checking and Abstract Interpretation*, 1997.
- [Mas92] François Masdupuy. Array operations abstractions using semantic analysis of trapezoid congruences. In *International Conference on Supercomputing*, 1992.
- [Mau98] Laurent Mauborgne. Abstract interpretation using typed decision graphs. *Science of Computer Programming*, 31(1):91–112, May 1998.
- [Mau99] Laurent Mauborgne. Binary decision graphs. In A. Cortesi and G. Filé, editors, *Static Analysis Symposium (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*, pages 101–116. Springer-Verlag, 1999.
- [Mic68] D. Michie. “memo” functions and machine learning. *Nature*, 218:19–22, April 1968.
- [Mis84] P. Mishra. Towards a theory of types in Prolog. In *International Symposium on Logic Programming*, pages 289–298. IEEE Computer Society Press, 1984.



- [MP88] Zohar Manna and Amir Pnueli. The anchored version of the temporal framework. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 201–284. Springer-Verlag, May/June 1988.
- [MR85] P. Mishra and U. S. Reddy. Declaration-free type checking. In *12th ACM Symposium on Principles of Programming Languages (POPL '85)*, pages 7–21, 1985.
- [MS81] Jocelyne Mongy-Steen. *Transformation de noyaux reconnaissables d'arbres. Forêts RATEG*. PhD thesis, LIF de Lille, February 1981.
- [Rab69] Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, July 1969.
- [Rey69] J. Reynolds. Automatic computation of data set definitions. In *Information Processing '68*, pages 456–461. Elsevier Science Publisher, 1969.
- [Saf88] S. Safra. On the complexity of omega-automata. In *29th Annual Symposium on Foundations of Computer Science*, pages 319–327. IEEE Computer Society, October 1988.
- [Sch90] Michael I. Schwartzbach. Infinite values in hierarchical imperative types. In A. Arnold, editor, *15th Colloquium on Trees in Algebra and Programming (CAAP '90)*, volume 431 of *Lecture Notes in Computer Science*, pages 254–268. Springer-Verlag, May 1990.
- [Sch97] David A. Schmidt. Abstract interpretation of small-step semantics. In *5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 76–99. Springer-Verlag, 1997.
- [Sch98a] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *25th ACM Symposium on Principles of Programming Languages (POPL '98)*, 1998.
- [Sch98b] David A. Schmidt. Trace-based abstract interpretation of operational semantics. *Journal of Lisp and Symbolic Computation*, 10(3):237–271, 1998.
- [Sei90] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, June 1990.

- [SKMB90] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *IEEE International Conference on Computer-Aided Design*, pages 92–95, 1990.
- [SNW94] V. Sassone, M. Nielson, and G. Winskel. Relationship between models of concurrency. In *Proceedings of the Rex '93 School and Symposium*, 1994.
- [Sør94] Morten Heine Sørensen. A grammar-based data-flow analysis to stop deforestation. In Sophie Tison, editor, *Trees in Algebra and Programming — CAAP '94*, volume 787 of *Lecture Notes in Computer Science*, pages 335–351. Springer-Verlag, April 1994.
- [SVW85] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for büchi automata, with applications to temporal logic. In *12th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1985.
- [Tar55] A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.
- [Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.
- [Tho88] Wolfgang Thomas. Computation tree logic and regular  $\omega$ -languages. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 690–713. Springer-Verlag, May/June 1988.
- [Tho90] Wolfgang Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 135–191. Elsevier Science Publishers, 1990.
- [TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2:57–82, 1968.
- [Var94] Moshe Y. Vardi. Nontraditional applications of automata theory. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 575–597. Springer-Verlag, April 1994.

- [vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computer Machinery*, 23(4):733–742, October 1976.
- [Ven97] Arnaud Venet. Abstract interpretation of the  $\pi$ -calculus. In *LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 51–75. Springer-Verlag, 1997.
- [vGG89] R. von Glabbeck and U. Goltz. Partial order semantics for refinement of actions. *Bulletin of the EATCS*, 34, 1989.
- [YS87] E. Yardeni and E. Y. Shapiro. A type system of logic programs. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pages 211–244. MIT Press, 1987.



# Index

- $\omega$ -deterministic relation, 60
- $\omega$ -regular relation, 48
- automaton, 7, 49, 55, 85–86, 88, 109
- BDD, 41, 50, 53, 92, 94, 160
- child, 8
- choice space, 63, 93–94
- connected, 11
  - sharing, 19–23
- entry, 12
  - equivalent, 45, 111
  - name, 44, 73–75, 96, 111, 113
- equivalence, 6, 19, 23, 30
- father, 8
- fixpoint, 5, 10, 15–16, 36, 89, 100, 107
- iterative relation, 47, 55
- key, 18, 22
- lattice, 6
- link, 93–106, 113, 119, 123, 124, 135, 136, 138
- maximal sharing, 18
- open relation, 47, 55
  - greatest, 56–57, 66–69
- ordering, 5–6, 30, 89, 100
  - alphabetic, 7
  - approximation, 15, 150
  - computation, 15
  - covering, 97, 114
  - labels, 90, 93
  - prefix, 7
  - subtree, 8
  - variables, 74
- parent, 8
- path, 7, 8, 11, 22, 43, 64, 112, 154
  - compression, 11, 50
  - to choice, 93, 97, 114
- periodic
  - entry names, 46, 50
  - word, 7
- prefix, 7
- prefix regular, 47
- projection, 13, 14, 44, 130–132
- ranked, 8
- redundant node, 52
- redundant nodes, 65, 72, 74
- regular
  - relation, 48, 59
  - tree, 9
- relation, 12–14
- return edges, 12
- root
  - graph, 12
  - tree, 8
- skeleton, 88–93, 102, 104, 114, 119, 124, 125, 131
- square relation, 13, 55
- subgraph, 12
- subtree, 8
- widening, 16, 135, 151