Declarative I/O in a functional logic language

Nancy Mazur

August 26, 1997

Acknowledgement

Herewith I would like to thank some people who have helped me realize this thesis.

First of all, I would like to thank my supervisor Prof. M. Hanus, for his constructive comments, which made this text become what it is.

I would also like to thank Prof. D. De Schreye, also my supervisor, for his support and enthusiasm in this project.

And finally, I wish to thank my parents who have introduced me to the beautiful world of computer science.

Dankwoord (Dutch)

Hiermee wil ik nog even van de gelegenheid gebruik maken om enkele mensen te bedanken die mij geholpen hebben bij het tot stand komen van deze thesis.

Vooreerst wil ik mijn promotor Prof. M. Hanus bedanken, die mij met zijn constructieve kritiek steeds op weg heeft geholpen.

Prof. D. De Schreye, ook mijn promotor, wens ik te bedanken voor zijn goede raad en enthousiasme in dit project.

Tenslotte dank ik mijn ouders die mij reeds op jonge leeftijd kennis hebben laten maken met de wondere wereld van de informatica.

Samenvatting (Dutch)

In het domein van declaratieve talen maakt men meestal een onderscheid tussen functionele talen en logische talen. Gedurende de laatste jaren werd de interesse steeds groter om deze twee werelden te verzoenen, om zo tot functioneel logische talen te komen. Deze talen bieden enerzijds de functionaliteiten aan die men uit functionele talen kent (geneste uitdrukkingen, hogere orde functies) en anderzijds de functionaliteiten eigen aan logische talen (logische variabelen, ingebouwde zoekmechanismen), om zo tot krachtigere, meer expressieve, efficiëntere declaratieve talen te komen.

Curry [11, 10], een functioneel logische taal ontwikkeld aan de technische universiteit Aachen (RWTH Aachen), wil zich als een programmeertaal voor algemeen nut profileren, met als bedoeling een standaard op het gebied van geïntegreerde functioneel logische talen te worden.

Om dit te verwezenlijken moet de taal verrijkt worden met mogelijkheden voor invoer/uitvoer. Deze uitbreiding mag echter het declaratieve karakter van de taal niet schaden.

Monaden, vooreerst een zuiver wiskundig object [17], maar ondertussen reeds intens gebruikt bij luie functionele talen zoals Haskell [20, 21, 25, 23], bieden een uitermate elegante oplossing tot het integreren van invoer/uitvoer in declaratieve talen.

Het hoofddoel van mijn thesis is Curry te voorzien van invoer/uitvoer door gebruik van deze mathematische objecten: monaden. Door het functionele karakter van deze taal kunnen monaden zoals ze voorgesteld en gebruikt worden in functionele talen direct overgenomen worden. Door het logisch karakter van de taal echter, treden er problemen op indien niet-deterministische berekeningen bij invoer/uitvoer optreden. Zulke niet-deterministische berekening moeten daarom vermeden worden. Een eerste oplossing bestaat erin een fout tijdens de uitvoering te genereren indien niet-determinisme optreedt. Een tweede en elegantere oplossing is zulke problemen reeds tijdens compilatie aan te wijzen. Dit kan door gebruik te maken van de techniek van abstract interpretatie.

Curry: een functioneel logische taal

Curry is een functioneel logische taal die zich baseert op twee uitvoeringsmechanismen ontwikkeld in het domein van functioneel logische talen: "residuation" (hierop steunt o.a. Escher [14]) en "needed narrowing" [2]. Praktisch betekent dit dat een programma opgesteld is uit functies die eventueel genest kunnen zijn. Ook hogere orde functies zijn mogelijk. Het laat toe logische variabelen te gebruiken, en dus eventueel niet deterministische evaluatiestappen.

Indien een vrije variabele als argument in een functie optreedt, wordt deze over het algemeen geünificeerd met de mogelijke waarden waarvoor de bepaalde functie gedefinieerd is. Soms is echter zulk een unificatie-stap ongewenst, waardoor het mogelijk is de evaluatie van bepaalde functies uit te stellen tot de argumenten wel voldoende geïnstantieerd zijn.

Voorbeeld 1
De volgende regels zijn geldige code in Curry:

```
p(100) = 1.
p(200) = 2.
```

Indien men in de oproep `p(X)`, `X` als een logische variabele beschouwt, dan kan deze geünificeerd worden zowel met de waarde 100 als met de waarde 200. In het eerste geval zal de uitkomst van de oproep 1 zijn, in het tweede geval zal dit 2 geven. Dit kan men voorstellen als:

$$p(X) \rightsquigarrow \{X = 100\}\ 1 \mid \{X = 200\}\ 2.$$

Indien hier echter geen unificatie gewenst is, zal dezelfde oproep uitgesteld worden. Het onderscheid kan door de programmeur opgegeven worden door specifiek te vermelden welke argumenten "*flex*" zijn (unificatie toegelaten), en welke "*rigid*" horen te zijn (de functie-evaluatie wordt uitgesteld zolang het argument niet voldoende geïnstantieerd is). □

Globaal bekeken geldt dat het uitvoeringsschema van functies steeds zo lui mogelijk is.

Intern wordt het gewenste uitvoeringsschema van de gedefinieerde functies in een programma door middel van definitionele bomen voorgesteld.

Invoer/Uitvoer in Curry

Karakteristieken die van natuur uit eigenlijk niet puur zijn, kunnen met behulp van monaden toch op een pure wijze geïntegreerd worden in zuiver declaratieve talen. Zo wordt in Haskell [20], een zuivere luie functionele taal, reeds uitvoerig gebruikt gemaakt van deze mathematische objecten om met globale toestandsvariabelen om te gaan, alsook om invoer/uitvoer te verwezenlijken.

Zonder in de details te gaan van wat monaden nu echt zijn, kunnen we toch hier illustreren hoe men ze in Haskell gebruikt in het kader van invoer/uitvoer.

Voorbeeld 2

```
main :: IO().
main = getChar >>= putChar.
```

`IO()` stelt de invoer/uitvoer monad voor. In wezen is het niets anders dan een abstract data type. Het maakt abstractie van de wereld zelf die dan door de programmeur niet meer direct bereikbaar is. Terwijl klassieke constructoren waarden voorstellen, zo stelt de invoer/uitvoer monad acties voor. Pas als deze acties uitgevoerd worden, kunnen ze effect hebben.

Om het monadisch systeem te kunnen gebruiken moet het voorzien zijn van enkele basis-acties (`getChar`,`putChar`), alsook van enkele primitieve functies die toelaten deze basisacties met mekaar te verbinden (`>>=`).

Uiteindelijk, om invoer/uitvoer uitgevoerd te krijgen, zal ook de top-level functie het type `IO()` hebben.

Het programmaatje opgegeven in dit voorbeeld kan men dan als volgt lezen: indien `main` wordt uitgevoerd zal eerst een symbool uit de invoerstroom gelezen worden. De ingelezen waarde wordt dan doorgegeven (`>>=`) aan de volgende actie, namelijk `putChar`, die dan deze zelfde waarde op de uitvoerstroom plaatst. □

Gezien ook Curry deels functioneel is, kan men dezelfde constructies als deze in Haskell gewoonweg overnemen.

Curry is echter ook deels logisch, wat tot problemen leidt, zoals de volgende voorbeelden aantonen.

Voorbeeld 3

```
puzzle :: io().
puzzle if isPuzzle(X) = printAndSolve(X).
isPuzzle(a) = true.
isPuzzle(b) = true.
```

Tijdens de uitvoering van `puzzle` zal eerst getracht worden `isPuzzle(X)` op te lossen. Er zijn twee mogelijk oplossingen voor deze oproep, nl. $\{X = a\}$ *true* | $\{X = b\}$ *true*. Er zouden dus twee puzzels gelijktijdig uitgedrukt en opgelost moeten worden, maar dit is onmogelijk. We kunnen niet zomaar de wereld in twee verdelen. □

Voorbeeld 4

```
respond_to :: char -> io().
respond_to(a) = putString("Hi").
respond_to(X) = putString("Ha").
```

Ook dit voorbeeld zal bij uitvoer hetzelfde probleem geven als het vorige. Indien namelijk deze functie opgeroepen wordt met als argument de waarde a, dan zouden twee acties tegelijkertijd moeten uitgevoerd worden, namelijk het uitdrukken van de tekens "Hi" en het uitdrukken van "Ha". □

De oorzaak van de disjunctie is in beide voorbeelden verschillend. Terwijl in het eerste voorbeeld vermeden zou moeten worden dat de functie `isPuzzle` opgeroepen wordt met een vrije variabele als argument, moet in het tweede voorbeeld de oproep van één specifieke instantie van de functie `respond_to` vermeden worden.

Om aan deze situatie te verhelpen bestaan er twee oplossingen. Een eerste mogelijkheid bestaat erin de uitvoering van een I/U functie te stoppen indien een disjunctie optreedt. Het zou echter veel interessanter zijn om reeds op het moment van compilatie waarschuwingen te melden dat eventueel een disjunctie zou kunnen optreden. Gezien één van de oorzaken van disjunctie afhankelijk is van het al dan niet geïnstantieerd zijn van specifieke argumenten, is het voor de hand liggend zulk een determinisme analyse te laten steunen op de techniek van abstracte interpretatie.

Determinisme analyse

Eén van de oorzaken van niet-determinisme ligt in het niet geïnstantieerd zijn van specifieke argumenten. Om dit te onderzoeken moet dus een dataflow analyse voorhanden zijn die de intantiëringen van de argumenten van functies kan nagaan. Dit is een typisch probleem dat opgelost kan worden door middel van abstracte interpretatie. De natuur van de informatie die verzameld moet worden komt overeen met deze die klassiek onderzocht wordt in logische talen, vandaar ook de keuze het opgebouwde determisme analyse systeem te laten steunen op principes uit de abstracte interpretatie van logische talen.

Abstracte interpretatie van logische talen

Abstracte interpretatie bestaat erin een gegeven programma uit te voeren op een abstracte representatie van de gegevens.

In het kader van logische talen zal men, eens de abstracte representatie vastgelegd is, primitieve operaties definiëren die de abstracte gegevens kunnen manipuleren.

Het eigenlijke analyse proces wordt dan gevormd door een correcte opbouw en interpretatie van een abstracte EN-OF boom die alle mogelijke concrete uitvoeringspaden omvat [3].

Determinisme analyse in Curry

Het in deze thesis ontwikkelde abstracte analyse raamwerk voor Curry steunt op dezelfde basisprincipes als dat voor logische talen. Men zal ook primitieve operaties definiëren, en de analyse zal ook gebeuren op basis van abstracte EN-OF bomen.

In plaats van met predikaten te werken zoals in logische talen, zal men hier met functies moeten werken. Het grote verschil met het analyse raamwerk voor logische talen is echter dat de abstracte EN-OF bomen nu dynamisch gegenereerd moeten worden op basis van de aard van het operationeel gedrag van de functies. Gezien het operationeel gedrag van deze functies eenduidig door hun definitionele boom wordt weergegeven, kunnen deze definitionele bomen gebruikt worden voor het opbouwen van de overeenkomstige abstracte EN-OF bomen.

Om nu specifiek determinisme informatie af te leiden zal men het raamwerk toepassen op een gecombineerd abstract domein. Enerzijds moet informatie afgeleid worden over de instantiëringen van de argumenten van een functie, en anderzijds moet de bepaald worden of sommige functies al dan niet deterministisch zijn.

Problemen - Onnauwkeurigheden

Het ontwikkelde analyse systeem geeft nauwkeurige resultaten bij relatief eenvoudige datastructuren. Er zijn enkele aspecten van Curry die moeilijk in het abstracte analyse model kunnen benaderd worden. Dit zorgt ervoor dat de resultaten voor complexere structuren heel wat minder nauwkeurig kunnen zijn.

Curry is een luie taal. Dit heeft twee onmiddellijke gevolgen wat de abstracte interpretatie betreft.

1. De analyse van luie functie posities moet op een strikte wijze gebeuren om te vermijden dat het abstracte domein willekeurig groot wordt. Deze strikte benadering beïnvloedt op een directe wijze de nauwkeurigheid van de afgeleide informatie.

2. Uitdrukkingen in kop-normaal-vorm worden bij uitvoeringstijd soms nooit tot normaalvorm gebracht. Dit gedrag is moeilijk te simuleren tijdens de abstracte interpretatie.

Curry laat ook hogere orde functies toe. Dit stelt een probleem indien functies geanalyseerd moeten worden die als resultaat partiële functies opleveren. Indien deze functies als argument van andere functies voorkomen heeft men alle informatie verloren over welke partiële functies nu echt met dat argument overeen kunnen stemmen. In dat geval zal de analyse dan ook slechte resultaten geven.

Besluit

Curry is een deels functionele taal, waardoor het vrij simpel is deze taal met monadische invoer/uitvoer te voorzien. Door het logische karakter van de taal echter moeten niet-deterministische berekeningen bij interactieve programma's vermeden worden.

Abstracte interpretatie biedt de mogelijkheid aan reeds tijdens het compileren van de code niet-deterministische evaluatieschema's te detecteren. Het lui evaluatie-schema wordt in het ontwikkelde analyse systeem door een strikt evaluatie-schema benaderd. Dit kan dan soms tot minder nauwkeurige resultaten leiden. Hogere orde functies vormen een tweede bron van onnauwkeurigheid.

Hoewel in deze samenvatting relatief weinig ingegaan kon worden op de determinisme analyse, is in het uiteindelijke werk net in dat deel veel tijd gekropen. Functioneel logische talen vormen een tamelijk nieuw onderzoeksdomein, en dus ook de abstracte interpretatie van deze talen. Het abstracte interpretatie proces moest quasi vertrekkend vanaf nul opgebouwd worden.

Nancy Mazur
26 augustus 1997

Contents

Introduction

Declarative programming languages have traditionally been categorised into two distinct worlds. At one side, we had the logic languages, and the other part represented the functional languages. But in fact there is only one declarative world.

During the past few years, research has been going on to reconcile these two worlds. Instead of speaking about purely functional languages, or purely logic languages, now functional logic languages are developed. These languages offer the basic features of functional languages (reduction of nested expressions, higher-order functions), as well as the basic features of logic languages (logic variables, partial data structures, built-in search mechanisms). This combination offers more expressive power with regard to pure functional languages, and more efficient evaluation mechanisms compared to pure logic languages.

There are different ways of combining these worlds though, and in the meantime different operational principles, all aiming at the integration of the two programming paradigms, have been developed (see [8] for a survey).

Curry [11, 10], a functional logic language developed mainly at the technical university of Aachen (RWTH Aachen), intended to become a standard in this field, combines what is thought to be the most promising operational principles: residuation (used in the languages Escher [14], Le Fun [1], ...) and needed narrowing [2].

It's aim is to become a general-purpose programming language. Therefore basic I/O constructs must be provided. The challenge of this work consists in introducing I/O without any loss of the declarative character of the languages. In classical languages (Prolog, Lisp, SML, ...), interaction is offered by means of side-effects. The direct consequence is that reasoning on I/O programs in these languages becomes quite difficult, if not impossible.

From the functional programming community, solutions on how to provide I/O within functional languages, while respecting the declarativeness of the language, have been proposed and used. The most promising of these is the use monads, mathematical constructs inspired by category theory [17].

The first goal of my thesis was simply to extend Curry with monadic I/O. But soon it became clear that declarative I/O in combination with the logic character of the language poses some problems. More precisely, non-deterministic computations steps must be prohibited in interactive programs, as it is not possible to duplicate the world. The easy solution is to produce run-time errors if such situation should occur. The more challenging solution is to provide Curry with an analysis tool which can detect possible non-determinism at compile-time (or while debugging). So the second goal of my work was to develop a generic analysis framework, that can then be used to the more precise problem encountered here: non-determinism.

The structure of the presented text is as follows. First, Curry will have to be introduced, what it looks like, and how programs actually behave at run-time (chapter 1). In chapter 2, monads are presented. This will be done based on their use in the lazy functional language Haskell [20]. Due to the functional character of Curry, we can directly adopt monads in it. The logic facet of Curry will, as already mentioned above, cause us some problems with non-deterministic evaluation schemes. The clear conditions for which such problems occur will be stated.

The second part of this text will be concerned with the detection of non-determinism by means of abstract interpretation. We will need to develop an abstract interpretation framework suitable for analysing programs in Curry. The work will be based on a top-down abstract interpretation framework for logic programs [3]. It will be presented in chapter 3. Chapter 4 presents some general thoughts that will have to be taken into account when adapting that framework to Curry. Before formally defining the analysis framework for Curry (chapter 6), we will introduce it on a more intuitive basis (chapter 5).

In chapter 7, the theory will be finally applied to the original problem: detecting non-determinism. Chapter 8 will then discuss the possible problems and imprecisions of the analysis, and will also propose some improvements.

Chapter 1

Curry

The overview which will be presented here will be constructed as follows. In a first place Curry will be introduced as a functional language. In a second section, the logic language point of view will be shown. Finally the combination of both will be discussed. A formal definition of Curry can be found in [10, 9].

In a last section, we will have to introduce more formalism. The computational behaviour of Curry programs can be uniquely described by means of definitional trees. These trees will be needed when developing an abstract interpretation framework for the language.

We assume that the reader is somewhat familiar with functional programming languages as well as with logic programming languages.

1.1 Curry: a functional programming language

1.1.1 Basics

Example 1.1
Consider the following valid Curry program:

```
data list(A) = [ ] ; [A|list(A)].

append :: list(A) -> list(A) -> list(A).
append([ ], Y)   =  Y.
append([X|Xs],Y) = [X|append(Xs,Y)].

rev :: list(A) -> list(A).
rev([ ])    = [ ].
rev([X|Xs]) = append(rev(Xs),[X]).
```

The reading of this program should be quite obvious.

As can be seen, Curry is a strongly typed language (Hindley-Milner-like polymorphic type system). Basically a program consists of data and function definitions.

From this point of view a *computation* consists of the evaluation of an *expression* to yield *values*. An *expression* is an application $\varphi(e_1, \ldots, e_n)$ where φ/n is a constructor symbol or a defined function, and $e_1, \ldots, e_n$ are expressions (n can be possibly zero). A *pattern* is an expression $f(t_1, \ldots, t_n)$ where each variable occurs only once, and where f is a defined function, $t_1, \ldots, t_n$ are expression which are only made up out of constructor symbols or variables. An expression is *reducible*, if at some position in the expression, a defined function can be rewritten using the rules defining this function. A *value*, or irreducible expression, is an expression that does not contain any function symbols. A *head normal form* is an expression of the form $c(e_1, \ldots, e_n)$ where c is a constructor symbol, and $e_1, \ldots, e_n$ are expressions. In this text we will also use the concept of a *constructor based expression*, which denotes a head normal form, where at least one of the sub-expressions is a reducible function.

1.1.2 General

Curry's evaluation mechanism, from a functional point of view, is lazy. This means that the evaluation of expressions is postponed until the moment these expressions are actually needed. Working with for example infinite lists (or any infinite data structure), is no problem.

Curry provides the usual higher-order features, which means that partial functions are themselves valid expressions, and can therefore occur as arguments of other expressions. The classic example of exploiting higher-order features is map-function which in Curry would look as follows.

Example 1.2

```
map :: (T1 -> T2) -> list(T1) -> list(T2).
map(F,[ ])     = [ ].
map(F,[X|Xs]) = [F@X|map(F,Xs)].
```

where "`@`" denotes the application operation (built-in)[1].

The use of λ-abstractions is also at hand.

1.2 Curry: a logic programming language

Example 1.3

Consider the following elementary lines of a possible Curry-program (assuming that booleans and integers are built-in constants):

[1] The reason why this application function is explicit, and not just written as $F(X)$, is that in the latter notation, F is not allowed to be a reducible expression itself.

```
p :: int -> bool.
p(1) = true.
p(2) = true.
```

Intuitively speaking, a query like $p(X)$, with X being a free variable, would yield that X can either be 1 or 2 and that the value of the predicate will then be *true*.

This is exactly what happens in Curry. The goal $p(X)$ in Curry will be evaluated to:

$$(X = 1 \,[\!]\, true) \vee (X = 2 \,[\!]\, true).$$

So in fact Curry is expanded with the notions of *free variables*, and *unification*, as it is known from logic languages.

The definition of *expression* has to be redefined. An expression can now be either a variable or an application $\varphi(e_1, \ldots, e_n)$ where φ/n is a constructor symbol or a defined function, and $e_1, \ldots, e_n$ are expressions (n can be possibly zero). The concept of the value of a function has to be revisited too. From a functional point of view, one is interested in the value of an expression. From a logic point of view though, the values to which initial free variables are bound are important as well (*substitutions*). This leads to the concept of *answer expression*. An *answer expression* is a pair $\sigma \,[\!]\, e$, where σ represents a substitution, and e forms the expression to which the initial expression got reduced. When the original expression did not contain any free variables, the substitution part will of course be empty, and can therefore be omitted. Since in general more then one answer may exist, the initial expressions are reduced to disjunctions of answer expressions[2]. Therefore, the definition of a *disjunctive expression* is a set of answer expressions, written as $(\sigma_1 \,[\!]\, e_1) \vee \ldots \vee (\sigma_n \,[\!]\, e_n)$[3].

1.3 Curry: a functional logic language

Up to now, we've shown how Curry can be considered as a functional language, and how it can be seen as a logic language. Like it is presented in the previous sections, the evaluation of an expression is done by using the rules defining the functions as a rewrite system, if all arguments are ground, and by unifying those variables that appear to be free at the moment a rule is to be applied. Mainly the guessing part of the right values for some variables can cause some problems.

[2] Such a disjunctive expression can never be infinite. Disjunctions are produced only when different rules (defining a same function) can be applied in order to reduce an expression. But a function is always defined by a limited number of rules.

[3] Throughout the text we will use a shorter notation: $\{\sigma_1\}\, e_1 \,|\, \ldots \,|\, \{\sigma_n\}\, e_n$.

- Some functions might not terminate if some of the arguments are unknown.
- In this model, each function requires an explicit definition of all functions (rules). Problems would arise when trying to extend the language with primitive operations, such as arithmetics. The only way to define the addition of integers would then be to enumerate for all pairs of integers, what the result of the addition would be. Not only would this be impossible to do, but when such an operation would be applied with one of the arguments being free, a true explosion of the search tree would be the result.

All these features can be supported by allowing function calls to be delayed when some argument is not instantiated. The programmer is provided with the possibility to annotate the functions with the information whether an argument is to be *rigid* (the function will be delayed if the particular argument is not instantiated enough) or *flex* (when the particular argument is free, it will be unified with the possible values for which the function is valid).

We can illustrate this behaviour with the following small example:

Example 1.4
Consider the program lines:

```
f :: int -> int.
f eval 1:rigid.
f(1) = 2.
f(2) = 4.

g :: int -> int.
g eval 1:flex.
g(2) = 7.

test1(X) = f(X) + g(X).
test2(X) = g(X) + f(X).
```

*test*1 and *test*2 are just two functions which calculate the sum of two other functions. They only differ in the order in which the two functions appear in the right-hand sides.

Some remarks:

1. The "+" operation is defined as an external function, which means that in order to reduce an expression such as $A + B$, both arguments must be instantiated (more precisely, if the first argument fails to be instantiated, then the evaluation process of this call is all together delayed).

2. Generally, the type of the function is not required to be explicitly given, as Curry will itself check the typing of the program.

3. $test1(X)$, $test2(X)$ have no annotations concerning the instantiation of the arguments. Curry works with defaults. Without any annotation, it uses a standard evaluation scheme.

We can now review how the calls $test1(X)$, and $test2(X)$ will be evaluated.

$$\begin{array}{lcl|l} test1(X) & \leadsto & f(X)+g(X). & \text{simple rewriting} \\ & \leadsto & \perp. & f(X) \text{ is rigid, and “+” is rigid.} \end{array}$$

$$\begin{array}{lcl|l} test2(X) & \leadsto & g(X)+f(X). & \text{again, simple rewriting} \\ & \leadsto & \{X=2\}\,7+f(2). & g(X) \text{ is flex, } X \text{ unified with 2} \\ & \leadsto & \{X=2\}\,7+4. & f(X) \text{ can finally be reduced} \\ & \leadsto & \{X=2\}\,11. & \text{the addition can be performed.} \end{array}$$

After a first rewrite step, $test1(X)$ is reduced to $f(X) + g(X)$. The add operation cannot be directly performed, as f(X) must be reduced first. But f(X) cannot be reduced, X being a free variable, and f being marked as “rigid”. The whole goal gets suspended. This is denoted by the symbol “$\perp$”.

The second goal, $test2(X)$ on the other hand, can be reduced easily. Just like in the previous example, the add operation cannot be directly performed, and $g(X)$ must be reduced first. This can be simply done by binding the argument to the value 2, and reducing $g(X)$ to 7. As X is now instantiated, $f(X)$ can be reduced too, which leads us to the answer expression shown.

If the definition of $g(X)$ would have comprised a second rule such as:

```
g(1) = 3.
```

then $test2(X)$ would have led to the disjunctive answer expression:

$\{X = 2\}\,11 \,|\, \{X = 1\}\,5.$

1.3.1 Some built-in operations and data types

The following data-types are built-in:

- boolean values: true, false.
- constraints: valid (also noted as { }).
- integers: $0, 1, 2, \ldots$ are considered as the basic constructors of this type. The basic operations on integers are also provided.
- lists.

- characters.
- strings.

The basic built-in operations with their signature are:

- equational constraints: $= :: T \to T \to constraint$.
- strict equality: $== :: T \to T \to bool$.
- application function: $@ :: (T_1 \to T_2) \to T_1 \to T_2$.

Furthermore, Curry allows the definition of external functions. The annotations for the arguments in such functions are automatically set to rigid.

Another feature present in the Curry-system is to encapsulate specific non-deterministic computations, avoiding herewith a global search. This can be quite useful to control the efficiency of the computations. This feature is also very important when non-deterministic computations are not wishful for some functions.

1.4 Definitional trees

The previous presentation should already be enough to get an idea of what Curry is, and what is does. It would also have sufficed for the next chapter which will introduce I/O into Curry by the use of monads. But for the subsequent chapters, which will discuss the central problem of I/O within Curry, namely non-determinism, we will need a formal representation of the possible computational behaviour of a function. This can be provided by the use of definitional trees.

A *definitional tree* is a hierarchical structure that contains all rules of a defined function. A definitional tree $\mathcal{T}$ for a specific pattern π can take only one of the following forms:

1. $\mathcal{T} = rule(l \to r)$: the definitional tree consists simply of a variant of a rule that defined the specific function having pattern π $(l = \pi)$.
2. $\mathcal{T} = branch(\pi, o, r, [\mathcal{T}_1, \ldots, \mathcal{T}_n]), n > 0$: o defines a position of a variable within the pattern π; r can be either $rigid$ or $flex$; let $c_1, \ldots, c_n$ be different constructors of a sort matching the one required at position o in the pattern, for all $i = 1, \ldots, n$, $\mathcal{T}_i$ is a definitional tree with a new pattern, namely π where the variable at o is substituted by $c_i(X_1, \ldots, X_m)$ with $X_1, \ldots, X_m$, new variables.
3. $\mathcal{T} = or(\mathcal{T}_1, \mathcal{T}_2)$: where $\mathcal{T}_1, \mathcal{T}_2$ are definitional trees with pattern π.
4. $\mathcal{T} = and(\mathcal{T}_1, \mathcal{T}_2)$: where $\mathcal{T}_1, \mathcal{T}_2$ are definitional trees with pattern π.

Let us now quickly overview the meaning of each of these cases.

1.4.1 $\mathcal{T} = rule(l \rightarrow r)$

This is the most evident case. Evaluating an expression e by using a definitional tree for which the pattern corresponds to e, is simply rewriting the initial expression to the expression that appears as the right-hand side (r) in this tree.

Example 1.5
Let $\mathcal{T} = rule(f(1, X) \rightarrow g(X))$ then the evaluation of $f(1, h(Z))$ using this definitional tree will result in a new expression: $g(h(Z))$.

1.4.2 $\mathcal{T} = branch(\pi, o, r, [\mathcal{T}_1, \ldots, \mathcal{T}_n])$

A branch-node, whether it is annotated with $rigid$ or $flex$, simply states that in order to use the definitional tree $\mathcal{T}$ to reduce a given expression, the subexpression appearing at position o must be evaluated until the expression matches the pattern for which the definitional tree is defined. If the tree is marked as $flex$, then the subexpression is allowed to be a free variable. In the other case, the subexpression must at least be in head-normal form. Further evaluation of the initial expression can then be done using the trees $\mathcal{T}_1, \ldots, \mathcal{T}_n$.

Example 1.6
Let $s/1$ be a unary constructor, and $f/1$ a function described by the definitional tree $\mathcal{T}$:

$$\begin{array}{lcl} \mathcal{T} & = & branch(f(X), 1, rigid, [\mathcal{T}_1]) \\ \mathcal{T}_1 & = & branch(f(s(X)), 1\#1, rigid, [\mathcal{T}_2]) \\ \mathcal{T}_2 & = & rule(f(s(1)) \rightarrow true) \end{array}$$

Let $g/1$ be some defined function. In order to evaluate a call like $f(g(Z))$ using the definitional tree $\mathcal{T}$ for f, $\mathcal{T}$ requires the argument of f to be evaluated. $\mathcal{T}$ does not specify any particular form to which g must be reduced, so a head-normal form is sufficient. Suppose $g(Z)$ gets reduced to $s(h(Z))$, where $h/1$ is yet another defined function, we arrive at the situation where the updated initial call $f(s(h(X))$ must be reduced according to the definitional tree defined in $\mathcal{T}$, namely $\mathcal{T}_1$.

$\mathcal{T}_1$ requires that the argument appearing at position $1\#1$[4] is instantiated ($rigid$). Thus, $h(X)$ must be evaluated first. If this function is reduced to the value 1, then the rule defined in $\mathcal{T}_2$ can be applied and the initial goal will have the result $true$.

[4] In general, a position $A\#B$ denotes the argument at the B'th position of the constructor appearing at the A'th position in the initial goal.

1.4.3 $\mathcal{T} = or(\mathcal{T}_1, \mathcal{T}_2)$

An *or*-node denotes the situation where two definitional trees can be used to calculate the value of a given expression.

Example 1.7
Suppose the following program:

```
f(1) = 0.
f(X) = 2 + X.
```

The definitional tree corresponding to f can be:

$$\begin{array}{l} or(\ branch(f(X), 1, rigid, [rule(f(1) \rightarrow 0)]), \\ \quad\quad rule(f(X) \rightarrow 2 + X)) \end{array}$$

In this case, the call $f(1)$ can be reduced to 0 using the first rule, but can also be reduced to 3 when using the second rule. Both solutions are correct.

1.4.4 $\mathcal{T} = and(\mathcal{T}_1, \mathcal{T}_2)$

The presence of an *and*-node signifies that there is possibly more then one argument that will have to be evaluated before the actual goal can be reduced. The computational behaviour will be to evaluate one of these arguments. If this evaluation process gets suspended due to some insufficiently instantiated argument, then a second attempt is made by first trying to evaluate the other arguments. This behaviour corresponds to a concurrent computation model.

Example 1.8
The most typical example of such an *and*-node is the definition of the conjunction operation.

```
/\ :: constraint -> constraint -> constraint.
{} /\ {} = {}.
```

for which the following definitional tree is defined:

$$\begin{array}{l} and\ (branch(A \wedge B, 1, rigid, \\ \quad\quad\quad [branch(\{\,\} \wedge B, 2, rigid, \\ \quad\quad\quad\quad [rule((\{\,\} \wedge \{\,\}) \rightarrow \{\,\})])]), \\ \quad\quad branch(A \wedge B, 2, rigid, \\ \quad\quad\quad [branch(A \wedge \{\,\}, 1, rigid, \\ \quad\quad\quad\quad [rule((\{\,\} \wedge \{\,\}) \rightarrow \{\,\})])]))) \end{array}$$

With f and g defined as in example 1.4, the computational behaviour for a call like $f(X) = 4 \wedge g(X) = 7$ would be to try to reduce $f(X)$ first. As $f(X)$ will be delayed (remember, it is defined as rigid), a second attempt will be made by evaluating $g(X)$ first, which will be succesfull.

1.5 Conclusion

Out of this chapter, three ideas should be kept in mind:

1. Curry is a functional language,
2. yet it is also a logic language.
3. but, finally, the exact computational behaviour for some function is uniquely described by means of a definitional tree associated to it.

Chapter 2

Monadic I/O within Curry

How can we provide Curry with interaction?

In logic programming languages, no real interest ever truly existed in theoretic foundations concerning I/O. This surely explains why most of the time imperative languages are used to implement the interaction part, whereas only the core of the program is truly built in a language like Prolog.

In the field of functional programming though, intensive research has been going on about how to provide pure functional languages with I/O in such a way that no aspect of the declarative character of the language is lost. But it all isn't as easy as it might seem:

> ...fitting action into a purely functional language feels like fitting a square block into a round hole. [21]

For some time, dialogues and continuations where thought to be the ideal solutions: interaction is provided and declarativeness preserved. But a price had to be paid, as the manipulation of these constructs was quite tiresome.

Then a more elegant solution came up: monads.

This chapter will be organised as follows. Philip Wadler has produced some excellent articles about the use of monads within pure lazy functional languages [23, 25, 21, 22]. The first section will somehow summarise the ideas underlying these articles and take a look at what monads are and how they are used (within the context of Haskell [20]). The next section will expose how to introduce these constructs into Curry, viewed as a functional language. Due to the logic nature of Curry though, a major problem of introducing I/O will be encountered: non-determinism.

2.1 What are monads?

In functional languages an expression *denotes* a specific value, whereas an I/O command should *perform* an action. Yet these two worlds, the world of values, and the world of actions, can be united by introducing a new type `IO(a)` which *denote* actions, that *when performed*, may possible do some I/O and return a value of type `a`. The accent here is mainly on "*when performed*" as will be made clear further on.

2.1.1 Primitive operations

Let's consider the following primitive I/O operations:

```
getChar :: IO(Char).
putChar :: Char -> IO().
```

The operation `getChar` denotes the action which, when performed, will read a character from the standard input and will return that character. On the other hand, `putChar(a)` denotes an action which, again when performed, writes the character `a` to the standard output. As can be seen from the typing of `putChar`, it will return no specific value (i.e the empty tuple).

But how is a command ever performed? The only possible way is by defining a top-level function that specifies the command to be performed by the program. In Haskell this is done with the variable `main`. Suppose the following program:

```
main :: IO().
main = putChar("H").
```

When executing this program, the character "H" will be printed. So in order to get actions performed, the type of the program must be `IO()`, which means that it does not need any parameter as input, and it will not produce any output value. This is quite logic, once a program is performed, what could we do with a possible output value?

2.1.2 Composing I/O operations

So we've seen some basic primitive operations, but as for now this represents a rather poor play-garden for real programmers. We need to provide some glue-material which allows to combine these basic primitive operations. This is done by the following combinators.

Haskell is a lazy language, and when performing I/O one would rather wish that the actions to be undertaken be exactly performed in the order predefined. So we need a first combinator that sequentialises the actions denoted by the primitive operations.

```
(>>) :: IO(a) -> IO(b) -> IO(b).
```

If `f` and `g` are commands, then `f >> g` denotes the command that, if it ever gets performed, will first do `f`, and then `g`. So the following program

```
main :: IO().
main = getChar >> putChar("H").
```

will read a character from the standard input, discard the value obtained, and print “H” on standard output.

Most of the time though, we do not really wish that the value obtained by some first action is discarded, before some second action can be performed. So a second combinator must be provided, which truly binds two commands.

```
(>>=) :: IO(a) -> (a -> IO(b)) -> IO(b).
```

Now if `f` and `g` are commands, then `f >>= g` denotes the command that, only if it is performed, will first do `f`, which will yield some value `x` of type `a`, and will then do `g(x)`, which will yield a value `y` of type `b`. If `main` is defined as follows:

```
main :: IO().
main = getChar >>= putChar.
```

then the program will read a character and print this same character to standard output.

Do we need some other basic combinators? Well, yes. One could imagine situations where some specific instantiation of a function should, when performed, just do nothing. With the combinators introduced so far, the action of doing nothing cannot be expressed. We therefore introduce two new combinators.

```
done   :: IO().
return :: a -> IO(a).
```

The first command, whenever performed, just doesn’t do anything at all. The second command does not do any action either, it just yields the value with which it was called. Here are some examples illustrating the use of the latter combinators.

```
putString           :: list(Char) -> IO().
putString []        = done.
putString [ch|str] = putChar(ch) >> putString(str).
```

Whenever performed, the command `putString(["H","i","!"])` will just produce as output: “Hi!”. A second interesting example is `echo`, which just echoes the input to the output (`\x -> e` is the notation used for lambda expressions).

```
echo :: IO().
echo = getChar >> \c -> if (c==eof)
                           then done
                           else putChar(c) >> echo.
```

2.1.3 So, what is a monad?

> Monads are essentially just another abstract data type. [25]

The data type `IO(a)` can be simply viewed as a function which takes as its input a value representing the entire current state of the world, and returns a value together with a new state of the world. This could be represented as:

```
type IO(a) = World -> (a,World).
```

But we do not want a programmer to access this world directly (as would be the case when providing I/O by means of side-effects). We therefore impose a level of abstraction. The only way to access the world is by using some primitive operations (like `getChar`, `putChar(a)`) which can be combined using the pre-defined combinators. And finally, the only way to actually get some action performed is to define a top-level function of type `IO()`, that defines what has to be done.

The use of monads is not only restricted to I/O. Many "impure" features can be made pure by the use of corresponding monads: state, exceptions, and others [24].

2.1.4 Equational reasoning

One of the basic characteristics of pure functional languages is equational reasoning ("referential transparency", "rule of Leibniz"), which just means: substituting equals for equals.

In standard ML [16], a non-pure functional language, I/O is performed by means of side-effects. The following example illustrates how equational reasoning is invalid in SML.

```
let val x = output(std_out,"H") in x; x end
```

The expected output would be "HH", as one can assume that `x` is bound to the action to produce an output, and `x` is called twice within the `let` construct. But this program will produce only a single "H". More specifically, this output will be done at the time `x` is bound.

An analogous construction in Haskell, using monadic I/O would be:

```
main = let m = putChar("H") in m >> m.
```

Here `m` denotes the action to print the character "H". And in effect, when main is executed, `m` will be performed twice, producing the expected output. Hence, equational reasoning is preserved.

2.2 Monads in Curry (functional point of view)

Considering Curry for the moment as a pure functional language, it presents itself just like Haskell: strong typing, higher-order features. This is all what monads are in fact based on (actually neither of both are truly required, but it makes the job easier [25]). So from a functional point of view, it is quite straightforward to introduce monads within Curry.

We just need to define a new constructor `io(A)` [1], where `A` can have any type, or is equal to `unit` (just a type which denotes the empty tuple).

The basic combinators (`>>` and `>>=`) can then be implemented in Curry itself. The sequential character of these commands can be assured by imposing that the first argument for each of these must be rigid. The second part of the command is only performed when the first part could get reduced to head-normal form matching `io(A)`.

The two other combinators, `done` and `return(X)`, together with the primitive operations such as `getChar` and `putChar(X)` can be defined as external functions (which implicates that they are all rigid in their arguments).

The definition of the command `echo` becomes in Curry:

```
echo :: io(unit).
echo = getChar >>= \C -> C== -1
                         then done
                         else (putChar(C) >> echo).
```

2.3 Monads in Curry (logic point of view)

For now we only considered Curry as a functional language. But Curry also incorporates the logic programming paradigm, allowing the use of free variables, unification, and therefore non-deterministic search. Consider the following, simplistic example, where I/O is used in combination with free variables.

```
puzzle :: io(unit).
puzzle if isValidPuzzle(P) = printAndSolve(P).

isValidPuzzle :: puzzleType -> bool.
isValidPuzzle eval 1:flex.
isValidPuzzle(puzzle1) = true.
isValidPuzzle(puzzle2) = true.

printAndSolve :: puzzleType -> io(unit).
printAndSolve = ...
```

[1] Note that in Curry only variable values are allowed to start with an upper-case.

The argument of the function `isValidPuzzle/1` is marked as *flex*, which means that, whenever this function is called with it's argument being a free variable, unification is allowed. Now how would the main function `puzzle` be solved? Considering that the `if` formalism is internally translated to the function `cond/2`, for which simply the first argument is marked as rigid, the reduction process for `puzzle` would comprise the following steps:

$$\begin{array}{lll} puzzle & \rightsquigarrow & cond(isValidPuzzle(P), printAndSolve(P) \\ & \rightsquigarrow & \{P = puzzle1\}\ cond(true, printAndSolve(puzzle(1)) \mid \\ & & \{P = puzzle1\}\ cond(true, printAndSolve(puzzle(2)) \\ & \rightsquigarrow & ??? \end{array}$$

And now we have a problem. The second step in the reduction process produces a disjunctive answer expression.

One thing is sure, we cannot output both puzzles at the same time. We can not even produce each of the outputs one after the other. Why not?

In section 2.1.3, we said that in fact, the I/O-monad represents actions on the world: it takes as input value the actual state of the world, and produces as output some value, and a new state of the world. As long as there are no disjunctions in the reduction scheme, then each new produced state of the world serves as input value for the next I/O action. When a disjunction occurs, this would mean that for each of the further actions, the same input world should be considered.

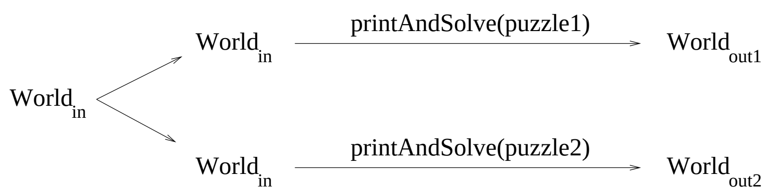

Figure 2.1: Duplicating the world

We would just be trying to duplicate the world. But we only have one. If a first action takes effect on the world, this world is irrevocably changed. $World_{in}$, the initial world for the second part of the disjunction, will not be the true original world anymore, and the declarative character of the language would be lost (as the first actions can be seen as some side-effect, influencing the input parameter, the world, of another function). Therefore such disjunctions must be avoided.

The only valid solution to this problem is to produce a run-time error, as during the reduction two possible solutions are obtained, and only one of them may actually be considered. But a priori none of them is better then the other one.

Do we have to abandon the concept of monadic I/O? Of course not. With any I/O construct, which preserves the declarative character of the language, the same problem would be encountered.

The next sections will discuss how to handle these non-deterministic computation steps, how to avoid them, and finally how to detect them at compile-time.

2.3.1 Handling non-determinism

The previous example can be generalised. Whenever the reduction of a top-level function of type `IO(unit)`, presents the following scheme:

$$\begin{array}{lll} top & \leadsto & \ldots \\ & \vdots & \\ & \leadsto & \{\sigma_1\}\ e_1 \mid \ldots \mid \{\sigma_n\}\ e_n \end{array}$$

a run-time error must be produced. It is not possible to reduce one of the answer expressions without influencing the state of the world, needed by each of the other answer expressions.

This detection at run-time is very simple. If an I/O command produces a disjunctive answer expression, the further evaluation is just halted.

2.3.2 Avoiding non-determinism

The question now arises whether a programmer working on an I/O command, must restrict its code to pure functional programming, avoiding every possible non-deterministic search, the main power of logic programming? Such a restriction would be quite a limitation.

Yet, Curry provides a way to encapsulate search. This means that even when some parts of the program will produce some disjunctive answer expression, only one of the solutions will be presented to the top-level function, hence avoiding run-time errors when the top-level function is of type `io(unit)`.

Therefore, if the programmer explicitly wants to make use of the logic capabilities of Curry, in order to avoid any trouble with non-determinism, he will need to encapsulate the non-deterministic parts of his program.

2.3.3 Detecting non-determinism

Whenever a programmer truly wants to use non-deterministic search, he's better off by encapsulating such a search. But programming errors are always imaginable, and run-time errors could be quite costly. It would therefore be interesting to provide Curry with a debugging tool, which would allow to detect possible non-determinism. Such a detection could also be automatically incorporated into the compiler. The compiler could then produce some informative warnings about some function calls of type `io(A)`, stating that they could possibly be non-deterministic.

In order to build such an analysis tool, we must first clearly specify for what cases a disjunctive answer expression can be obtained. As in Curry, the exact computational behaviour of a defined function, can be represented by definitional trees, we can simply overview each of these definitional trees, and see where non-determinism might occur[2].

2.3.3.1 $\mathcal{T} = rule(l \to r)$

If a specific function is to be computed using a simple rule, then this function will only be non-deterministic if the right-hand side of the function is non-deterministic.

2.3.3.2 $\mathcal{T} = branch(\pi, o, r, [\mathcal{T}_1, \ldots, \mathcal{T}_n])$

The meaning of a *branch*-node is, that whenever a function appears at the specified position in the calling function, this function must be evaluated first. If the result is in head-normal form, then further computation may be done using the definitional tree that has a pattern that matches this head-normal form. If the result is a free variable, then we distinguish two cases: if the node is marked as *rigid* then the function is suspended, if it is marked as *flex* then a disjunctive answer expression is obtained, where this free variable is unified with each of the possibilities defined in the further definitional trees.

Thus, the evaluation of a function using a *branch*-node, is non-deterministic if one of the following holds:

1. the sub-function appearing at the specified position is non-deterministic.
2. the expression appearing at the specified position is a free variable, whereas the node is marked as *flex*, and there is more then one definitional tree ($n > 1$).
3. the further evaluation of the function using some sub-definitional tree $\mathcal{T}_i$ is non-deterministic.

Branch-nodes marked as *rigid* cannot directly lead to non-determinism, as no free variables are allowed (the goal gets suspended), and no unification occurs. If the argument at the marked position is in head-normal form, then too, no direct non-determinism can take place (independently of the exact annotation).

[2] In this discussion we will omit the types of the functions, as they do not really matter here.

If the *branch*-node is set to *flex*, and if there is only one definitional tree defined in it, no direct non-determinism will occur either, as there will be only one value to which the free variable can be unified. The following example illustrates this.

```
f eval 1:flex.
f(1) = true.
```

The definitional tree corresponding to this function is:

$$branch(f(X), 1, flex, [rule(f(1) \to true]).$$

Using this definitional tree, the reduction of a call like $f(X)$ with X a free variable will simply be $f(X) \rightsquigarrow \{X = 1\}\ true$, which is deterministic.

2.3.3.3 $\mathcal{T} = or(\mathcal{T}_1, \mathcal{T}_2)$

An *or*-node is the most direct source of non-determinism, as it states that for some values for the arguments of a given function, at least two possible ways of further computation exist. Here a simplistic example:

```
f(1) = 0.
f(X) = X.
```

The evaluation of a call like $f(2)$ will be: $f(2) \rightsquigarrow 2$, whereas the call $f(1)$ will produce a disjunction: $f(1) \rightsquigarrow 0 \mid 1$.

2.3.3.4 $\mathcal{T} = and(\mathcal{T}_1, \mathcal{T}_2)$

An *and*-node, introduces no direct problem with non-determinism. It simply states that only if the evaluation using $\mathcal{T}_1$ should get suspended, then $\mathcal{T}_2$ should be tried instead. Of course, the evaluation of a function using an *and*-node-based tree, will yield non-determinism, if the subtree used leads to some disjunction.

2.3.3.5 Overview

We can summarise the sources of non-determinism to only three possible cases. A function is non-deterministic if one of the following holds:

1. during computation, a free variable appears at a *flex* position, and more then one possible unification exists.
2. during computation, an *or*-node is encountered.
3. one of the functions through which the main function is defined, is non-deterministic.

2.3.3.6 Building an analysis tool

If the *or*-nodes would have been the only source of non-determinism, then the detection of it could have simply consisted of checking if one of the definitional trees the main function depends upon contains such an *or*-node.

But a second source of non-determinism exists, which depends on the way some functions are called. The only way to analyse a function at compile-time consists of trying to mimic what might happen at run-time.

This act of mimicking is also known as "abstract interpretation" or "static analysis", which is what the subsequent chapters will be about.

2.4 Conclusion

The use of monads within the functional community is already widespread. They allow the manipulation of I/O, state, ... within pure functional languages, without any loss of the declarative character of the languages. Curry is partly a functional language, which makes it easy to introduce monads in Curry as well. But we have seen that, due to the logic nature of Curry, non-deterministic search at the top-level must be avoided, as we cannot duplicate the world. The easy way to handle non-determinism if it occurs, is just to produce run-time errors. It would be more interesting though to detect possible non-deterministic computation steps at compile-time. In order to do this we need to mimic concrete computations. This can be done using the technique of abstract interpretation.

Just like the field of functional logic languages is relatively new, so is the field of abstract interpretation within these languages. The next chapters (chapters 3, 4, 5, 6) are therefore concerned with the construction of a generic abstract interpretation framework for Curry. This framework can then be applied to the problem of determinism (chapter 7).

Chapter 3

A top-down abstract interpretation framework for logic languages

The theory of abstract interpretation, firstly introduced in the context of imperative languages [6] and now widely adapted to declarative languages, allows to verify run-time properties of program parts, without actually executing these programs. Such properties are useful for debugging, code optimisation, program transformation and program correctness proofs. Techniques of abstract interpretation find their application in groundness analysis, mode analysis, sharing analysis, and many other domains.

In this chapter some general concepts about abstract interpretation will be presented. The discussion is mainly based on [3], where a general abstract interpretation framework for logic programs is developed. This framework, by its generality, its simplicity and its elegance in proving the correctness of the collected information, will be the basis of the abstract interpretation procedure that will be constructed in the next chapters.

A good tutorial introducing abstract interpretation can be found in [4].

3.1 Central Ideas

Suppose a program P which works on a set of data D. Suppose we would like to verify some property S of this program. One way of doing this is to run that program for all data-values within D. If D contains few elements, and if the cost of running the program is acceptable, then this verification technique could indeed be applied. When the amount of elements in D gets bigger, or even becomes infinite, this technique is not usable anymore.

The idea of abstract interpretation consists of translating the possibly infinite set of concrete data values D into some finite set of descriptions of the elements in D (abstract data values). The operations on those concrete values must also be translated into some abstract operations which deal with these abstract data elements. We then get some sort of abstract version of the initial concrete program. This translation is correct if the results of running the 'abstract' program correspond with the results which can be obtained with the concrete program.

Schematically we have:

$$\begin{array}{ccc} \text{Concrete Programs} & \rightarrow & \text{Abstract Program} \\ \downarrow & & \downarrow \\ \text{Concrete Results} & \stackrel{?}{\leftarrow} & \text{Abstract Results} \end{array}$$

The hand-crafting approach of this translation work would be to transform each new program into some abstracted version of it. The disadvantages are obvious.

A more interesting approach of the abstraction process is to translate the underlying semantics of the language used. Once this work is done, the abstract interpreter so constructed can be used to analyse each new possible program without any extra overhead. Therefore, the process of abstract interpretation can be thought of in terms of the following steps:

- determine the concrete operational semantics of the programming language
- construct an abstract domain which covers the possible concrete values of the data used, and which highlight the properties that are to be checked for
- translate the concrete operational semantics into some abstract operations which deal with the abstract data values instead of the concrete ones
- build the abstract interpreter

The further discussion in the context of logic programming languages will follow these same steps.

3.2 Concrete computation

The most common computation rule used within logic languages is depth-first, left-to-right (Prolog). This means that a given query is reduced according to some simple steps:

1. Finding a clause of which the head can be unified with the given goal.
2. After unification, all the literals in the used clause are processed one after the other. Each literal will be based on the bindings made in the earlier steps.

3. After computing all the literals, the obtained bindings are sent back to the original goal.

In this computation scheme we can clearly distinguish three main operations which will occur for every call:

1. Entering a clause body: this operation consists of translating the initial information of the variables to information of variables contained in the body of the selected goal.
2. Unification: Unification, when applying a goal, is in fact an implicit operation. The explicit version is also present in the language: $X = Y$. Operations such as these are not defined by some clauses, but are part of the language (built-in predicate).
3. Exiting a clause body: by computing each of the literals, one obtains new instantiations of the occurring variables. These bindings must be retranslated in terms of the original variables occurring in the goal.

These basic operations are repeated for each possible goal that is to be performed. They are independent of the exact definitions of the predicates, and therefore they can be considered as the essence of the language. With the aid of these basic building blocks of the operational semantics of the logic language (here, Prolog), one can present the computation of a goal as in figure 3.1.

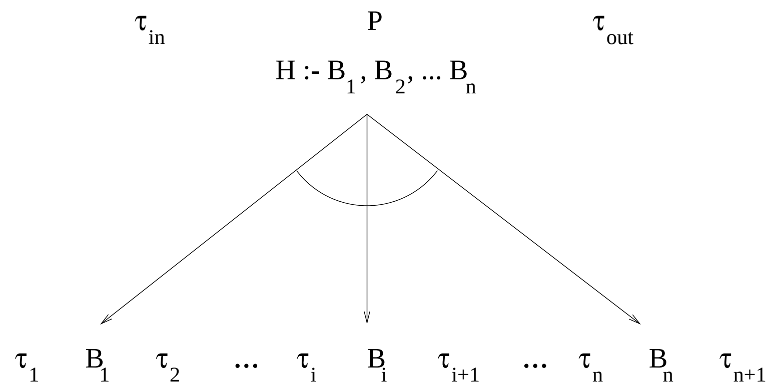

Figure 3.1: solving a predicate P with a clause $H : -B_1, \ldots, B_n$

The AND-node means that each of the literals is processed one after another. The symbols τ_i adorning the goals on the left, represent the concrete substitutions of the variables at the time the goal is called. The substitutions of the variables at the end of the computation are written on the right side of goals.

3.3 Abstract computation

Abstract interpretation consists of translating all these basic operations in terms of abstract data, instead of the standard semantics. In order to do so, we'll have to:

1. define the abstract data which will be used
2. effectively translate the basic operations in terms of these
3. construct a general analysis algorithm based on the abstract operations

3.3.1 Abstract domain

Central to the abstract interpretation process is the abstract representation of the data used. The exact definition of the abstract values depends on the information one wishes to collect from a program. As the discussion in this chapter is aimed to be as general as possible, we will present some general characteristics of the abstract domains. These characteristics are specifying enough in order to allow further construction of the abstract interpretation process without actually determining some specific abstract domain.

Definition 3.1 (Substitution)
In Prolog, the data carried around consists of the concrete instantiations of the variables used. A concrete instantiation of variables at some program point is represented by some *substitution* $\theta = \{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\}$ where $\{X_1, \ldots, X_n\}$ are distinct variables.

Definition 3.2 (Abstract substitution)
In logic programs one wishes to make abstraction of the concrete substitutions. Therefore the concept of *abstract substitution* is introduced. A particular abstract substitution collects the interesting information one wishes to gather about some set of concrete substitution.

Definition 3.3 (Concretization function)
The meaning of a particular abstract substitution β is given by the *concretization function* $\gamma :: A \rightarrow D$. $\gamma(\beta)$ represents all the set of concrete values that the abstract substitution β may cover. If β is an abstraction for the substitution θ we have: $\theta \in \gamma(\beta)$.

Figure 3.2 illustrates all these concepts.

Definition 3.4 (Precision, Order relation)
The *precision* of an abstract substitution is related to the size of the set of concrete substitutions it represents. An abstract substitution β_1 is said to be more precise than an abstract substitution β_2 if the set of variables it

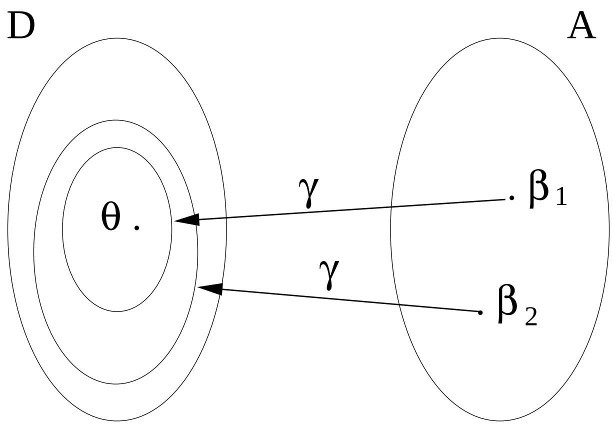

Figure 3.2: Graphical representation of the concrete domain (D) consisting of concrete substitutions and the abstract domain (A), the set of abstract substitutions. γ represents the concretization function relating both sets.

covers corresponds with the set covered by β_2, and if the set of concrete substitutions it represents is smaller than the set represented by β_2. More formally:

$$\beta_1, \beta_2 \in A : \beta_1 \sqsubseteq \beta_2 \Leftrightarrow \gamma(\beta_1) \subseteq \gamma(\beta_2) \tag{3.1}$$

Definition 3.5 (Complete lattice, $\perp$, $\top$)
The set A of all abstract substitutions is mostly required to be a *complete* lattice. For each set $B \subseteq A$, A must contain its infinum and its supremum. This implies that the set A will also contain its own infinum (called *bottom*, also written as $\perp$), and its own supremum (called *top*, also written as $\top$). The meaning of these extrema is obvious: while $\perp$ represents no concrete elements at all, $\top$ on the other hand represents the set of all possible concrete substitutions over the given set of variables.

Definition 3.6 (Least upper bound)
Based on the ordering relation 3.1 a function calculating the *least upper bound* of a set of abstract elements can be defined. We have: $\beta_1, \beta_2 \in A$: $lub(\beta_1, \beta_2) = \beta$ such that β is an upper bound for β_1 and β_2, and for each upper bound β_i^{up} for β_1, β_2 we have: $\beta \sqsubseteq \beta_i^{up}$.

The use of the operations such as *lub*, will be made clear when defining the primitive operations within the abstract interpretation model.

Note that just as a concrete substitution θ is meaningless without the association to the call to which it belongs, the abstract substitution is in the same manner meaningless. Although the concept of abstract substitution was introduced here based on the concrete substitution, the abstract information it contains can be of any possible kind.

3.3.2 Primitive operations - Abstract AND-OR tree

A concrete computation can be clearly described by the steps presented earlier: applying a goal, exiting a goal, eventually applying a built-in predicate. These are also the procedures that will be needed within an attempt of abstract interpretation. But their definition must be somehow extended.

In most of the cases, a predicate is not defined by a single clause. At run-time it is known which of the clauses will be applied. In the abstract domain though, due to the abstraction of the data used, one must consider that each of the clauses is a potential clause with which the given goal can be resolved. This leads us to the concept of (abstract) OR-nodes. Such nodes simply translate the following idea: each of the branches represents a potential clause that can be used at run-time.

We can now define the previously determined primitive operations within the context of static analysis.

1. Entering the analysis of a user-defined predicate, called *procedure entry operation*: it deals with
   - constructing an OR-node where each of the branches corresponds to a clause defining the predicate
   - constructing for each of the branches an AND-node, where each branch consists of a literal appearing in the clause body
   - translating the given initial abstract substitution over the initial variables of the goal, to a substitution over the variables appearing in the bodies of the clauses.
2. Exiting the analysis of a user-defined clause, called *procedure exit operation*: after the analysis of each of the possible clauses of the OR-node, the information collected at each of the branches must be summarised in terms of the original variables. This is done by applying the least upper bound operation on all of the exit abstract substitutions obtained.
3. Abstract interpretation of built-ins: some predicates are built-ins, i.e. no clauses exist which define the predicate, but the system computes results according to the internal definition of the concerned predicate. Unification is such a built-in mechanism, but also more common operations such as I/O, arithmetics, etc... are usually built into the language. Each of these built-ins must be abstracted separately.

The abstract interpretation procedure can then be represented as in the figure 3.3.

A general abstract interpretation algorithm can then be written as follows. Given a program P, and a call Q with an abstract substitution defined

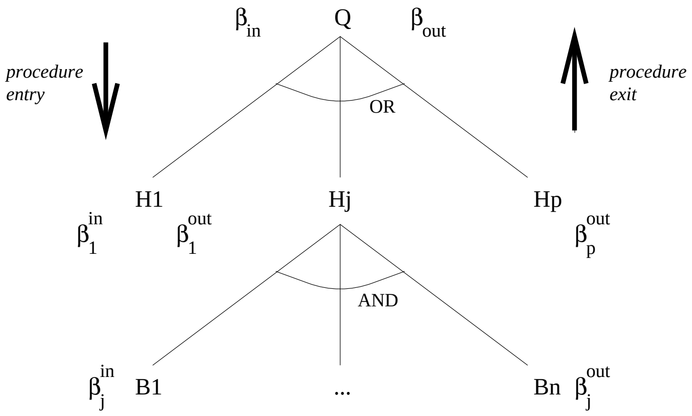

Figure 3.3: abstract AND-OR tree

β_{in} on it, the analysis of Q yielding a resulting success substitution β_{out} is performed as follows:

- Q is a built-in predicate: perform abstract interpretation of the built-in.
- Q is a user-defined predicate:
  - apply procedure entry
  - apply abstract interpretation on each of the literals appearing in the bodies of the clauses defining the predicate
  - apply procedure exit

3.3.3 Fix-point computations

The algorithm presented is totally useless if no extra care is taken with recursive functions. Indeed, mimicking concrete computation by just using the previous procedures on abstract substitutions would result in a non-terminating process, as for each predicate all possible clauses are considered each time.

The way to handle recursive calls is done by what is called fix-point computation. The principle is quite simple. When at some stage a recursive call P relative to a previous call P' is performed which is instantiated with the same abstract substitution β as the one adorning the initial call β', then instead of processing the predicate just like a normal predicate (and resulting in non-termination), an approximation of the result substitution is used (often the first approximation is set to the bottom element of the abstract domain). Once computation leads us back to the initial call P', we've calculated a valid approximation of the call, and restart the abstract interpretation of P. Now, when arriving at P', this latest approximation is used. Each time, repeating the process updates the calculated abstract success substitution. When two consecutive updates are equal, then the process is stopped, and the resulting abstract substitution is calculated.

Due to the monotonicity of the calculations and assuming a finite abstract domain, one can guarantee that the process will terminate, and that the abstract substitution calculated will be correct, i.e. it is a representation for all possible concrete substitutions that can result by a concrete computation.

The same philosophy will be used when analysing functional logic programs.

3.3.4 Static analysis algorithm

With the concepts defined earlier, we can now define the general top-down procedure with which each possible predicate can be performed abstractly. Suppose a predicate P, with an initial abstract substitution β_{in} defined on its variables.

1. P is a built-in. Apply abstract interpretation of the built-in. A success abstract substitution β_{out} is calculated.
2. P is user-defined, and does not appear within the previous steps of the analysis. (non-recursive call).
   (a) apply procedure entry
   (b) apply abstract interpretation on the calls appearing in each of the bodies. Within a clause body, the calculated abstract success substitution for a literal becomes the abstract call substitution for the literal appearing on the left of the previous one.
   (c) apply procedure exit: summarise the collected information for each of the OR-nodes, in terms of the calling abstract substitution β_{in}. A success abstract substitution β_{out} is calculated.
3. P is a recursive call. If the calling abstract substitution relative to the variables within P, is identical to the calling abstract substitution of the ancestor call P', then a fix-point calculation is started. Again, an approximate success abstract substitution β_{out} is calculated.
4. P has an ancestor node with a call to the same predicate as P, but their calling abstract substitutions differ, for simplicity's sake, proceed as in 2[1].

[1]This simplification is only possible if the abstract domain is not infinite. As the abstract interpretation process for Curry will rely on finite abstract domains, a true discussion about infinite domains is not necessary.

3.4 Correctness of the abstract analysis procedure

An abstract interpretation process is said to be *correct* if at each program point during the analysis, the abstract information collected correctly abstracts the concrete information that would be present at the same program point when running the program in the standard domain.

The elegance of the presented approach to abstract interpretation relies in the theoretical simplicity of the correctness proofs. Indeed, the analysis algorithm itself can be proven to be correct, regardless of the exact abstract domain used. Based on sufficient conditions which can be defined for the abstract primitives, the algorithm can be proven once and for all.

When applying the abstract interpretation process to some particular instance of the abstract domain, the only proofs to be provided are the proofs that the primitives, with regard to the abstract domain, fulfil these sufficient conditions.

The details of the correctness proofs and the correctness conditions are not reviewed here, as the correctness of the algorithm used for analysing Curry-programs will have to be given explicitly.

3.5 Conclusion

The theory of abstract interpretation consists in fact of two main tasks:

- find a suitable abstract domain which collects the necessary information.
- adapt the procedural semantics of the language in such a way that it can now deal with the abstract information instead of the concrete one.

This chapter mainly relied on the abstract interpretation framework developed for logic programs. The concrete computation of logic programs can be summarised to some elementary computation steps working on concrete substitutions. In the framework the concrete computations are translated into abstract AND-OR trees. The information flow along these trees is directed by some primitive operations which act on abstract substitutions: the primitive operations *procedure entry*, *procedure exit.* Special care is taken with built-in predicates. In order to be able to analyse recursive functions, we must ensure that in these cases the abstract interpretation process terminates. To enforce this, a fix-point calculation must be provided.

In the following chapters the same concepts of abstract substitutions, abstract AND-OR trees, primitive operations, fix-point computations are going to be adapted within the context of functional logic programs.

It is important to notice the top-down character of this abstract interpretation framework. Information regarding a particular goal is restricted to the variables needed when applying a procedure entry operation. All information about other variables is discarded. In an analogous way, all information around variables that are not initially present in the calling abstract substitution is discarded when applying the procedure exit operation on the calculated abstract success substitutions of the clause bodies. There is a strict limit. Beyond this limit, information is filtered in order to restrict it to what is needed. Indeed, letting through information about variables that are not needed in a direct way, allows the domain of the abstract substitutions to become infinite, i.e. the set of variables on which the abstract substitution is defined could grow infinitely. In the case of such infinite domains, program points where fix-point computations should be performed would be far more difficult to recognise.

This aspect of the top-down analysis will have to be respected in the analysis of functional logic programs too, and will result, as will be shown, in some clear imprecision of the analysis.

Chapter 4

Abstract Interpretation in Curry - Generalities

The starting point for the static analysis of Curry-programs is the abstract interpretation framework for logic programs[1] [3]. Essentially there are two ways of dealing with the abstract interpretation problem for this language:

1. transform Curry-programs into equivalent logic programs, and then apply the basic framework in a straightforward way

2. transform the framework so that it can deal with Curry-programs

In this text, some sort of middle way is chosen.

In this chapter some first thoughts about how to adapt the Curry programs or the framework are given. As will be shown, the main transformation of the Curry-programs consists of simply flattening them, whereas the framework is adapted so as to construct dynamically the requested abstract AND-OR nodes. These adaptations unfortunately introduce some inevitable loss of precision.

To make the presentation more understandable, it will be based on a particular application of abstract interpretation, namely groundness analysis. The abstract domain required for this is in fact the same as the one which will be used in the context of determinism analysis (chapter 7).

Initially, I planned to provide a formal proof of the correctness of the abstract interpretation framework which will be developed here and in the next chapters. Due to lack of time though, my initial goals had to be kept modest, and such proofs will not be given in this text. The correctness proofs should be similar to the ones concerning the abstract interpretation framework for logic languages. In a first stage one has to prove that concrete evaluation schemes are covered by the abstract AND-OR trees that the

[1]The discussion presented here is done with regard to simple logic languages, which don't use any concept of delay.

analysis constructs. Considering that the primitive operations are correct (i.e. sufficient conditions of correctness are provided), one should be able to prove that the data flow along the abstract AND-OR tree will be correct. Finally, when applying the framework to some particular abstract domain, the correctness of the analysis can be demonstrated just by proving that the primitive operations with regard to this abstract domain do satisfy the sufficient conditions of correctness.

4.1 An example - Discussion

The functional logic language Curry and logic programming languages are fundamentally different. As the first one copes with functions, logic programs know only predicates. Functions can be nested while predicates cannot.

But the differences not only reside in the "syntax". In order to execute a goal, all literals appearing in the body of the clause used, will be executed too (if the goal is to succeed, of course). Moreover, the exact order in which they are performed is exactly defined by the order in which they appear in the clause (in Prolog this order is depth-first, left-to-right). When a given Curry rule is applied, one can not be sure that all the functions appearing in the right-hand side will be reduced, neither does one know the exact followed sequence of reduction steps. As the language Curry incorporates a lazy evaluation mechanism, we can not even determine exactly when a given expression will be evaluated or if it will be evaluated at all.

All these differences will be presented here based on an example where groundness information is to be collected. We will first define an abstract domain, and based on it, derive possible abstract interpretation scenario's.

4.1.1 Abstract domain

Collecting groundness information is a typical example of analysis performed on logic programs. To make the example as simple as possible, let us assume that the abstract information β over a set of variables $\{X_1, X_2, \ldots, X_n\}$ is either $\perp$ or a set of assignments of the form: $X_i := m_i$ where $m_i \in \{g, a\}$, where g and a are the abbreviations for respectively "ground" and "any"[2].

The abstract substitutions are given a meaning by the concretization function γ defined as follows:

- $\gamma(\perp) = \emptyset$ (the empty substitution).
- a concrete substitution $\theta = \{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\} \in \gamma(\beta) \Leftrightarrow$
  - $(\text{var } \beta) = (\text{var } \theta)$

[2]The more common representation of such groundness information is $\{X_i \mid X_i := g\}$

- if $X_i := g \in \beta$ then t_i is a ground term.
- if $X_i := a \in \beta$ then t_i is any possible term.

The ordering information between the assignments of a variable in two different abstract substitutions β_1, β_2 is given by the order relations:

$$\begin{array}{ccc} \perp & \leq & g \\ \perp & \leq & a \\ g & \leq & a \end{array}$$

The top element $\top$ is simply the abstract substitution where each variable is assigned the value a. From this relation the least upper bound operation can be easily derived.

4.1.2 Example Curry program

The discussion which will be presented throughout this chapter will be mainly based on this extract of a Curry program:

$$f(X,Y) = g(h_1(X,Y), h_2(X,Y)) \tag{4.1}$$

where g, h_1, h_2 are functions, and X, Y are possible arguments.

4.1.3 Aim of the analysis

The information we would like to collect has three parts:

1. Information about the way the different sub-functions will be called.
2. Information about the state of the variables after a function has been calculated.
3. Information about the possible results of the functions.

4.1.4 Some evident differences

By just looking at the example program, we can already distinguish some clear differences with logic programs which will have to be coped with.

Difference 1 Functions have results[3].

Difference 2 Functions can be nested.

[3] At each stage of the discussion a review is presented of the notable differences. These differences are stated with regard to Prolog.

Functions, unlike predicates, have results and can be nested. By introducing new variables we would tend to rewrite rule 4.1 in a Prolog-like form:

$$f(X,Y,R) \leftarrow h_1(X,Y,R_1), h_2(X,Y,R_2), g(R_1,R_2,R). \tag{4.2}$$

Life would be so easy if such a transformation could be applied. Not only could the problem of results be solved, but with such a transformation, the problem of nested function calls would be cleared too. The general difficulty of analysing functional logic programs would be reduced to the simple analysis of the logic version of the program.

If a strict innermost, left-to-right computation rule would be at the heart of the language, then indeed, such a flattening could be appropriate. But Curry is based on an extension of needed narrowing. The following subsection lists some operational differences which make clear that a straightforward flattening like the one presented is not allowed.

4.1.5 Operational differences

In a logic program, it is always clear which literals will be selected, as well as when they will be reduced. When a call $f(X,Y,R)$ is executed, the application of clause 4.2 would mean that, unless the call is unsuccessful, all the literals appearing in the body will be checked. Just by looking at the clause we can predict in what order the literals will be executed: simply left to right (Prolog).

Curry's computation rule is not that simple.

4.1.5.1 Ordering

Deriving run-time properties means looking at how functions are called, what instantiations of the variables they are called with, what resulting instantiations are obtained.

Suppose we know the following information about the calls to the functions h_1 and h_2[4], where β_{in} is the calling abstract substitution, and β_{out} is the success abstract substitution:

	β_{in}		β_{out}	
	X	Y	X	Y
$h_1(X,Y)$[5]	a	a	a	a
	a	g	g	g
	g	a	g	g
$h_2(X,Y)$[6]	a	a	g	a
	a	g	g	g
	g	a	g	a

[4]For the time being we'll ignore here the results of the functions.

Suppose the function g requires the first argument to be evaluated first, and only afterwards the second. We then know what the instantiations of the variables of h_1 and h_2 will be, at call time as well as at at exit time. The next figure summarises this information.

$$\begin{array}{ccccc} X/a & & X/a & & X/g \\ & \xrightarrow{\quad h_1 \quad} & & \xrightarrow{\quad h_2 \quad} & \\ Y/a & & Y/a & & Y/a \end{array}$$

Figure 4.1: $h_1 \to h_2$

When on the contrary, the analysis is done in the reverse order, we obtain a total new image of how each of the functions is called.

$$\begin{array}{ccccc} X/a & & X/g & & X/g \\ & \xrightarrow{\quad h_1 \quad} & & \xrightarrow{\quad h_2 \quad} & \\ Y/a & & Y/a & & Y/g \end{array}$$

Figure 4.2: $h_2 \to h_1$

Which order is the correct one? A priori both versions are correct. The central question is: in which order should the analysis be performed? As we can see, the precision of the results obtained can be strongly influenced by this factor. The idea which will be followed throughout the text is to let the analysis follow as closely as possible the intended order in which arguments should be reduced in the concrete domain. Later we will see, that this intended order can be partly deduced from the definitional trees of the functions. We will also see that due to the lazy character of the language, the order is not always so obvious.

4.1.5.2 Not sufficiently instantiated variables

Curry allows a combination of residuation steps and narrowing steps. For residuation steps to succeed, the variables must be sufficiently instantiated.

Let the function h be defined by:

$$h \,\mathrm{eval}\, 1 : rigid.$$
$$h(true) = true.$$

At run-time, a call like $h(X)$ $\{X$ being a free variable$\}$ would be suspended as the argument is required to be instantiated. How should the

[5] e.g. $h_1(X,Y) = X = Y$.

[6] e.g. $h_2(X,Y) = X = 1$.

analysis in the abstract domain deal with such a situation? The answer to this question depends on what the purpose of the abstract interpretation is, and therefore on what abstract domain it relies.

If the abstract domain used would contain some information on whether a variable is free or not (freeness analysis), we could deduce for sure that a call to h with its argument marked as free, would never be reduced. The result of the analysis could then be $\perp$ stating that there exists no exit substitution for this particular goal, as the reduction of the goal is not possible at run-time.

But even with a more complex abstract domain such as freeness, the analysis could encounter a situation where the same function h is annotated with the abstract substitution $\{X := a\}$. Figure 4.3 shows the only correct way to interpret this case.

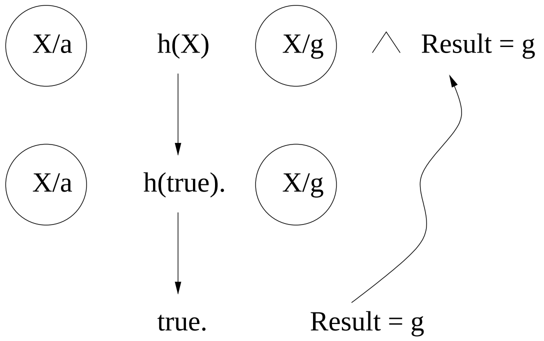

Figure 4.3: Abstract interpretation of $h(X)$

Indeed, it is not known in advance whether at run-time the function will be reduced or not. There are two possible solutions. The first one would be the analysis to return $\top$ as answer, expressing a "don't know" situation. But this would lead to a huge loss of precision. The second solution is the one exposed in the figure: analyse the function as if it would be reduced at run-time. The interpretation of the result obtained in the example then becomes: if at run-time h is successful, we know that X, it's argument, as well as the result, will be ground.

To summarise the situation, and supposing that the actual abstract domain used contains some mode information, we can distinguish three possible cases, as shown in following table.

X	Behaviour
marked as "ground"	proceed as normally, by just analysing further the given function.
marked as "any"	idem.
marked as "free"	$\beta_{out} = \perp$.

Difference 3 Unlike predicates, which can either fail, or succeed[7], function calls get suspended when some arguments appear not to be sufficiently instantiated. The correct way of dealing with such situation in the abstract domain, depends on the information available on the particular variables.

4.1.5.3 Definitional tree

Difference 4 Information regarding the exact way a function call is to be processed is described in the definitional trees of the concerned functions.

In Curry, all relevant information about the way a function is to be processed is described in its definitional tree.

For example, when at some stage a function definition is given by a *branch*-node, where some position is marked as rigid, the expression appearing at that position in a call to this function must be reduced first.

Suppose g's definitional tree is marked as:

$$branch(-, 1, rigid/flex, [branch(-, 2, rigid/flex, [\ldots]), \ldots]) \tag{4.3}$$

then the flattened version of 4.1, could indeed be clause 4.2. And we could build an abstract AND-tree just as it is known from the logic programs.

So, one might think that with the definitional trees, we can simply rewrite all the rules into some Prolog clauses, and construct appropriate abstract AND-OR trees. Again, it is not as easy as it seems.

4.1.5.4 Laziness

When all positions of all the functions are marked explicitly as rigid or flex, then in effect, it is possible to predict the exact order in which the appearing expressions will be reduced. But in general, some positions may be lazy. This means that expressions appearing at the concerned positions are only evaluated when their results are actually needed, if they are needed at all. In section 4.1.5.1 we've already pointed out that depending on the analysis order, we might get different results. Due to the lack of information we have on the actual order that will be followed at run-time, the analysis must go through all possible sequences. Moreover, the analysis must consider the cases that possibly one, or more of the functions occuring at lazy positions will not be reduced. It is mostly neglecting this factor which can lead to false results, as the next example will show.

Let the function g of our running example (4.1) be defined by:

$$\begin{aligned} g(true, B) &= h(B). && (4.4)\\ h(B) &= true. && (4.5)\end{aligned}$$

[7] As said, this discussion is done with regard to logic languages which do not know the concept of delay.

In this definition we see that if some function would appear at the second position of g, then it's evaluation will be postponed until function h would require it to be evaluated. But in our example the result of h can be calculated independently of it's argument.

Now let's consider the table at page 35, relating the calling abstract substitutions with the exit substitutions for h_1 and h_2. If the analysis of f would assume that both, h_1 and h_2 will be reduced at run-time, in one order or the other, we would, as already shown earlier, obtain as results:

	β_{out}	
	X	Y
h_1 then h_2	g	a
h_2 then h_1	g	g
lub	g	a

Calculating the least upper bound of both abstract substitutions yields the abstract substitution $\{X := g, Y := a\}$. But we know that at run-time, h_2 will never be performed. Actually, the reduction scheme of $f(X, Y)$ will be as follows:

$$\begin{array}{rcl|l} f(X,Y) & \leadsto & g(h_1(X,Y), h_2(X,Y)) & \text{rule 4.1} \\ & \leadsto & g(true, h_2(X,Y)) & h_2 \text{ is reduced} \\ & \leadsto & h(h_2(X,Y)) & \text{rule 4.4} \\ & \leadsto & true. & \text{rule 4.5} \end{array}$$

In this reduction scheme, both X and Y remain free variables. But this concrete substitution is not covered by the abstract substitution we obtained by analysing both sequences, and our abstract calculation is therefore false.

So in order for our analysis to be correct we must not only consider all the possible orders, but also all the cases that one or more of the lazy arguments will not be reduced. We therefore have to complete the analysis with some extra sequences.

	β_{out}	
	X	Y
h_1 then h_2	g	a
h_2 then h_1	g	g
only h_1	a	a
only h_2	g	a
none	a	a
lub	a	a

The result we obtain can be much less precise, but at least it will be correct.

Difference 5 Curry allows lazy calculations.

4.1.5.5 Head normal form $\Leftrightarrow$ normal form

In Prolog programs the computation rule is depth-first. This means that no matter what happens, a literal is always reduced until its truth value is calculated, that is, until the point where no further computation is possible. Such behaviour can occur within Curry when all functions have to be reduced to their normal form.

This is not always the case.

In our example, let:

$$\begin{array}{lcl} g(c(A,B),X) & = & true. \\ h_1(X,Y) & = & c(k(X),k(Y)). \end{array}$$

Where $c/2$ is a constructor, and $k/1$ is just some function.

And the same main rule:

$$f(X,Y) = g(h_1(X,Y), h_2(X,Y)).$$

Supposing all the positions are marked as flex, the concrete call $f(X,Y)$ would yield the expression $g(h_1(X,Y), h_2(X,Y))$ in a first rewrite step. h_1 is the first argument to be evaluated. It is rewritten into $c(s(X), s(Y))$. For the function g this one step is enough in order to proceed, i.e. the result is set to the value "*true*". Summarised, at run-time the following reduction steps will occur:

$$\begin{array}{lcl} f(X,Y) & \rightsquigarrow & g(h_1(X,Y), h_2(X,Y)) \\ & \rightsquigarrow & g(c(k(X),k(Y)), h_2(X,Y)) \\ & \rightsquigarrow & true. \end{array}$$

This illustrates a somewhat unusual behaviour: both functions h_2 and k are never called, and the computation steps are independent of the arguments X and Y. One single reduction step of the function h_1 is enough for the main function to be reduced.

So what we have deduced now as information about the variable substitutions can be written in terms of the abstract domain as follows:

	β_{in}		β_{out}		Result
	X	Y	X	Y	
$f(X,Y)$	a	a	a	a	g

The abstract interpretation process must ignore the definition of h_2 and k. If it doesn't ignore it, and suppose h_2 or k would bind the arguments to some ground expressions, then we would obtain the following abstract success substitution.

	β_{in}		β_{out}		Result
	X	Y	X	Y	
$f(X,Y)$	a	a	g	g	g

This is definitely a false interpretation.

Difference 6 The degree until which an expression must be reduced is determined by the definition of the function depending on that expression.

4.2 Normalising rules

The main problem which should have been made clear throughout the previous section is the correct approximation of the ordering of the functions. The philosophy upon which the further analysis will be based is the following: represent the right hand sides of functions in such a general way that all possible sequences of function evaluations are left open. It is up to the analysis to dynamically construct correct approximations of the order of the function evaluations. Throughout the text we will base the discussions on a normalised representation of these right-hand sides.

Definition 4.1
A *simple expression* is an expression where all the arguments, if any, are variables.

Definition 4.2
A *pool of auxiliary functions* V is a set of equalities where the left-hand side is a variable and the right-hand side consists of a simple expression.

Normalising a rule is done by separating the information contained in the right-hand side into two parts: the main expression of the right-hand side, in its simple form, and a pool of functional dependencies V.

For each expression f_i that occurs as an argument of some other function, an auxiliary variable X_i is introduced. The equations $X_i = f_i$ form the entries of V. Whether the appearing arguments are ground or not, is neglected in this step and we just create the appropriate entry in the set V. The auxiliary variables are then treated as normal variables within the abstract domain, with the restriction that initially (before any of the auxiliary functions is analysed) their abstract value is set to top (meaning that they can represent any possible value).

Before giving the exact definition of the normalisation process, let's review what is meant through some examples.

Example 4.1
Some normalisation examples (where a and b are both constructors).

Original rhs	New rhs	V
g(h(X))	g(Z)	{Z=h(X)}
b	b	∅
g(h(s(X),p(Y,a)))	g(A)	{A=h(B,C),B=s(X),C=p(Y,Z), Z=a}

normalise(rhs,β) $\rightarrow$ new rhs, V
normalise($term$) = $term$,V
where $V = \emptyset$
if $term$ is ground.
normalise(X) = X,V
where $V = \emptyset$;
if X is a variable.
normalise($f(t1, t2, ...tn)$) = $f(X1, X2, ...Xn)$,V
where X_i are fresh variables
(t'_1, V_1) = normalise(t_1)
(t'_2, V_2) = normalise($t2$)
...
(t'_n, V_n) = normalise(tn)
$V = \{X_1 = t'_1, X_2 = t'_2, ...X_n = t'_n\} \cup$
$V_1 \cup V_2 \cup \ldots V_n$
if f is some constructor or defined function.

Figure 4.4: Normalisation function

Figure 4.2 formalises this process.

This formalism sticks to the concept of a result, and at the same time flattens the nested function calls. So we adopt some kind of middle coarse here. We're not rewriting the rules to some clear Prolog clauses, but we do flatten the expressions. This formalism allows us to somehow adhere to the concept of answer expression as it is known in Curry [10]: one part of a single answer expression states the resulting bindings of the initially free variables in the function call, and a second part states the result value of the function call.

The results and auxiliary variables introduced can be considered as ordinary variables in the system, and can be abstracted in the same way[8].

As a fore-taste, the analysis of the function used in the example of the previous section, could be viewed in terms of the normalisation procedure as follows

Example 4.2
We can first normalize the given function:

$$f(X,Y) \quad \Rightarrow \quad normalize(g(h_1(X,Y), h_2(X,Y))$$

[8]When implementing such a pool of simple expression, extra care has to be taken regarding the relation between the appearance of an auxiliary variable in the abstract substitution, and the appearance of this variable in the pool.

$$\Rightarrow \quad g(A, B) \wedge$$
$$V = \{A = h_1(X, Y), B = h_2(X, Y)\}$$

Supposing that g is defined by (4.3) we could view on an intuitive basis, the analysis of a call f with some abstract call substitution β_{in} as an AND-tree (figure 4.5).

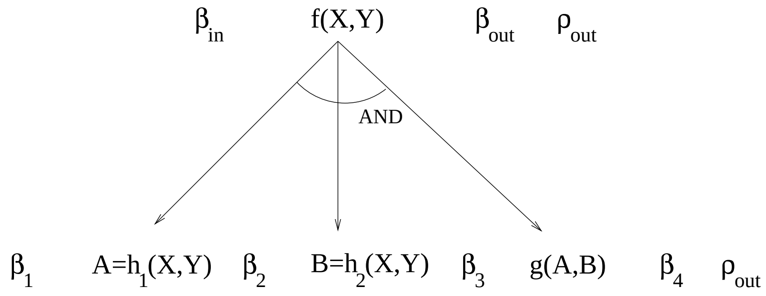

Figure 4.5: An intuitive analysis of $f(X, Y)$ where β_i represents an abstract substitution, and ρ_i represents the abstract value of a result of a function.

The figure presents a really intuitive analysis scheme. The ways the intermediary abstract substitutions and the resulting abstract substitution are computed are not yet defined, but can be intuitively understood based on the concepts presented in the previous chapter. It is worth noting that the abstract result of the initial goal is represented separately from the output abstract substitution.

4.3 Abstract domains used

As we stick to the concept of functions having results, we'll use a double abstract domain. We have the classical abstract domain A_β regarding the abstract substitutions β_i over some variables. The meaning of the abstract substitutions is given by a concretization function γ_β:

$$\gamma_\beta : A_\beta \rightarrow 2^{D_\beta}$$

where D_β is the set of possible concrete substitutions. The top and bottom elements are represented by: $\top_\beta$,$\bot_\beta$.

We'll use a second abstract domain A_ρ to represent abstract values of the possible results ρ_i of the functions. The meaning of the abstract result values is given by a concretization function γ_ρ:

$$\gamma_\rho : A_\rho \rightarrow 2^{D_\rho}$$

where D_ρ is a set of possible concrete values for a result. Top and bottom element are represented by: $\top_\rho$, $\bot_\rho$.

It is clear that these domains may not be totally independent. Results can occur as arguments for other function calls. The set of results R_1 approximated by a given abstract value ρ, and the set of concrete values R_2 associated with the translation of the abstract result into the abstract domain A_β must be such that $R_1 \subseteq R_2$.

4.4 Head normal form $\Leftrightarrow$ normal form

The problems of not-sufficiently instantiated variables and laziness will be discussed when presenting the analysis framework itself. Yet, what is worth discussing in this preliminary chapter is the fact that some expressions sometimes need not to be reduced further than some requested degree. This characteristic is one of the consequences of the lazy behaviour of the language.

As noted earlier, within the general framework for logic languages, such a problem does not arise, as each predicate is reduced until it cannot be reduced further on. How do we represent this degree of evaluation? How can we predict to what degree a function is to be analysed? How do we use such information in the abstract interpretation process?

As these problems are not easily solvable, we'll design an analysis tool which for the time being, will just ignore the factor degree. The abstract interpretation of an expression that is not directly reducible to a normal form, will return the worst case approximation for the resulting abstract substitution, as we do not know if some intermediary head normal form will be sufficient or not.

The discussion about the loss of precision that is produced by this assumption will be given in chapter 8. Techniques of adapting the analysis to such situations will also be presented there.

4.5 Higher-order functions

There is one extra dimension in Curry, not present in normal logic programming languages: higher-order functions. How such higher-order functions can be analysed will be discussed in the following chapters, as it requires deeper technical detail then used here in this introductory chapter to abstract interpretation of Curry-programs.

But as a fore-taste we can already mention that the analysis of such higher-order features will suffer from lack of precision depending on whether the function that appears as an argument can be known at analysis time or not.

4.6 Conclusion

In order to analyse a Curry program with a model similar to the one existing for logic languages we'll have to take some assumptions into account. By transforming rules into their normalised versions, we allow every possible interpretation scheme. Special constructions such as head normal forms are hard to interpret, and will be approximated badly. Partial functions will suffer from the same lack of precision. The correct interpretation of not-sufficiently instantiated variables is mostly dependent on what kind of information is collected (section 4.1.5.2).

The following chapters will present the abstract interpretation of a Curry program as a dynamic construction of AND-OR trees. Primitives such as *procedure entry*, *procedure exit*, etc... will be used in almost the same sense as foreseen in the general logic framework. In chapter 6 they will be redefined in terms of the functionalities present in Curry.

Chapter 5

Abstract Interpretation of Curry programs: intuitions

This chapter will show on an intuitive basis that in fact the abstract interpretation of Curry-programs over a given abstract domain can be done by using the same central concepts present in the general logic framework: abstract AND-OR trees.

As said earlier, all the information about the execution of a function is declared in its definitional tree. This tree may actually contain several kinds of nodes: *branch*, *or*, *and*. The leaves of the tree are the rules. Based on this information, we will show how to dynamically construct abstract AND-OR trees in each of these cases.

5.1 Interpretation of *branch*-nodes

We have: $branch(\pi, o, r, [T_1, \ldots, T_n])$

Such a node could be viewed as an artifact which states that some position should be in normal form (as said, for now we do not consider the case where head normal form would suffice), in order to proceed with applying a rule, or whatever is defined in the trees T_i. Whenever a function appears at such a position, this function must be reduced first.

Example 5.1
Indeed, suppose the following rules:

$$\begin{array}{lcll} f(a) & = & a. & \\ f(b) & = & b. & \\ k(X) & = & f(Y). & \text{with} \quad V = \{Y = g(X)\} \\ g(Y) & = & \ldots & \end{array}$$

With f described by the definitional tree:

$$branch(-, 1, rigid/flex, [f(a) \to a, f(b) \to b])$$

In order to evaluate let's say $k(b)$, $g(b)$ must first be reduced to some normal form, hopefully equal to a or b, before being able to consider the result of the main function.

Translated into the abstract interpretation framework, a *branch*-node just creates an AND-node at this level. Each of the rules appearing in a *branch*-node is part of an OR-node, as it can not be known which of the rules will be actually applicable.

After the analysis of each of the branches of the OR-node, a procedure exit operation is performed. This operation, just as in the general framework, calculates the least upper bound of the obtained results, and translates it to the variables of the level above the OR-node. In figure 5.1, β_3 is computed by the procedure exit operation on β_{r1}^{out} and β_{r2}^{out} with regard to β_2. A second procedure exit operation is necessary to translate β_3 in terms of the original variables of the calling goal in order to obtain β_{out}.

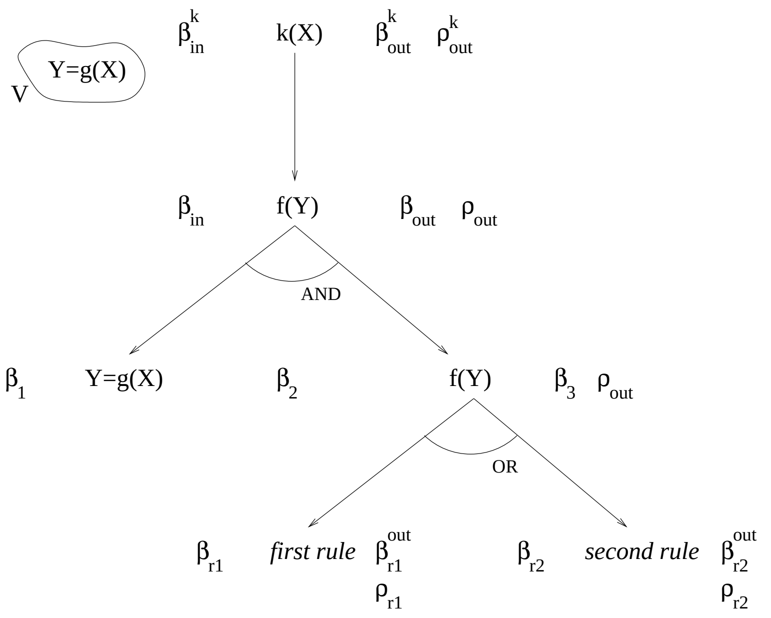

Figure 5.1: analysis of $k(X)$: AND- and OR-nodes

A *branch*-node can therefore be viewed as a composition of one AND-node and one OR-node. The first branch of the AND-node consists of the function determined by the position o in the *branch*-definition. The second branch of this AND-node consists of the continuation of the analysis of the *branch*-node, which is in fact an OR-node made out of the trees $T_1, \ldots, T_n$ defined in the *branch*-node.

More generally, we can formulate this process in terms of the auxiliary variables and the set V. Let $f(\overline{X})$ be some function which is to be analysed. Let V be the pool of simple expressions associated with f at this analysis-point. If at this particular analysis point the next analysis step is a *branch*-node defined by: $branch(\pi, o, r, [T_1, \ldots, T_n])$, and $f_o(\overline{Y})$ is the function associated with position o (this function occurs in V), then we can construct the following abstract AND-OR tree:

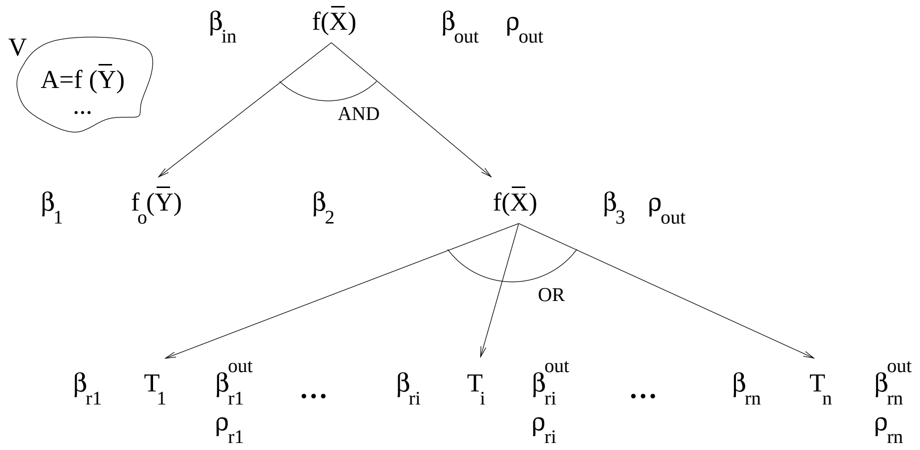

Figure 5.2: $branch(\pi, o, r, [T_1, \ldots, T_n])$

For each of the occurring branch nodes, some function in V will be analysed. Once this analysis completed the abstract value associated with the auxiliary variable in the abstract substitution is updated. When each position in a function is defined by a *branch*-node, by the time a rule is to be applied, all auxiliary variables will be instantiated with some abstract result. Eventually one could remove the corresponding entries in V. For simplicity's sake we'll assume that V remains always unchanged.

5.2 Interpretation of *and*-nodes, *or*-nodes.

We have: $and(T1, T2)/or(T1, T2)$[1]

The definitional tree of a function might contain an *and*-node, or an *or*-node. In the concrete domain, the result of a function will be computed following either the first branch of such nodes, or the second branch (or possibly both in case of the *or*-node). When analysing this function, a priori it is not known which of these branches will succeed, so both possibilities must be analysed in order to obtain correct results. Such nodes can simply

[1] *and* and *or* nodes as known in Curry are different from the AND and OR nodes in the general framework. To avoid possible confusion, the curry *and* and *or* nodes are in italic.

be abstracted by OR-nodes. After the analysis of both branches, a procedure exit operation is performed on the calculated results.

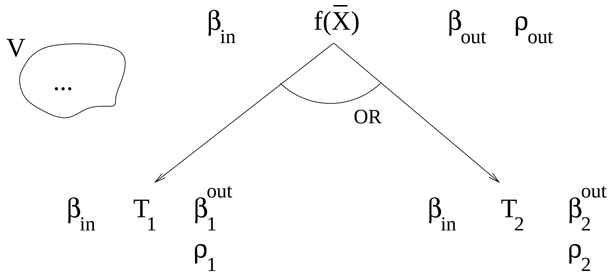

Figure 5.3: analysis scheme of a function $f(\overline{X})$, with some V, with a definitional tree given by: $and(T1, T2)/or(T1, T2)$

5.3 Interpretation of Rules

We have: $lhs \rightarrow rhs$

If each function were to be defined by some simple rules where no extra functional dependencies would be introduced ($V_i = \emptyset$), then those rules could easily be expressed as Prolog-clauses where the bodies would consist of single literals. Applying one specific rule is in such a case nothing else than a procedure entry operation as known from the logic programming framework. Just like in a procedure entry operation the abstract substitution would be expanded with local information, while eventually discarding the information about some unneeded variables.

But for general Curry programs $V_i \neq \emptyset$. As seen in the previous sections, due to the presence of *branch*-nodes, some functional dependencies will be analysed and the associated abstract variables instantiated. By the time a rule is to be applied, most of the auxiliary functions will have been analysed. Not necessarily all of them.

We can distinguish three possible cases with regard to the state of the auxiliary variables at the time a rule is to be entered:

1. all secondary functions have been analysed

2. some auxiliary variables are associated with functions that have not yet been analysed, but seem to be not-needed in the right-hand side. (a typical example for such non-strict behaviour is "if-then-else")

3. some arguments are associated with functions that have not yet been analysed, and that are needed in the right-hand side of the rule. This is typical for the lazy character of the evaluation strategy that Curry

relies on. Some function evaluations are just postponed until the moment they are needed.

Let's review these three cases.

5.3.1 No pending functions are present

This means that the rule can be applied without any preliminary interpretations. It clearly corresponds to the situation where a clause is applied, i.e. where a procedure entry has to be performed. The analysis of the left-hand side, with some abstract substitution, depends on the analysis of the right-hand side.

Entering a rule just means to calculate the new abstract substitution over the right-hand side, to generate a normalised form of this right-hand side, and to analyse the new situation using the definitional trees of the functions occurring in this right-hand side.

5.3.2 Non-strict functions

When some positions are not needed during the concrete evaluation, they may not be analysed either. This was already shown within the context of laziness in the previous chapter, and is also valid here.

Example 5.2

$$f(true, X) = false. \tag{5.1}$$
$$k(X) = true. \tag{5.2}$$
$$g(X) = f(k(X), p(X)). \tag{5.3}$$

with f eval $1 : rigid$.

Figure 5.4 shows the abstract AND-OR tree associated with this piece of Curry-program. The horizontal line symbolises a procedure entry operation. The role of this operation has become somehow extended. Not only does it discard some eventual information concerning variables that are not needed, it also discards the initial pool of auxiliary expressions V_g, and computes a new pool V_f (in this example it is the empty set).

The start position of the analysis is the goal $g(X)$ with some β^g_{in} and $V_1 = \emptyset$. The definitional tree of $g(X)$ is just a simple rule. As V is empty, we can apply the rule, i.e. we perform a procedure entry operation. Applying rule (5.3), we get $f(A, B)$ with β^f_{in} and $V_2 = \{A = k(X), B = p(X)\}$. As we know that the first position is rigid, we know that the auxiliary function must be reduced in order to further analyse the tree. This forms the first branch of our AND-node. The second branch of this node consists of the further analysis of $f(A, B)$, but now with an updated abstract substitution

β_2. We know that in the concrete domain, the evaluation of $p(X)$ would never be done, so in the abstract domain too, we must leave the analysis of $p(X)$ out. At this stage, all needed secondary functions have been analysed, therefore a procedure entry operation may be performed by just applying rule (5.1), of which the right-hand side consists of a simple constructor, which ends further expansion of the analysis tree at this level.

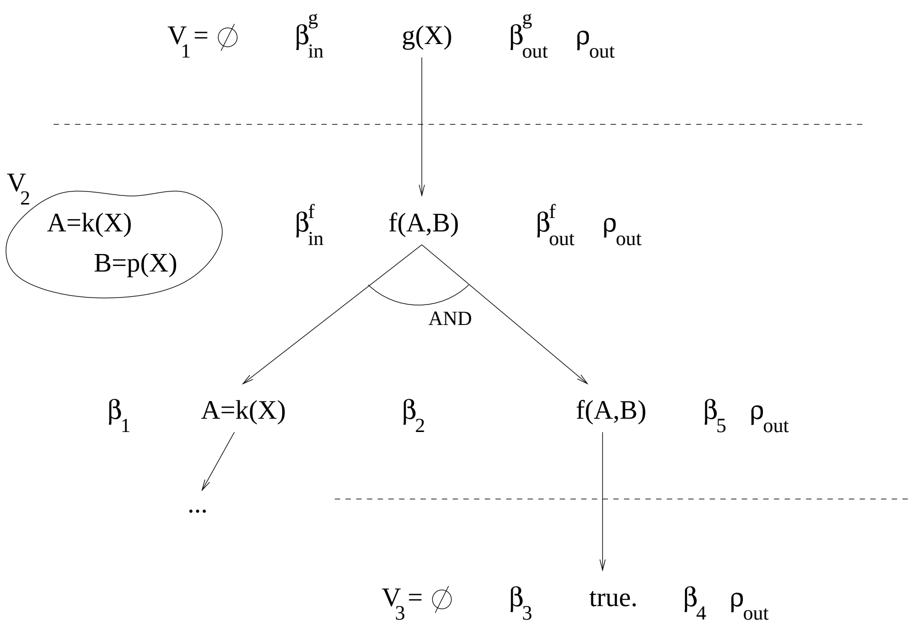

Figure 5.4: Abstract interpretation of a non-strict function.

So non-strict positions are just handled by the procedure entry operation, which just discards the unneeded information, and constructs a new pool of auxiliary functions for the further analysis.

5.3.3 Lazy positions

Unless specified otherwise, Curry always offers a lazy evaluation strategy. Trees that do not define a certain position as rigid or flex, always allow lazy evaluation. Non-strict functions, mentioned in the previous subsection, are specific cases of the general lazy philosophy.

How can we take lazy evaluations into account?

5.3.3.1 Keeping lazy lazy

With the introduction of V, the problem of analysing a simple function with $V = \emptyset$, and the problem of analysing a simple function with a non-empty V, have become equivalent. When entering a rule that has some lazy positions, we could just keep the functional dependencies left over, and add them to the new pool created during the normalisation process of the right-hand side. V just becomes bigger. By doing this we interpret lazy runtime behaviour with lazy analysis behaviour.

The reasoning seems natural. Lazy functions, whenever needed, will be evaluated at some moment. Why not think that the analysis of those functions can be postponed too?

For most functions, the analysis will succeed. For other functions though, this strategy may result in a non-terminating analysis process, as will be illustrated in the following example.

Example 5.3

$$f(1, X) = 1. \tag{5.4}$$
$$f(2, X) = g(X). \tag{5.5}$$
$$g(X) = f(1, X-1). \tag{5.6}$$

Using a lazy analysis strategy, when analysing $f(X, Y)$ we would obtain an ever-growing V. The following table shows the subsequent calls of f to be analysed, with their associated V.

		V
1	$f(X, Y)$	$\emptyset$
2	$f(X_1, Y_1)$	$\{Y_1 = Y - 1\}$
3	$f(X_2, Y_2)$	$\{Y_2 = Y_1 - 1, Y_1 = Y - 1\}$
4	$f(X_3, Y_3)$	$\{Y_3 = Y_2 - 1, Y_2 = Y_1 - 1, Y_1 = Y - 1\}$

Even if fix-point calculation is present it will not occur here. Due to the fact that computation points are not only defined by their abstract substitution, but also by V, V will be different each time, and therefore the analysis will never recognise that it is checking the same situation over and over again.

Thus, treating lazy as lazy is not a good solution. Just as we rely on the fact that the domain of the abstract substitution (i.e. the variables) is limited in size, we must guarantee that V will also remain limited.

5.3.3.2 Analyse eagerly

A better way of solving the "ever growing V problem", is to analyse laziness eagerly. Just like eager evaluation strategies, we're going to analyse all arguments before a given rule will be applied. So when arriving at a stage where a rule is to be applied, we just pick up all the functional dependencies that are left over and analyse them in advance. Due to the fact that the right-hand sides of rules can only introduce a limited amount of auxiliary function, we can then guarantee that V will always remain finite.

Example 5.4
In the previous example there was only one lazy function call. The eager approach consists of analysing what is left over in V, before applying a given rule. The analysis of $f(X_1, Y_1)$ with $V = \{Y_1 = Y - 1\}$ according to rule 5.5, will be realized by first cleaning up V, and only then applying the rule. The consecutive program points where f is involved are listed in the following table.

		V
1	$f(X,Y)$	$\emptyset$
2	$f(X_1,Y_1)$	$\{Y_1 = Y - 1\}$
3	$f(X_2,Y_2)$	$\{Y_2 = Y_1 - 1\}$

In fact, this strategy has already been insinuated when discussing laziness in the previous chapter (page 38), and the same considerations that have been exposed then, must be taken into account. As at analysis time we have no a priori knowledge in what order some functions will be performed, or whether they will be performed at all at run-time, we must consider all possible scenario's. Therefore, lazy positions introduce new abstract OR-nodes in the analysis tree. Figure 5.5 illustrates this for some function $f(X,Y)$ where both positions are lazy.

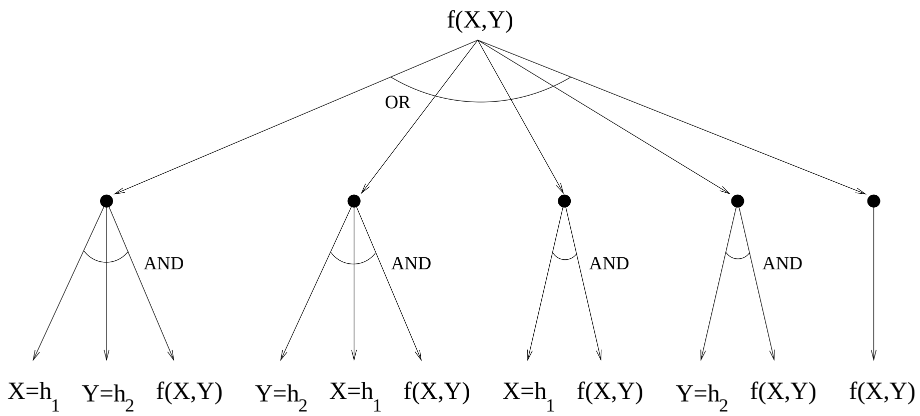

Figure 5.5: Abstract interpretation of a function with lazy positions.

This way of handling laziness should be handled with some care though. We must restrict the abstract values of the auxiliary functions at the moment such lazy positions are analysed. In those branches of the OR-node, where not all auxiliary functions are worked out, the corresponding variables must have an abstract value which reflects the idea: "we assume that the lazy function will not be evaluated, but we are not sure". This can be done by guaranteeing that the abstract values of these auxiliary variables will be set to "any" (top), at the moment the initial function will be analysed further.

With the abstract domain defined in section 4.1.1, the abstract value of the auxiliary functions will be "any" when the corresponding

So the easiest way to handle lazy evaluation is to treat it eagerly, by creating an OR-node with as branches all the possible sequences, and eventually obtain imprecise results, but at least correct ones. But again, just like all the previous cases, with the aid of the definitional tree, we can approximate the exact computation by an AND-OR tree.

Yet, there is one huge problem when analysing laziness eagerly, as the next section will show.

5.4 Abstracting higher-order functions

In the previous section, when discussing how to interpret rules, we came to the conclusion that we must treat lazy evaluation eagerly, in order to guarantee the abstract interpretation process to end.

Consider the following trivial example of the use of partial functions.

$$\begin{array}{lcl} main(X) & = & do\_something(f, X) \\ f(X) & = & \ldots \\ do\_something(A, B) & = & \ldots \end{array}$$

The analysis of the abstract function call $main(X)$, with some abstract substitution defined over X, would require the analysis of the function $do\_something(A, B)$ with some new abstract substitution and a pool $V = \{A = f\}$. Since we want to analyse this situation eagerly, we must go through f before proceeding with the definition of *do_something*. But what can we say about f in terms of the abstract substitution. What can we say about its result? We don't know anything about f, because f, as is, without arguments, can not be reduced.

Well, the previous sentence gives us already a possible solution to our problem. As we really cannot say anything about the partial function call f, the resulting abstract substitution and the abstract result just might express this lack of information. This can simply be done by setting both to their respective top-elements. This means a tremendous loss of precision.

Depending on the abstract domain used, one can improve the precision of the obtained results. In the case of a simple groundness analysis, the information previously available must not directly be thrown away. Variables that are ground before the analysis of the partial function, will remain ground after this analysis. The variables of which no precise groundness information was available (i.e. their abstract value corresponding to "any"), will keep the same abstract values. In such a case, the only true loss of information is the abstract value of the result of such a call, which will have to be set to "any".

When the purpose of the analysis is to check determinism, we will see that the only acceptable solution of the analysis of partial functions will be the worst case.

Let's illustrate all this with two other trivial examples.

Example 5.5

$$\begin{array}{lcl} main(X,Y) & = & apply(f(X),Y). \\ f(X) & = & g(X). \\ g(X,Y) & = & \ldots \\ apply(A,B) & = & A@B. \end{array}$$

In this example we can make one reduction step when analysing $f(X)$, but when arriving at $g(X)$, we're again in a situation where we cannot say anything about g.

Example 5.6
Rewriting somehow the first example to something like the following lines would on the contrary not present any problems.

$$\begin{array}{lcl} main(X) & = & f@X. \\ f(X) & = & \ldots \end{array}$$

In this case all the needed information to handle the partial function is present. As @ is a built-in, the correct interpretation of it is left to the definition of this built-in within the abstract domains used.

As the correct way of handling partial functions depends on the abstract domain used in the analysis, we'll treat this interpretation as a new primitive belonging to the same category as the analysis of built-in functions.

5.5 Conclusion

What this chapter has demonstrated, is that all what is to know about the computation of some goal is contained in the definitional trees of the functions in the program. The same definitional trees can be used to build up abstract AND-OR trees.

These generated trees do not have exactly the same shape as we know it from the logic programming framework. In the general framework we have: for each predicate to be analysed, the first level consists of one OR-node that distinguishes the different clauses that can be applied, and in a second level we have AND-nodes to represent the sequence of literals present in the bodies of the clauses. Here the constructed AND-OR tree within one level is some mixture of AND's and OR's.

The interpretations of each kind of node, or each kind of rule are summarised in tables 5.1 and 5.2.

Just as this chapter was concerned with a rather intuitive approach, the next chapter will state the abstract interpretation procedure in a more formal way.

node	interpretation
$branch(\pi, o, r, [T_1, T_2, .., T_n])$	(analysis of argument at position o) AND (analysis(T_1) OR analysis(T_2) OR ... ... OR analysis(T_n))
$and(T_1, T_2)$	(analysis(T_1)) OR (analysis(T_2))
$or(T_1, T_2)$	(analysis(T_1)) OR (analysis(T_2))

Table 5.1: node $\rightarrow$ interpretation

rule (lhs $\rightarrow$ rhs)	interpretation
no lazy positions	procedure entry $\rightarrow$ analysis(rhs)
lazy positions $(i \ldots j)$	OR $(\prod(i \ldots j))$ where $\prod(i \ldots j)$ stands for the analysis of the arguments for each of the permutations of the elements $i \ldots j$

Table 5.2: rule $\rightarrow$ interpretation

Chapter 6

Abstract Interpretation of Curry: formalisation

The central concept within the presented abstract interpretation framework remains the abstract AND-OR graph. Unlike for logic languages, the AND-OR graph is constructed dynamically. Depending on the way some functions are called, AND and OR nodes must be added.

Also, if the analysis contains some freeness information, some abstract function calls must perhaps not be analysed when we know that the corresponding concrete function calls will get suspended. In the subsequent discussion though, we will assume that nevertheless, analysis is continued.

We've also seen that analysing lazy behaviour eagerly introduces new OR-nodes, and eventually may reduce drastically the precision of the results obtained.

The assumption made in the previous chapters is still valid here: constructor based expressions which depend on functions that remain to be evaluated, are not handled (i.e. as resulting abstract interpretation the top element of the abstract domain is returned).

Let us first present a small overview of what was discussed in the previous chapter.

6.1 General Comparison

One level of analysis of a user defined predicate within the framework in [3] can be represented as in figure 6.1 (horizontal lines are used to delimit what is meant here by "level"). The procedure entry primitive creates one OR-node, and several AND-nodes corresponding respectively with the different clauses defining the predicate, and the literals of the bodies of each of the clauses. For each of the AND-branches it calculates the respective starting abstract substitutions. The procedure exit operation permits one to exit such a level of analysis by resuming the information collected and translating it to the initial variables with which the predicate was called. We can therefore view the procedure entry as the operation which allows one to enter such a level of analysis, and the procedure exit to exit the level. These are the only operations which allow communication between two levels.

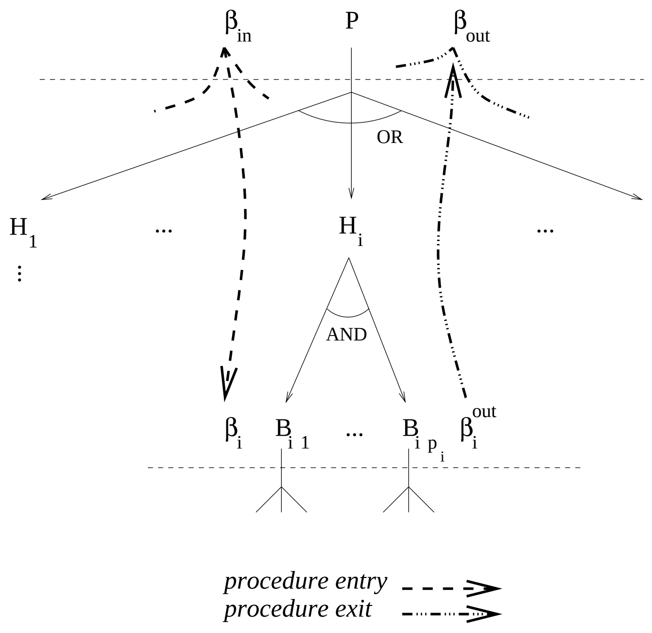

Figure 6.1: Logic framework

The framework presented for the functional logic language Curry can be viewed in an analogous way. But instead of leaving the task of the construction of the OR and AND nodes to the procedure entry operation, we view this operation as the procedure which just calculates the general starting abstract substitution and pool of auxiliary functions within such a level. The general abstract interpretation algorithm dynamically generates the OR and AND trees. Unlike in the framework for logic languages, in one such level there can be several OR nodes, just like there can be several AND nodes. The procedure exit operation consists of summarising the information collected, but now the information is not translated to the initial variables of the call, but to the variables corresponding with the OR-node with which it has been created. Therefore, just like in the analysis scheme for logic languages, procedure entry allows one to enter a new analysis level and procedure exit can be used to exit one such level.

Graphically we can view this as in figure 6.2.

With this general image in mind we can now present the actual abstract interpretation strategy, starting with the redefinition of the primitive operations.

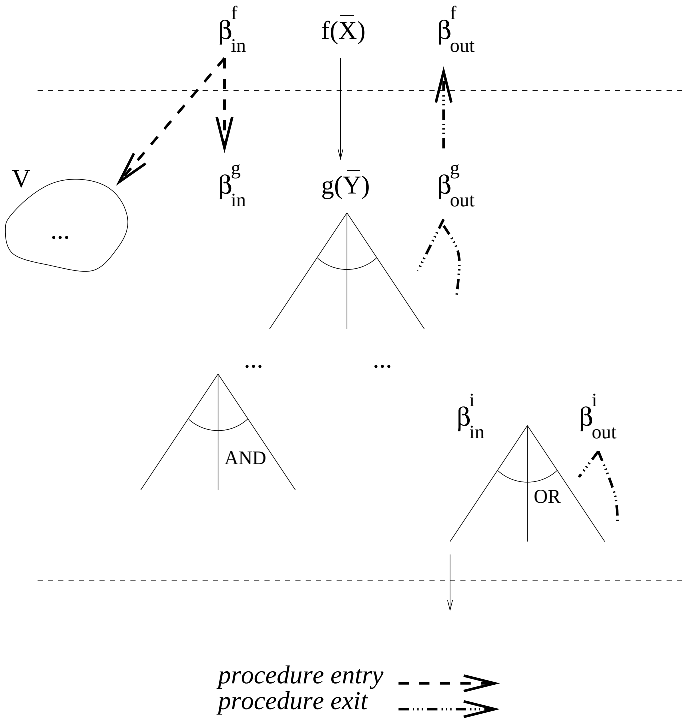

Figure 6.2: General Curry analysis tree

6.2 Primitive Operations

6.2.1 Procedure entry

A procedure entry is only then performed when a rule can be applied. All needed auxiliary arguments are already worked out. The only task this operation has is to correctly initialise a new analysis level. This means the renaming of the needed variables (eventually). This also means the construction of a correct set of auxiliary expressions, which allows the correct interpretation of each of the possible evaluation scenario's. And last but not least, the procedure entry operation has to deduce the correct starting abstract substitution from the abstract substitution of the upper analysis level.

To put it more systematically we have the following input:

- the initial goal G.
- the calling abstract substitution β_{in}.
- the rule that is to be applied ($lhs \rightarrow rhs$).

The consecutive tasks of the procedure entry operation are:

renaming: This is a classical step of the procedure entry operation. In the implementation used (Prolog), no renaming is needed, as the arguments/variables are represented by unique free logic variables.

unification: Implicit unification of the given goal with the left-hand side of the rule[1].

normalisation: The rule is given in the form as it can be found in the definitional tree of a function. This form is to be transformed to it's normalised form. In this step a new set V is calculated, and the introduced auxiliary variables are inserted into β_{in}, as well as the local variables occurring in the right-hand side. We therefore obtain an updated abstract substitution β_{Temp}, a pool of auxiliary expressions V, and a new goal NG to be analysed further on.

restriction: The information of the initial abstract domain has to be restricted in such a way that the information not needed in the right-hand side is discarded from it. We obtain a final abstract substitution β_{new}

A sufficient condition for correctness of the procedure entry operation is, informally stated, that when applying the specific rule to a concrete substitution of the the initial abstract goal, the corresponding abstract right-hand side at least covers the corresponding concrete right-hand side.

6.2.2 Procedure exit

Whenever at some analysis stage an OR-node was created within some level, after the analysis of each of the branches of such node, the information obtained in each of these branches must be summarised and rewritten in terms of the abstract substitution with which the OR-node was initially created. Summarising the information is nothing else then performing a least upper bound operation on all of the success substitutions calculated.

The operation needs as input:

- the initial abstract substitution β_{in} as it was at the moment the OR-node was created.
- the result abstract substitution calculated for each of the OR-branches: $\beta_1^{out}, \beta_2^{out}, \ldots, \beta_n^{out}$ (where n is the number of branches of the OR-node).

[1]The unification procedure corresponds with an abstract interpretation of a built-in. Usually, in static analyses of logic programs, the authors work with flattened forms of the clauses (i.e. each predicate is called with variables which are mutually different, literals of the form $X_i = p_j(\ldots)$ are introduced whenever needed). As stated in [13], the choice between implicit unification and explicit unification is just a matter of design.

- the abstract values of the possible results: $\rho_1, \rho_2, \ldots, \rho_n$

The procedure exit operation then performs the following tasks:

- eventual *renaming* of the variables occurring in each β_i^{out}
- $\beta = \text{lub}(\beta_1^{out}, \beta_2^{out}, \ldots, \beta_n^{out})$
- $\beta_{out} = extension(\beta, \beta_{in})$: this operation extends the information collected in β to a new abstract interpretation β_{out}, based on the information that was initially present (β_{in}).
- $\rho = lub(\rho_1, \rho_2, ..., \rho_n)$

This operation has the responsibility to compute an approximation of the possible concrete success substitutions and the possible concrete result values, that are on the one hand correct, and on the other hand as precise as possible.

For all possible concrete substitutions represented by β_{in}, the procedure exit must guarantee that the calculated abstract success substitution β_{out} covers all the concrete success substitutions which can be obtained when calculating based on the concrete substitutions. An analogue condition for the calculated ρ is required.

6.2.3 Abstract interpretation of built-ins

Built-in calls must be analysed in a separate way as they do not follow the normal computation strategy. In Prolog, typical built-ins are unification, but also calls such as: read, write, etc...

In Curry most of the functions are described by a definitional tree. Even built-in functions such as input/output, arithmetic calculations, etc... have their definitional tree to define the calculation process[2]. Only four types of true built-ins are present:

1. equational constraints (=).

[2]These definitional trees actually contain the information that in order to proceed with the reduction of the concerned function, all arguments must be ground (*branch*-nodes). Such functions can in fact be viewed as being defined by an infinite set of ground rules. The operator "+" could for example be defined by:

$$\begin{array}{rcl} 0+0 &=& 0. \\ 0+1 &=& 1. \\ &\vdots& \\ 1+0 &=& 1. \\ 1+1 &=& 2. \\ &\vdots& \end{array}$$

2. strict equality (==).
3. function application (@).
4. partial functions[3].

The abstract interpretation of a built-in function F with abstract call substitution β_{in}, computes the abstract success substitution β_{out} and the abstract result ρ. The exact interpretation is of course dependent of the built-in and the abstract domain used.

Informally, the computed β_{out} must represent all the possible concrete success substitutions and results, which can be obtained when applying the built-in on any concrete substitution represented by β_{in}.

6.3 Basic abstract interpretation algorithm

Now that the basic operations have been reviewed, we can present the main lines of the static analysis process of some given goal $Goal$, adorned by some initial abstract call substitution β_{in}, and complemented with some information about eventual auxiliary functions (V).

6.3.1 "Analyse"

The main procedure of the algorithm is **analyse**. This procedure calculates the abstract answer expression of a goal, i.e. abstract substitution over the given variables (β_{out}), and abstract value of the result (ρ) one may expect at runtime. It needs the following information: the goal to be analysed ($Goal$), the abstract values of the arguments that appear in the goal (β_{in}) and eventually some functional dependencies (V).

analyse$(Goal, V, \beta_{in}) \rightarrow \rho, \beta_{out}$
case 1: $Goal$ = constructor based expression
then $((\rho, \beta_{out}) = (\top_\rho, \top_\beta)$
case 2: $Goal$ = unreducible expression
(i.e. the expression is in some head normal form)
then $(\rho,\ \beta_{out})$ is to be deduced from β and V.
case 3: $Goal$ = built-in function
then (ρ, β_{out}) = analysebuiltin$(Goal, V, \beta_{in})$
case 4: $Goal$ = defined function
let $Tree$ = definitional tree of the function
then (ρ, β_{out}) = analyseWithTree$(Goal, V, \beta_{in}, Tree)$

[3] As mentioned in section 5.4, we'll consider the partial functions as a built-in functionality, of which the correct interpretation is dependent on the abstract domain used.

The procedure **analyse** distinguishes four cases of expressions which are to be interpreted.

The first possibility is that the expression is some constructor-based expression. According to the assumption made in the previous chapters, we define the result of the analysis of such an expression to be the worst possible, i.e. $\top_\rho$, and $\top_\beta$.

The second possibility is that the function is in some other unreducible form, so that we can calculate the abstraction of the result with the information in β_{in}, and β_{out} is just equal to β_{in}.

When the goal to be analysed is a built-in function, we analyse the expression according to the rules defined on the built-in function. The procedure **analysebuiltin** is here an abstraction of all the possible abstract interpretations of built-ins. How built-ins are to be evaluated is dependent on the abstract domain used. These different analysis procedures are therefore not detailed here.

In the most general case, the expression will be a defined function though. The analysis of the given goal will then be processed according to the definitional tree of the function.

6.3.2 "AnalyseWithTree"

In curry, we have four possible nodes in the definitional tree of some function: a rule, a *branch*-node, an *or*-node or an *and*-node.

6.3.2.1 Analysis of a rule

analyseWithTree$(Goal, V, \beta_{in}, rule(lhs \to rhs)) \to \rho, \beta_{out}$

$ListPos = \text{getRemFuncPos}(Goal, \beta_{in}, V, lhs, rhs)$

$ListSeq = \text{computeAllSequences}(ListPos)$

$(List\_\rho, List\_\beta_{out}) =$

$\quad \text{analyseForAllSeq}(Goal, \beta_{in}, V, ListSeq, ListPos, lhs, rhs)$

$(\rho, \beta_{out}) = \text{procedure\_exit}(\beta_{in}, List\_\rho, List\_\beta_{out})$

In order to guarantee that all secondary functions are deleted before an actual rule is applied, and that the eager evaluations of those functions are correct, we must create at this stage an OR-node, where the branches are all the possible sequences of the function analysis left over. **GetRemFuncPos** computes the positions of the arguments in the given goal, that are associated with yet unevaluated functions, and which are actually needed in the right-hand side.

getRemFuncPos$(Goal, \beta_{in}, V, lhs, rhs) \to ListPos$

$ListUnevalPos = \text{getUnevalPositions}(Goal, \beta_{in}, V)$

$ListNeededPos = \text{getNeededPositions}(lhs, rhs)$

$ListPos = \text{intersect}(ListUnevalPos, ListNeededPos)$

ComputeAllSequences is a simple procedure which just produces all possible sequences of the elements of all possible subsets of the given list of positions. **AnalyseForAllSeq** then collects all the information about the analysis according to each of those sequences, and results in two lists: a list of possible results, and a list of all the obtained $\beta_{out,i}$. **Procedure_ exit** summarises the information and translates it in terms of the initial abstract call substitution.

6.3.2.2 Analysis of *or*-, and *and*-nodes

analyseWithTree$(Goal, V, \beta_{in}, or(T1, T2)) \rightarrow \rho, \beta_{out}$
  $(List\_\rho, List\_\beta_{out}) =$ analyseForAllTrees$(Goal, V, \beta_{in}, [T1, T2])$
  $(\rho, \beta_{out}) =$ procedure_exit$(\beta_{in}, List_\rho, List_\beta_{out})$

AnalyseForAllTrees is just a procedure which simulates the behaviour of an OR-node. All subsequent paths are analysed independently, and the results are listed in $List\_\rho$ and $List\_\beta_{out}$.

The *and*-node is treated in the same way.

analyseWithTree$(Goal, V, \beta_{in}, and(T1, T2)) \rightarrow \rho, \beta_{out}$
  $(List\_\rho, List\_\beta_{out}) =$ analyseForAllTrees$(Goal, V, \beta_{in}, [T1, T2])$
  $(\rho, \beta_{out}) =$ procedure_exit$(\beta_{in}, List_\rho, List_\beta_{out})$

6.3.2.3 Analysis of *branch*-nodes

The analysis of a *branch*-node is to be done in a different way.

analyseWithTree$(Goal, V, \beta_{in}, branch(\_, Pos, Mode, ListT)) \rightarrow \rho, \beta_{out}$
  $\beta_{new} =$ pre_entryBranch$(Goal, V, \beta_{in}, Pos, Mode, ListT)$
  $(List\_\rho, List\_\beta_{out}) =$ analyseForAllTrees$(Goal, V_{new}, \beta_{new}, ListT)$
  $(\rho, \beta_{out}) =$ procedure_exit$(\beta_{in}, List_\rho, List_\beta_{out})$

The **pre_entryBranch** procedure analyses the secondary function that occurs at position Pos, if there is a function associated with this position. If not, then, with the aid of the abstract substitution the conditions to proceed are verified[4]. After this step, the analysis can proceed by analysing all the different paths described in $ListT$.

pre_entryBranch$(Goal, V, \beta_{in}, Pos, Mode, ListT) \rightarrow \beta_{out}$
  $Arg =$ getArgAtPos$(Goal, Pos)$
  $Info =$ getInfoArg(Arg, V, β_{in})
  if $Info$ is a function to be analysed

[4] It is this function that should possibly be modified if some different behaviour is requested as we will see in the case of determinism analysis.

then
  $(\rho_{Temp}, \beta_{Temp}) = \text{analyse}(\mathit{Info}, V, \beta_{in})$
  $\beta_{out} = \text{update}(Arg, \beta_{in}, \rho_{Temp}, \beta_{Temp})$
else
  $\beta_{out} = \beta_{in}$
  (there is nothing to do)

As said, the procedure **pre_entryBranch** analyses the function occurring at position Pos, if there is one, and updates the initial abstract substitution with the results obtained. The small help procedures **getArgAtPos**, **getInfoArg** have respectively the meaning: determine which argument appears at the specified position, and query the pool V in order to determine if the argument is associated with some not previously analysed secondary function. **Update** has the task to update the information about a particular argument in the initial abstract substitution, by the aid of the calculated new abstract value related to that argument, and the new abstract substitution that was obtained when analysing the particular function initially associated with that argument.

6.3.3 "AnalyseForAllTrees" and "AnalyseForAllSeq"

AnalyseForAllTrees and **analyseForAllSeq** are defined as follows:

$$\textbf{analyseForAllSeq}(\_,\_,\_,[\,],\_\_,\_) \rightarrow [\bot_\rho],[\bot_\beta]$$
$$\textbf{analyseForAllSeq}(Goal, \beta_{in}, V, [Seq_1|RestSeq], ListPos, lhs, rhs) \rightarrow [\rho_i|List\_\rho], [\beta_{out,i}|List\_\beta_{out}]$$
$$(\rho_i, \beta_{out,i}) = \text{analyseForSeq}(Goal, \beta_{in}, V, Seq_1, ListPos, lhs, rhs)$$
$$(List\_\rho, List\_\beta_{out}) = \text{analyseForAllSeq}(Goal, \beta_{in}, V, RestSeq, ListPos, lhs, rhs)$$

in an analogous way:

$$\textbf{analyseForAllTrees}(\_,\_,\_,[\,]) \rightarrow [\bot_\rho],[\bot_\beta]$$
$$\textbf{analyseForAllTrees}(Goal, V, \beta_{in}, [T_i|RestT]) \rightarrow [\rho_i|List\_\rho], [\beta_{out,i}|List\_\beta_{out}]$$
$$(\rho_i, \beta_{out,i}) = \text{analyseForTree}(Goal, V, \beta_{in}, T_i)$$
$$(List\_\rho, List\_\beta_{out}) = \text{analyseForAllTrees}(Goal, V, \beta_{in}, RestT)$$

What remains worthy to be detailed here is the procedure **analyseForSeq**, as it defines a clear AND-node, and is the only procedure that calls the procedure entry operation. A sequence is nothing else but a list of integers that says more about the order in which the arguments are to be analysed.

analyseForSeq$(Goal, \beta_{in}, \_, [\,], ListPos, lhs, rhs) \rightarrow \rho, \beta_{out}$

β^1_{new} = ensureAllUnevalSetToAny$(\beta_{in}, ListPos)$

$(NewGoal, \beta^2_{new}, V_{new})$ = procedure_entry$(Goal, \beta^1_{new}, lhs, rhs)$

(ρ, β_{out}) = analyse$(NewGoal, \beta^2_{new}, V_{new})$

analyseForSeq$(Goal, \beta_{in}, V, [Pos|RestPos], ListPos, lhs, rhs) \rightarrow \rho, \beta_{out}$

Arg = getArgAtPos$(Goal, Pos)$

$Info$ = getInfoArg(Arg, V, β_{in})

(ρ_{temp}, β_i) = analyse$(Info, \beta_{in}, V)$

$\beta_{in,New}$ = update$(Arg, \beta_{in}, \rho_{temp}, \beta_i)$

$NewListPos$ = remove$(Pos, ListPos)$

(ρ, β_{out}) =

analyseForSeq$(Goal, \beta_{in,New}, V, RestPos, NewListPos, lhs, rhs)$

In the first case, all positions that had to be checked are already analysed (empty list). But a sequence does not necessarily mention all positions that contain unevaluated functions, as an analysis must consider that some positions won't ever be reduced. $ListPos$ therefore eventually contains these remaining positions. The arguments occurring at these positions, must be decoupled from their possible entries in the associated pool of functions V. This is the purpose of **ensureAllUnevalSetToAny**. In the case of groundness analysis, their abstract value can just be set to any (which explains the name chosen for this procedure). When other abstract domains are used, possibly another kind of operation on the variables in β_{in} will be required.

In the second case, some functions remain to be analysed. After the analysis of the concerned position one needs to update the initial abstract substitution, as the variable Arg is now instantiated with the obtained abstract result. Other variables might have new abstract values. Recursion is ensured by the call to **analyseForAllSeq**, this time though with a new list of positions, as we do not want that the result obtained for the argument gets discarded when in some subsequent step **ensureAllUnevalSetToAny** is performed.

6.3.4 A small overview

To summarise this model we can reduce the algorithm into two points with regard to the goal G that has to be analysed, and β_{in} and V which adorn this goal:

1. The goal is a built-in: perform abstract interpretation
2. The goal is a user-defined function:
   (a) construct an abstract AND-OR tree based on the definitional trees of the function G and of the functions present in the set V.
   (b) perform procedure entry whenever necessary

(c) perform procedure exit each time all the branches of some OR-node have been analysed.

This scheme is obviously totally useless when recursive functions are to be analysed. Within the algorithm presented here, an infinite abstract AND-OR graph would be constructed. The next section handles this case.

6.4 Fix-point computation

In our analysis model, functions are not only specified by their abstract substitution sets, but also by the pool of auxiliary functions that may exist. The notion of level is here a bit different than the one in the basic logic framework. Before a level limit is crossed, all functional dependencies are worked out. When crossing the limit (procedure entry operation), a new goal is constructed, with a new abstract substitution, but also with a new pool of supplementary functions. This state is also the starting point of the main analysis procedure. So in terms of our algorithm, a function appears to be recursive when the abstract substitution and V correspond with β and V of a previous call to the same main function.

At each new function that ought to be analysed we can therefore distinguish three cases:

1. the same function does not appear anywhere in the abstract AND-OR tree constructed so far.

2. the function already appears somewhere earlier in the tree, but with different call conditions (different β and V)

3. the function occurs earlier in the tree, with the same call specifications.

In the first two cases the general algorithm is suitable. In the last case, we would obtain an infinite abstract AND-OR tree.

In order to analyse recursive functions, a fix-point computation must be started.

At the moment some function appears to be recursive, the result substitution is first approximated (deduced eventually from the information already available, in our implementation we just chose as approximation the bottom element of the abstract domain). The calculation continues based on the approximation. When the analysis returns at the initial call of the recursive function, the resulted abstract success substitution is saved, and a new analysis is started. The recursive function is again encountered, but now the approximation chosen is the abstract call substitution previously calculated. Again, when arriving at the initial call, we calculate the new result abstract. This process is repeated until the new approximation calculated equals the previous one. This process will calculate a correct abstract success substitution in a finite number of analysis steps. The termination of the process is indeed guaranteed by the fact that the procedure exit operation relies on the least upper bound calculation defined on the abstract domain. As we required the concretization function to be monotonic, the computation will terminate (in the worst case with as result abstract success substitution the top element of the abstract domain).

This is the general philosophy as it is explained in [3]. The way the fix-point calculation is implemented here is inspired by the works of [19], where the concept of a memo- table is used. Each entry in the memo-table contains the following fields:

1. the initial goal.
2. the abstract call substitution β_{in}.
3. V.
4. ρ.
5. the abstract success substitution β_{out}.
6. a status field. This field can have the following values: *analysing*, *complete*, *approximate* or *fix-point*.

When a new function is to be analysed, a memo entry is created. The content of the first three fields are known. The next two fields can be filled with anything, as long as they represent some approximation of the possible result (in our case: *bottom*). The last field is then set to *analysing*. At the moment when the analysis of a particular function terminates, the fourth and fifth field can be updated, and the last field is set to *complete*.

When a recursive call occurs (i.e. a memo-entry exists with corresponding β and V), the last field of the entry is set to *fix-point*. All the functions which depend on the result of the recursive call are then only approximations. So instead of exiting their analysis by setting their status field to complete, we set it to *approximate*. This way, a clear distinction is made between the clean non-recursive functions, where once their result success abstraction calculated, the computations need not to be redone, and functions which are recursive in an indirect way, and which need to be recalculated each time a better approximation of the recursive function they depend on is found.

6.5 Abstract interpretation algorithm revisited

With regard to the algorithm presented earlier, and the fix-point computations necessary to cope with recursive functions, here a small overview of the general analysis procedure is presented:

Static analysis of a goal F with abstract substitution β_{in} and secondary functions defined in V:

1. if F is a built-in, analyse the functions in V and the function itself accordingly.
2. if F has no ancestor node with a call for the same function.
   (a) analyse the functions in V according to the definitional tree of the function.
   (b) perform a procedure entry operation at the moment a rule can be applied.
   (c) apply abstract interpretation of the goals appearing in the right-hand side.
   (d) apply procedure exit.
3. F has an ancestor with a call F' to the same function. The abstract informations related to the functions are the same, up to renaming. The abstract success substitution of F is approximated by the value bottom. A fix-point computation is started.
4. F has an ancestor with a call F' to the same function, but the abstract substitutions are not compatible. Proceed as in case 2.

Chapter 7

Determinism Analysis

The abstract interpretation framework developed in the earlier chapters can now finally be applied. The aimed elegance of this framework was that in order to apply it, the only operations that must be specified are the primitive operations: procedure entry, procedure exit, and abstract interpretation of built-ins.

I/O within Curry is required to be deterministic. In chapter 2 it was shown that deterministic I/O can be ensured by producing a run-time error when eventual non-deterministic steps within an interaction-function are performed. This is a rather drastic way of dealing with the problem. A more elegant solution would be to let the compiler produce eventual warnings if such situation should occur. The realisation of this solution is to build an analysis-tool that checks for non-deterministic calculations.

The first section of this chapter will reformulate the exact conditions in which non-deterministic calculations might occur. In the subsequent section we'll define a suitable abstract domain. The next sections are then concerned with the definition of the primitive operations relative to this abstract domain. But the decision on whether some computation is non-deterministic or not, will have to be taken at a yet lower level then the primitive operations. Some small changes to the basic analysis algorithm will have to be taken into account. At the end a small determinism analysis example will be presented.

7.1 Conditions for non-determinism

The execution of I/O programs may not produce any disjunctive answer expression. The language is already foreseen with a run-time detection system. This detection is very simple. If at some program point a disjunction of answers is produced, then the system is halted, no matter what. Producing run-time errors is a not so elegant solution. It would be much more interesting for the programmer to receive warnings of possible non-determinism at compile-time. And this is where an abstract interpretation of the program is appropriate.

As already shown in chapter 2, there are only two situations that can cause a disjunction in the answer expression.

Situation 1
Suppose:

$$\begin{array}{lcl} nondet1 & :: & io(). \\ nondet1 & \mathit{if} & generate(X) = putChar(X). \\ generate & :: & char \rightarrow bool. \\ generate(a) & = & true. \\ generate(b) & = & true. \end{array}$$

The function *generate* is translated in Curry into a *branch*-node:

$$\begin{array}{l} branch(generate(X), 1, flex, [\ generate(a) \rightarrow true, \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad generate(b) \rightarrow true\ ]). \end{array}$$

When calling *nondet*1, *generate* is called with a free logic variable. In this situation we have two possible bindings for this free variable: $X = a$ or $X = b$. A disjunction in the answer expression will be the result, and a run-time error will be produced, signalling a non-deterministic calculation. If *generate* would have been called with an instantiated variable, or if its evaluation scheme would have been rigid, no such disjunction would occur. The calculus would be deterministic too when only one rule defining *generate* would have existed.

Situation 2
Suppose:

$$\begin{array}{lcl} nondet2 & :: & char \rightarrow io(). \\ nondet2(a) & = & putLine('Hello'). \\ nondet2(X) & = & putLine('Anything'). \end{array}$$

This function is translated in Curry into an *or*-node.

$$\begin{array}{ll} or & (branch(\_, rigid, [nondet(a) \rightarrow putLine('Hello')]), \\ & (nondet2(X) \rightarrow putLine('Anything')). \end{array}$$

The function call *nondet*2(a) would want to produce two possible outputs: 'Hello' and 'Anything'. A call to *nondet*2 with its argument free, would want to produce the same disjunctive answer.

More formally we have: Let $f(X_1, \ldots, X_n)$, let $\mathcal{T}$ be the definitional tree associated with f. If at run-time, the reduction process of a call to f produces a disjunctive answer expression then:

1. the evaluation of f possibly arrived at a point where further reduction was based upon some *branch*-node

$$\mathcal{T}_b = branch(\_, Pos, flex, [\mathcal{T}_1, \ldots, \mathcal{T}_p])$$

   contained in $\mathcal{T}$, where $p > 1$ and where X_{Pos} is a free variable.

2. or $\mathcal{T}$ contained some *or*-node.

3. or one of the subgoals defining f was non-deterministic.

7.2 Abstract domains

7.2.1 Information to collect

What kind of information do we have to collect?

To detect situations of non-determinism caused by *or*-nodes, their are two solutions. In order to detect non-determinism, or rather determinism, we would need to check with which possible concrete instantiations the considered function can be called. In order to do so, we would need an abstract domain containing all concrete values. But then, we might as well just perform the program at real-time. So the best an abstract analysis can do is just check if a function with a definitional tree containing an *or*-node is encountered or not. This does not need any particular abstract information to be present.

Detecting situations of non-determinism caused by *branch*-nodes though, requires some more expensive analysis, as some notion of groundness information is needed. Indeed, in order to detect non-determinism caused by situations as in situation 1, it is sufficient to know if a variable is ground, in order to decide whether a calculation will be deterministic or not.

So in fact, the determinism analysis can be viewed as a combination of two analysis: groundness analysis, and on top of it, determinism analysis.

Determinism Information

$\Uparrow$

Groundness Information

$\Uparrow$

Program

Although theoretically one could truly separate these two analysis, in the presentation here, as well as in the actual implementation, these two are directly combined into one analysis.

The main procedure **analyse** (page 62) presented in the global abstract interpretation algorithm returns two kinds of information:

1. Abstract value of the result: ρ_{out}.

2. General abstract information about the function call and more particularly the variables in the call: β_{out}.

In our case, β_{out} would contain two fields: information regarding the exit abstract substitution over the variables, and information concerning the deterministic character of the function analysed. In the subsequent sections we'll directly split β_{out} into these two parts, and consider that the main procedure **analyse**, as well as the underlying procedures, return a triple instead of a couple:

1. Abstract value of the result: ρ_{out}.
2. Abstract exit substitution over the variables present in the goal: β_{out}[1].
3. Determinism annotation: δ_{out}.

Anyway, we'll have to deal with three abstract domains, representing respectively the abstract results, the abstract substitutions and the determinism information.

7.2.2 Groundness information: abstract substitutions, abstract results

7.2.2.1 Abstract substitutions

The abstract domain used to represent substitutions corresponds to the one which was presented already earlier (chapter 4, page 33), and needs not to be repeated here. It's basic elements are assignments of the form $X_i := m_i$, where $m_i \in \{g, a\}$, representing respectively "ground" and "any".

7.2.2.2 Abstract results

The domain for results is as simple as the one for the substitutions:

Definition: The abstract value ρ for a result can have the values g or a ($= \top_\rho$), abbreviations of "ground" and "any".

Concretization function γ_ρ:

- $\gamma_\rho(\perp_\rho) = \emptyset$ (no result).
- let R be a set of results, $R \subseteq \gamma_\rho(\rho) \Leftrightarrow$
  - if $\rho = g$ then R contains only ground expressions.
  - if $\rho = a$ then R may contain any possible expression.

[1]From the context it should be clear when β_{out} is meant to be the general abstract information, or more particularly the abstract substitution.

Ordering, least upper bound: The ordering of the abstract elements is the same as the ordering of the basic elements within the abstract substitutions. We have: $\perp_\rho \leq g$, $\perp_\rho \leq a$, and $g \leq a$.

The compatibility between the abstract substitutions and the abstract results is trivial.

7.2.3 High-level abstract domain: determinism

On top of the preceding abstract domains we define the abstract domain which aims to collect the interesting information concerning determinism. Unlike the previous domains expressing possible concrete values of variables, this domain will characterise possible concrete kinds of computations.

Definition: The deterministic character δ of an abstracted function call F is specified by either: $\perp_D$, *det*, *nondet* or $\top_D$.

Concretization function γ_D: Let C be a set of concrete computations, let δ be some abstract determinism information concerning the abstract call F. $C \subseteq \gamma_D(\delta) \Leftrightarrow$

- if $\delta = \perp_D$ then C is the empty set.
- if $\delta = det$ then C contains only computations which are deterministic.
- if $\delta = nondet$ then C contains computations which are deterministic or non-deterministic.
- if $\delta = \top_D$ then C contains any kind of concrete computations.

Note that in fact *nondet* and $\top_D$ can regroup the same set of function calls. The reason why we do distinguish these two is the following. If an abstract function call is annotated with *nondet*, then we know for sure that one of the necessary conditions for non-determinism was fulfilled. On the other hand, we will encounter function calls that we cannot handle in this analysis model, so we will just decide with the worst case. Telling that a function call can be non-deterministic, or telling that we cannot analyse it, are two different kinds of information.

Order relation, least upper bound: The order relation between elements of this abstract domain is defined by the following rule:

$$\perp_D \leq det \leq nondet \leq \top_D$$

The definition of the least upper bound operation can directly be derived from it.

7.2.4 Dependency between the abstract domains

The three abstract domains are not totally independent. We're mainly interested in the determinism annotation of some function. If a function appears to be non-deterministic, or if it's determinism annotation is set to $\top_D$, then the abstract value of the result and the abstract success substitution of the concerned function is of no interest for us anymore. Indeed, supposing that this particular function is part of another function, then the second one will be forcibly nondeterministic too, independently of the abstract values of the result and the arguments of the first function.

An analogue relation between the three abstract domains exist when δ is set to $\bot_D$.

The next table shows the values of the abstract results, and abstract success substitutions depending on the value of δ:

δ	ρ_{out}	β_{out}
$\bot_D$	$\bot_\rho$	$\bot_\beta$
$nondet$	$\top_\rho$	$\top_\beta$
$\top_D$	$\top_\rho$	$\top_\beta$

7.3 Primitive operations

In this section we will define the different primitive operations[2]: procedure entry, procedure exit, abstract interpretation of $X = Y$, abstract interpretation of $X == Y$, abstract interpretation of partial functions, and finally abstract interpretation of $X@Y$.

7.3.1 Procedure entry

This operation is performed whenever a rule is to be applied. All possible relevant functional dependencies are worked out. This procedure just prepares the field for an abstract interpretation process of the right hand side of the rule that is used.

As input we have:

- the current goal G.
- the abstract substitution β_{in} over the variables $\{X_1, \ldots, X_n\}$ appearing in G.

[2] Instead of clearly theoretically speaking about the procedures, some implementation details will already be taken into account. One of these details is the representation of the variables. Variables which will be taken up in the abstract substitutions are implemented as free logic variables. With such a representation, renaming of variables is not necessary. So in the presentation of the procedures, everything concerning eventual renaming is omitted.

- the rule ($lhs \rightarrow rhs$) to be applied.

It will produce as output:

- a new goal NG.
- a new environment V of functional dependencies.
- an abstract substitution β_{new} over the variables appearing in NG and V, based on the abstract substitution β_{in}.

The different actions undertaken by this operation are:

1. **initialisation:** A new abstract substitution β_{init} is created, with β_{init} being an empty list.
2. **unification:** The arguments appearing in the initial goal are unified with the arguments in the left-hand side. The arguments of the initial goal are always variables, whereas the arguments with which they should be unified can be either variables too, or constructors. The correct abstract values must be deduced just like with normal unification occuring in logic languages. (see example 7.1). Nevertheless, entries into the abstract substitution are deduced from the given abstract substitution β_{in} and added to β_{init}, yielding β_1.
3. **normalisation:** A new environment V is produced by normalising the right-hand side of the rule.
4. **updating the abstract substitution:** The entries $X = Y$ in V where X is a fresh auxiliary variable, and Y is a ground expression are deleted from V, and the entries $\{X := g\}$ are added to β_1 yielding β_2. For all auxiliary variables Z_i that are not yet present in the abstract substitution, the corresponding entry $\{Z_i = a\}$[3] is added to β_2. Herewith, a new abstract substitution β_{new} is obtained.

The following example should make clear what is meant here:

Example 7.1
Let

$$
\begin{array}{rcl}
F & = & f(X_1, X_2, X_3) \\
\beta_{in} & = & \{X_1 := g, X_2 := a, X_3 := a\} \\
rule & : & f(c(A), B, C) \rightarrow g(h(A, B, C), 1).
\end{array}
$$

[3] There should be some way to distinguish these a priori values given to the auxiliary variables, in order to know whether the corresponding function has already been analysed or not. One way of doing this would be to introduce a new abstract value (e.g. a^*). Another possibility could be to maintain a list of not yet analysed auxiliary variables. We could also just clean up V when needed. At this stage we will not go into details of how to represent this relation. In appendix A, more will be said about this.

Then $procedure\_entry(F, \beta_{in}, rule) = (NewGoal, V, \beta_{new})$, where

$$\begin{aligned} NewGoal &= g(Y_1, Y_2) \\ V &= \{Y_1 = h(A, X_2, X_3)\} \\ \beta_{new} &= \{Y_1 := a, Y_2 := g, A := g, X_2 := a, X_3 := a\} \end{aligned}$$

7.3.2 Procedure exit

The procedure exit operation in our model is not only used when the analysis of some subgoal is to be summarised and rewritten in terms of the environment in which it was called, but is used more generally at those analysis points where the analysis returns from some abstract OR-node.

This means that we have as input the following parameters:

- The original environment: $Goal$, β_{in}, V.
- The calculated results of the analysis of the branches of the abstract OR-node: this means in effect a list of triples $(\beta_i, \rho_i, \delta_i)$ with $i = 1 \ldots p$.

The produced output is a new triple: $(\beta_{out}, \rho_{out}, \delta_{out})$.

The actions undertaken by this operation can be described as follows:

1. **lub of β_i:** let $\beta = lub_\beta(\beta_1, \ldots, \beta_p)$.
2. **lub of ρ_i:** let $\rho_{out} = lub_\rho(\rho_1, \ldots, \rho_p)$.
3. **lub of δ_i:** let $\delta_{out} = lub_D(\delta_i, \ldots, \delta_p)$.
4. **extend β:** update β_{in} with β yielding β_{out}. This just means within this context to replace the original assignments $X_i := m_i \in \beta_{in}$, with the corresponding new assignments $X_i := m_i^{new} \in \beta$.

Example 7.2
Let

$$\begin{aligned} F &= f(X_1, X_2, X_3) \\ \beta_{in} &= \{X_1 := g, X_2 := a, X_3 := a\} \end{aligned}$$

Calculated list of results ($ListResults$):

i	β_i	ρ_i	δ_i
1	$\{X_3 := g\}$	g	det
2	$\{X_3 := g\}$	a	det

Then $procedure\_exit(F, \beta_{in}, ListResults) = (\beta_{out}, \rho_{out}, \delta_{out})$, where

$$\begin{aligned} \beta &= \{X_3 := g\} \\ \beta_{out} &= \{X_1 := g, X_2 := a, X_3 := g\} \\ \rho_{out} &= a \\ \delta_{out} &= det \end{aligned}$$

If for one of the triples, δ_i would have been *nondet*, we would not loose any information by setting β_{out} to $\top_\beta$ and ρ_{out} to $\top_\rho$ as the general goal will be non-deterministic too.

7.3.3 Abstract interpretation of "$X = Y$"

In order to correctly analyse the expression $X = Y$, we need as input parameters its calling abstract substitution β_{in}, and the environment V. Due to the normalisation process that must have preceded, we know that both X and Y are some variables, eventually associated with some auxiliary functions that need to be processed first. Depending on the exact meanings of X and Y (whether or not some secondary function is associated with it) we can distinguish different cases.

7.3.3.1 No pending functions left

This situation corresponds to simple unification, just like it is known in logic languages. The implicit unification spoken about earlier can be viewed as an example of this case.

Given an initial abstract substitution β_{in}, the abstract interpretation of $X = Y$ within these conditions yields an updated abstract substitution β_{out}, a result ρ_{out} for the expression, and a determinism annotation δ_{out}, where

- ρ_{out} is set to ground (in the concrete domain, such an expression can only fail, or yield valid as answer).
- δ_{out} is set to *det*, as unification is always deterministic.

β_{out} is calculated in two steps. First an intermediary β is computed. The following table shows how. The elements of the table are the abstract values of X and Y.

β_{in}		β	
X	Y	X	Y
g	g	g	g
a	g	g	g
g	a	g	g
a	a	a	a

β_{out} is then computed by just updating β_{in} with β, just like it is done in the procedure exit operation.

7.3.3.2 Only X is associated with an unevaluated function

Let $F(\overline{Z})$ be the corresponding function. In order to reduce the expression $X = Y$, F must be analysed first. With the given abstract substitution β_{in}, and the environment V, the analysis of $F(\overline{Z})$ can be started. This yields an updated abstract substitution $\beta_{out,Temp}$, an abstract result ρ_{Temp}, and a determinism annotation δ_{Temp}.

If δ_{Temp} has the value *det*, we can update the initial abstract substitution β_{in} with the information concerning the new state of the variables after the analysis of $F(\overline{Z})$ contained in $\beta_{out,Temp} \cup \{X := \rho_{Temp}\}$. This situation brings us back to the first case, where none of the arguments X or Y are associated with unevaluated functions, and we can proceed as in section 7.3.3.1.

If on the other hand δ_{Temp} should not be *det*, then we can deduce the correct values for the abstract result and abstract substitution of the interpretation of $X = Y$ by the dependence relation that exists between the abstract domains used (section 7.2.4).

Graphically this situation corresponds to an abstract AND-node, as presented in figure 7.1.

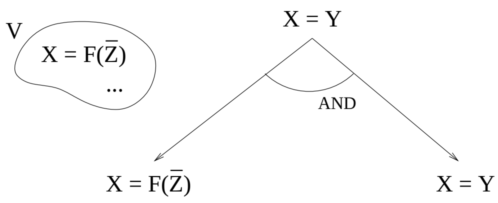

Figure 7.1: Analysis of $X = Y$ where $(X = F(\overline{Z})) \in V$

7.3.3.3 Only Y is associated with an unevaluated function

This is exactly the same situation as the previous one.

7.3.3.4 Both X and Y correspond to unevaluated functions

Let these functions be $F_X(\overline{Z}_X)$ and $F_Y(\overline{Z}_Y)$. Both of these functions will have to be analysed. The question arises: in what order? Well, just as we always create an abstract OR-node when no information concerning the exact analysis order of some functions is available, we'll also create at this place an abstract OR-node. In this case, each branch is constituted of an abstract AND-node. The situation is just as shown in figure 7.2.

After the analysis of both functions $F_X(\overline{Z}_X)$ and $F_Y(\overline{Z}_Y)$, the same simple situation of unification is obtained, a situation which has already been discussed earlier.

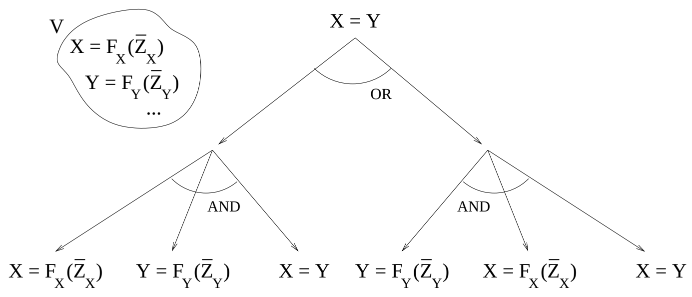

Figure 7.2: Analysis of $X = Y$ where $\{X = F_X(\overline{Z}_X), Y = F_Y(\overline{Z}_Y)\} \subseteq V$

7.3.4 Abstract interpretation of "$X == Y$"

The abstract interpretation of the strict equality function is similar to the abstract interpretation of $X = Y$, with this difference that the basic table defining the resulting abstract values associated with the input abstract values of X and Y is given by:

β_{in}		β	
X	Y	X	Y
g	g	g	g
a	g	a	a
g	a	a	a
a	a	a	a

If one of the arguments appears not to be ground, after the analysis of $X == Y$, both are set to "any".

Whenever one or both of the arguments of this operation is associated with some secondary function, a similar behaviour of the analysis as for the analysis of $X = Y$ will have to constructed (construction of an abstract AND-node, or of an abstract OR-node).

7.3.5 Abstract interpretation of partial functions

The way partial functions are analysed in our system is very simple. Due to lack of any kind of information about when and how such a function would be reduced at run-time, we'll have to set the abstract results to their respective top-elements. Whatever the initial abstract substitution β_{in} is, whatever V may contain, we have: $\rho_{out} = \top_\rho$, $\beta_{out} = \top_\beta$ and $\delta_{out} = \top_D$.

7.3.6 Abstract interpretation of "$X@Y$"

Again, the meaning of X and Y is recorded in the abstract domain, or in the pool V. Especially the meaning of X will be important here. Just like for the equational constraint, we will distinguish different cases.

7.3.6.1 X is not associated with an unevaluated expression

This situation in fact cannot occur. As X stands for some partial function, it would have been intercepted earlier in the abstract interpretation tree. In the previous section we assumed that the analysis of a partial function yields as results the top elements of each of the types of information collected.

7.3.6.2 Only X corresponds to an unevaluated function

Here we can distinguish two cases, that can be illustrated by the following examples.

Example 7.3
Suppose $X@Y$ with $\beta_{in} = \{X := a, Y := g\}$ and $V = \{X = fun\}$, where $fun/1$ is a defined function. In this case all necessary information to proceed with the calculations is present. The only thing to do is to add Y to the set of arguments of the function associated with X, and just analyse the resulting expression. In this case the analysis of $X@Y$ would just mean to analyse $fun(Y)$.

Example 7.4
Suppose $X@Y$ with $\beta_{in} = \{X := a, Y := g, Z := g\}$ and $V = \{X = fun2(Z)\}$, where $fun2/1$ is a function defined by the rules: $fun2(A) = g(A)$ and $g(A, B) = (\ldots)$. Here, $fun2$ itself is not a partial function, but its result is one. Again, we cannot say anything about the result of $fun2$, and we will have to approximate the result of the analysis of $X@Y$ by the worst case: top.

So we have:

- *X is associated directly with a partial function F.* We can just append Y to the list of arguments with which F is called, and proceed with the analysis as if nothing was.
- *X is associated with any other function F.* The analysis produces β_{out}, ρ_{out} and δ_{out}, with the respective values $\top_\beta$, $\top_\rho$ and $\top_D$.

7.3.6.3 Both X and Y correspond to unevaluated functions

As in the previous case the exact meaning of Y was unimportant, the analysis of this case can be handled as in 7.3.6.2. The evaluation of the function of Y will occur according to the definitional tree of the function associated with X.

7.4 Analysis of *branch*-nodes

Up to now, we've only seen how to manipulate the abstract information through the different primitive operations. But, when looking closer at the definition of these primitives, one might have noticed that decisions regarding determinism are never actually made.

In order to do so, some slight changes to the abstract interpretation framework must be done. As we have seen, a first situation for which non-determinism might occur is at the level of *branch*-nodes. The analysis of such *branch*-nodes will be reviewed here. The next section is concerned with the analysis of *or*-nodes.

In the original framework, the abstract interpretation of some function defined by some *branch*-node was defined by (page 64):

$\textbf{analyseWithTree}(Goal, V, \beta_{in}, branch(\_, Pos, Mode, ListT)) \rightarrow \rho, \beta_{out}$
$\quad\beta_{new} = \text{pre\_entryBranch}(Goal, V, \beta_{in}, Pos, Mode, ListT)$
$\quad(List\_\rho, List\_\beta_{out}) = \text{analyseForAllTrees}(Goal, V_{new}, \beta_{new}, ListT)$
$\quad(\rho, \beta_{out}) = \text{procedure\_exit}(\beta_{in}, List\_\rho, List\_\beta_{out})$

The original purpose of **pre_entryBranch** was just to analyse the function that is eventually associated with the argument appearing at the mentioned position, creating in fact a simple abstract AND-node at this level. In the case of determinism analysis, we want a somehow different behaviour, as we want the result of the analysis to be *nondet* depending on the nature of the *branch*-node, and the abstract value of the argument associated with the concerned position.

7.4.1 Redefinition of "pre_entryBranch"

The new role of this procedure is partly the same as the old one. If some function is associated with the position defined in the *branch*-node, this function should be analysed first. But after the eventual analysis of the secondary function, it should compare the mode of the *branch*-node, "rigid" or "flex", with the obtained abstract value of the argument.

This procedure can be therefore defined as follows:

$\textbf{pre\_entryBranch}(Goal, V, \beta_{in}, Pos, Mode, ListT) \rightarrow \beta_{out}, \delta_{out}$
$\quad Arg = \text{getArgAtPos}(Goal, Pos)$
$\quad Info = \text{getInfoArg}(Arg, V, \beta_{in})$
$\quad RestInfo = \text{getAllRestInfo}(Info, \beta_{in})$
$\quad(\rho_{Temp}, \beta_{Temp}, \delta_{out}) = \text{matchInfo}(Info, RestInfo, Mode, ListT)$
$\quad$ if $\delta_{out} = det$
$\quad$ then
$\quad\quad \beta_{out} = \text{update}(Arg, \beta_{in}, \rho_{Temp}, \beta_{Temp})$

else
  if $\delta_{out} = \perp_D$
  then
    $\beta_{out} = \perp_\beta$
  else
    $\beta_{out} = \top_\beta$

Here **pre_entryBranch** just filters all the information needed regarding the particular argument appearing at position Pos in the initial goal. The actual analysis of an eventual secondary function, and the matching of all this information with the mode defined in the *branch*-node, is left over to the procedure **matchInfo**.

7.4.2 "MatchInfo"

The role of **matchInfo** is to analyse a secondary function, if one should exist, and to calculate a new abstract value of the concerned argument (ρ_{Temp}), a new abstract substitution β_{Temp} concerning the variables of the eventual secondary function, and to produce a correct determinism annotation δ_{out}.

Depending on whether the information regarding the particular argument is a function or not, we have two different behaviours for this procedure.

7.4.2.1 *Info* $= g$ or *Info* $= a$

Due to the fact that there is no function to evaluate, **matchInfo** will return as abstract result the same information that was already available for the argument. We therefore have: $\rho_{Temp} = Info$. The value of *RestInfo* does not matter here (actually it always corresponds to the empty list), and β_{Temp} will always be set to [ ] (empty list). Only the value of δ_{out} is related to the input parameters of this procedure. The behaviour of **matchInfo** in this case can be summarised in the following table.

Info	$Mode$	$ListT$	δ_{out}
g	$rigid$	_	det
a	$rigid$	_	det
g	$flex$	_	det
a	$flex$	$[T]$	det
a	$flex$	$[T_1, \ldots, T_n]$	$nondet$

When the mode of the *branch*-node is "rigid", the computation in the concrete domain will always be deterministic. If this mode is "flex", then the computation will be deterministic only if the argument is ground, or if it is any, but there is only one definitional tree defining the subsequent calculations.

7.4.2.2 *Info* = some goal

When *Info* actually corresponds to a goal, this goal should be analysed first. If this analysis appears to result in $\delta = det$, then a new call to **matchMode** can be made with the updated value of the concerned argument. This new call will be such that the information regarding the argument will be simply ground or any, so that the previous case is applicable. When the determinism annotation should not be *det*, then by the dependency relation between the determinism information and the abstract substitutions, the new abstract result substitution can be easily deduced.

7.5 Analysis of *or*-nodes

The second case where non-determinism should be signalled is the case of an *or*-node appearing in the definitional tree of some function. The adaptation of the analysis framework consists of simply changing the definition of **analyseWithTree** where the tree is an *or*-node (page 64) to a simpler version where the output consists of the top-elements of each of the abstract results.

analyseWithTree$(Goal, V, \beta_{in}, or(T1, T2)) \rightarrow \top_{\rho}, \top_{beta}, \top_{D}$

7.6 Examples

The following examples will show how determinism analysis is performed. For some function calls the determinism analysis will yield precise results, for some others we will have to settle with imprecise results.

Example 7.5

$$\begin{array}{lll} nondet & :: & io(\ ). \\ nondet & if & generate(X) = putChar(X). \\ generate & :: & Char \rightarrow Bool. \\ generate(a) & = & true. \\ generate(b) & = & true. \end{array}$$

Definitional trees:

$$\begin{array}{ll} nondet: & rule(nondet \rightarrow cond(generate(X), putChar(X)). \\ cond: & branch(cond(A, B), 1, rigid, [rule(true, B) \rightarrow B]). \\ generate: & branch(generate(X), 1, flex,\ [generate(a) \rightarrow true, \\ & \qquad\qquad\qquad\qquad\qquad\quad generate(b) \rightarrow true]). \end{array}$$

Figure 7.3 shows the constructed abstract AND-OR tree which allows us to state that indeed, the function call nondet is possibly non-deterministic. In this case, the analysis yields a precise result.

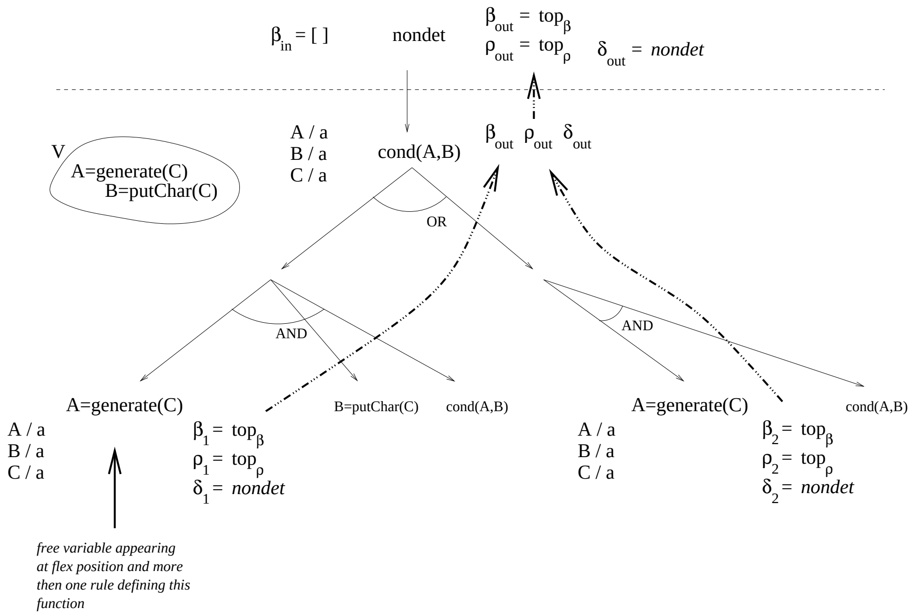

Figure 7.3: Abstract Interpretation of *nondet*

The analysis of *nondet* is equivalent to the analysis of $cond(A, B)$ with $\beta = \{A/a, B/a, C/a\}$ and $V = \{A = generate(C), B = putChar(C)\}$, which explains the horizontal line signalling the procedure entry operation. As the second argument of *cond* is lazy, we have to create an OR-node, which reviews two possibilities: B is analysed before the actual call to *cond*, or B is ignored. For each of these OR-branches, an AND-node is created. For each of these AND-nodes, $A = generate(X)$ is analysed first. Each time this analysis yields *nondet*. This result is sufficient to break up the analysis of the further branches in the AND-nodes. The procedure exit operation is finally applied to the results of the branches of the OR-node. One more procedure exit operation is required to cross the level-border, i.e. to translate the obtained results in terms of the initial abstract function call (in this case, the results remain $\top$).

The next example will show that a resulting determinism annotation $\delta_{out} = nondet$, does not forcibly mean that the actual concrete computation will be non-deterministic.

Example 7.6

$$
\begin{array}{lcl}
det & :: & io(\ ). \\
det & = & solution(2). \\
solution & :: & Nat \rightarrow io(\ ). \\
solution(1) & = & putChar(a). \\
solution(X) & = & putChar(b).
\end{array}
$$

Definitional trees:

$$\begin{array}{ll} det: & rule(det \rightarrow solution(2)). \\ solution: & or\ (branch(solution(A), 1, rigid, \\ & \qquad\qquad [rule(solution(1) \rightarrow putChar(a))], \\ & \quad rule(solution(B) \rightarrow putChar(b)). \end{array}$$

In the concrete domain, we would have the following reduction process of the call *det*:

$$\begin{array}{rcl} det & \rightsquigarrow & solution(2). \\ & \rightsquigarrow & putChar(b). \end{array}$$

This is clearly deterministic.

In the abstract domain though, after a first procedure entry step, the goal to be analysed is $solution(X)$ with $\beta_{in} = \{X := g\}$. The function *solution* is defined by an *or*-node, which as we have seen leads the analysis process to directly conclude that the function will be non-deterministic. Is this result false? No, it isn't. As we have seen, *nondet* is more general then *det*, which means that even deterministic functions are included in the set of functions adorned with the determinism annotation *nondet*. The obtained result is just less precise than we would have wanted it to be.

In appendix C, some examples of using the analysis tool are shown.

7.7 Conclusion

Determinism is an undecidable property. We can only define conditions for which non-determinism might occur. So, whatever analysis is performed, whatever abstract domains are used, there will always be cases where possible non-determinism is detected, but where in fact no true non-determinism is present at run-time. We will have to live with that.

This chapter showed the use of the abstract interpretation framework developed in the previous chapters to analyse for non-determinism. Simple abstract domains for the substitutions and the results are used as a basis. Perhaps too simple, through which we might greatly decrease the analysis precision. The following chapter will discuss how one can increase the precision of the analysis.

Chapter 8

Problems - Limitations

The determinism analysis, although correct, yields in certain cases imprecise results. This imprecision has two components. There are some imprecisions inherent to the abstract interpretation framework developed. But also the simplicity of the abstract domain used decreases the precision one would like to obtain.

In this chapter some of the major problems inherent to the framework, and related to the abstract domains used will be overviewed. Some possible solutions will be presented.

8.1 Imprecise analysis framework

The inherent imprecision in the abstract interpretation framework proposed is mainly due to the lazy computation strategy of the language interpreted. We can summarise the occurrences of laziness into three cases:

- Lazy positions within a function definition.
- Expressions are only reduced to the degree needed.
- Higher-order functions.

We will review here the way they were handled, and propose eventual improvements.

8.1.1 Lazy positions

We handled the first case by providing a correct way of interpreting the arguments of a function call eagerly. Some information can be lost with this strategy as all the possible orders of evaluation of these arguments must be considered.

Yet, the information about which orders can actually occur, could be derived in a preliminary analysis phase.

Example 8.1
Suppose we have the functions:

$$\begin{array}{lcl} f(X,Y) & = & g(h_1(X,Y), h_2(X,Y)). \\ g(A,B) \quad \mathit{if}\ A & = & \ldots \end{array}$$

Although the definition of g does not explicitly state in what order the arguments should be performed whenever bound to some function, by just looking at the right-hand side, we may conclude that the first argument will be the first one computed.

So the information of the possible run-time sequences could be derived in a preliminary phase. Each function could be annotated by some extra information saying more about the lazy positions.

The framework proposed needs no fundamental adaptation. The only procedure to change is *getAllSequences* (page 63). Instead of producing all possible orders, it will extract the needed information collected in the first analysis phase.

8.1.2 Reduction degree

The eventual reduction of expressions in head-normal form is directed by the function which depends on it. We could view this as a communication that takes place between the calling function, and the depending expression. Each time the expression is reduced with one step, a query is made to the calling function. If no further reduction is needed, then the job is done. If the expression is not instantiated enough, a further reduction step will be performed.

This behaviour should now be fit into a clear top-down analysis framework, where the information flow is limited. At the moment the abstract tree gets extended, some initial information is passed downwards (*procedure entry*). Only when the analysis of the extensions of the abstract tree are completed, information is passed upwards. At one single node, information can only pass twice: once downwards, and the second time upwards (see figure 8.1).

How do we approximate the "conversation" that actually takes place at run-time? How do we formulate the informations that can be passed across that line during such a communication process in just one information pass? We must keep in mind that across such a limit, no information about new variables may be provided.

There are three ways to handle this.

The safe way. The safest way is by approximating the conversation by no communication at all. The resulting success approximation is just set to the worst case. This is the way we handled this in the framework proposed.

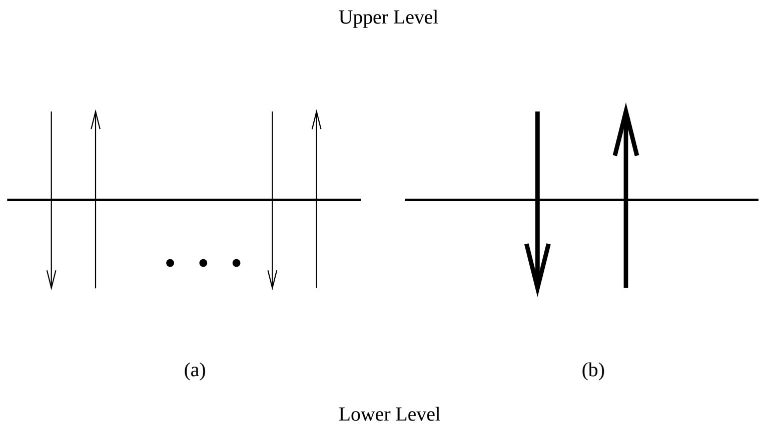

Figure 8.1: Communication between levels: a) in a concrete computation; b) in the abstract interpretation framework

The easy but risky way. Another option is to "trust" the definitional trees in such a way that the analysis considers as if every possible head normal form encountered is meant to be reduced to its normal form.

Let us review this based on the most classical example within the world of abstract interpretation: the function *append*.

$$\begin{array}{lcl} append([\,], Y) & = & Y. \\ append([X|Xs], Y] & = & [X|append(Xs, Y)]. \end{array}$$

With the safe approach, the function call $append(X, Y)$ with some abstract substitution defined on the variables would be interpreted as an eventual non-deterministic function. Indeed, arrived at the stage where the right-hand side of the second rule should be analysed, there is no way to tell whether the function appearing in the constructor-based expression will be evaluated at run-time or not.

The risky approach would consider the function as if it would have been written by the following rules:

$$\begin{array}{lcl} append([\,], Y) & = & Y. \\ append([X|Xs], Y] if Z & = & append(Xs, Y) = [X|Z]. \end{array}$$

The analysis of the function call $append(X)$ with X ground would then correctly derive that the associated concrete computation will be performed deterministically.

Yet, not all of the functions may be interpreted in such a naive way. In [12], where abstract interpretation of a functional logic language based on innermost narrowing is discussed, the authors provide definitions which allow to clearly determine whether a function can be interpreted naively (such classes of functions are known as functions which preserve locality), or if special care must be taken.

Due to lack of time, the theories developed there could not be applied into the abstract interpretation framework which was developed here.

Providing defaults. A third possible solution in increasing the general precision of the analysis system consists of providing the system with determinism information by hand.

While analysing, the tool consults and constructs a memo-table, which contains all the actual information about functions that are analysed at the moment, and functions that already have been analysed. We could just add explicitly some functions to this memo-table.

We could for example provide Curry with a new type of entries. The following example shows how this could be done.

```
append :: list(A) -> list(A) -> list(A).
append([ ],Y)     = Y.
append([X|Xs],Y) = [X|append(Xs,Y)].
determinism append(g,g) = g ; (g,g) ; det.
```

The new line could then be interpreted as follows: If *append*/2 is called with both arguments ground, then the result will be ground, the arguments will remain ground, and the computation will be deterministic.

The analysis tool can then use this information to derive preciser information regarding calls which depend on this function. But the results obtained must then be handled with care, as one has to be sure that the determinism information provided is correct.

8.1.3 Higher-order functions

The way we handled higher-order functions was also quite drastic. If the information about the exact partial function that is to be applied can directly be known, then the abstract interpretation poses no problem. On the other hand, when a higher-order function is encountered which can not just be analysed by appending the variable to which it should be applied to its list of arguments, we ended the analysis by stating that the success abstract information is $\top$.

In this area work is going on to transform higher-order functionalities to simple first-order functions, so that a normal analysis tool like the one developed here can be applied on it, yielding therefore preciser results, as no higher-order functions exist in such a system.

8.2 Imprecise abstract domain

The determinism analysis proposed is based on an underlying groundness analysis. This groundness analysis knows only two abstract values: "ground" or "any". For this simplistic abstract domain, one very quickly arrives at no information at all, i.e. variables annotated with the abstract value "any". One could refine this abstract domain by adding a third abstract value "free", which should denote those variables which are free. Such a domain (freeness) mostly requires also sharing information, which makes the whole abstract interpretation process already quite more complicated. Even with a more precise abstract domain, one could be unsatisfied with the results obtained. More fundamental techniques such as re-execution [5, 15] could then be used.

Nevertheless, improving the precision of groundness analysis is a well discussed topic. I suppose that the techniques developed in that area can be in some way adapted to the framework proposed here.

8.3 Conclusion

The abstract interpretation process can be improved firstly by handling the imprecisions inherent to the analysis framework, as for example performing a pre-analysis. For some functions we could reduce the security level built-in into the analysis framework. The functions concerned by such a relaxation are not explicitly defined here. We could also just provide determinism information manually.

A second field for which precision could increase is to modify and complexify the abstract domain used.

Chapter 9

Conclusion

Curry is partially a functional language. Monadic I/O which is mainly used within lazy functional languages can therefore easily be adopted in Curry. But we have seen that, due to the logic nature of the language, non-deterministic evaluation steps within interactive programs must be avoided.

In order to detect such non-deterministic steps at compile-time, a top-down analysis framework was constructed. This framework in itself will produce bad results if the functions to be analysed are written using constructor-based expressions. Another major lack of precision is introduced by the approximation it makes when lazy function calls are to be analysed.

The first source of imprecision could be handled by either trusting the definitions, and pushing the analysis tool to analyse constructor-based expressions as if each of the expressions appearing in it will always be reduced, or by explicitly providing information on what the expected analysis results should be.

The lack of precision introduced by the eager analysis of lazy function calls could be reduced by performing a pre-analysis that determines the order in which some of these lazy positions will be reduced at run-time. This would not only increase the precision of the results obtained, it also would greatly decrease the size of the abstract AND-OR tree, and therefore decrease the analysis time.

Finally the developed framework was applied to the problem of detecting non-determinism. In fact, such an analysis is composed of two parts: groundness analysis, and on top of it, determinism analysis. The only way of improving the precision of the results obtained by this analysis consists in complexifying the abstract domains used.

Albeit, the analysis performs well for strict evaluation schemes, and ground expressions.

The main goal of this thesis was to provide monadic I/O within Curry. However, the biggest part of the time was spent on developing the generic abstract interpretation framework, and implementing the determinism analysis tool. Yet, this work is not really complete, as true correctness proofs of the framework and its application are not included. This is mainly due to simple lack of time.

Well, and now the final word. Did I really enjoy this work? Yes and no. Some parts were interesting, some other weren't. Sometimes the stress was high, and at such moments nothing is fun. But I did learn one thing: I started to truly love declarative programming.

Nancy Mazur
August 26, 1997

Appendix A

Implementation details

The prototype `TasteCurry` on which my implementation is based, as well as my own implementation, are written in `SICStus Prolog, version3`.

A.1 Monadic I/O

Providing I/O with monadic I/O was as simple as pointed out in chapter 2. The main monad combinators `>>` and `>>=` are written in Curry itself. The other basic combinators, `done` and `return(A)`, as well as some primitive I/O commands are declared in Curry as external, and implemented in Prolog.

A.2 Determinism Analysis

The second part of my work consisted in developing an abstract analysis framework, and implementing it for the specific case of determinism analysis. I'll just quickly overview the general structure of my code, some global considerations on the representation of the data, and a final note on the efficiency of the code.

A.2.1 General Scheme

The implementation consists of five global modules, that can eventually be extended to six. Figure A.1 shows the general structure of my implementation.

curry.pl This module can be seen as the input to my system. It is the basic Curry implementation. The only functionality **analyse.pl** uses from this module, is that it reads in the Curry-programs, and translates the operational behaviour into definitional trees.

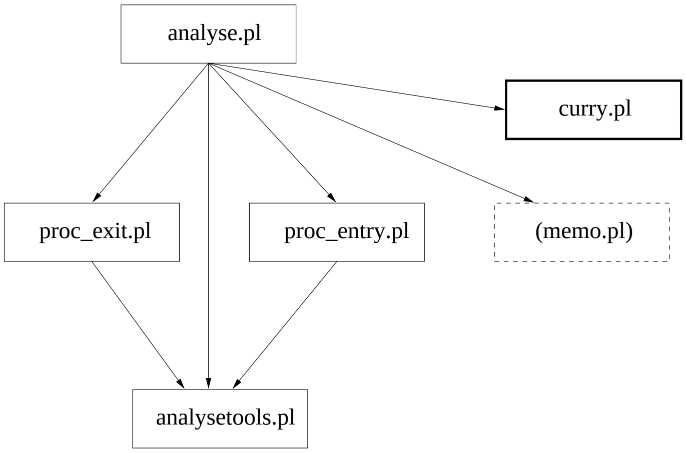

Figure A.1: General scheme of the analysis tool

analyse.pl This is the top-level module to my system, it builds up the abstract AND-OR trees, and deduces the correct data-flow along these trees.

proc_entry.pl Some of the work of the main module is delegated to some specific modules, such as this one, which contains the code of the primitive operation "procedure_entry".

proc_exit.pl This part is only concerned with the primitive operation "procedure_exit".

analysetools.pl It simply contains some helping tools: the manipulation of lists, more specifically of the abstract domain, predicates to write the results to standard output, etc...

(memo.pl) This module could be used to provide the system with predefined entries to the memo-table.

The implementation of the analysis tool is done according the formal description presented in chapter 6. Yet, this description didn't present how fix-point computation is considered. In the implementation, concerns about fix-point calculations are done on top of the actual analysis.

A.2.2 Representation of the abstract domains and V

As the implementation is done in Prolog, we can represent each variable occuring in the goal that has to be analysed, by unique free logic variables. Within one analysis tree, each distinct variable can be associated with a unique free logic variable. This implies that no renaming is required when a "procedure_entry" or "procedure_exit" occurs.

In the theoretic discussion about the abstract analysis of Curry programs, a clean separation between the abstract domain and the pool of auxiliary functions V was made. The only requirement was that there should be a link between both. The auxiliary variables introduced in V are just added to the domain of the abstract substitutions. Once an auxiliary function gets analysed, the abstract value of the corresponding variable must be updated. Care should then be taken, that the same auxiliary function is not analysed a second time in the same abstract subtree.

In the actual implementation the management of the link between V and the abstract domain is strongly simplified. Instead of keeping three abstract domains (ρ, β and δ, plus a pool of functions, we've simply added V to the abstract substitutions. The following example illustrates this.

$$\begin{array}{rcl} V & = & \{X = f(Y)\} \\ \beta & = & [\,(X/a^*)\,,\,(Y/g)\,] \\ \\ & \Downarrow & \text{becomes} \\ \\ \beta & = & [\,(X/f(Y))\,,\,(Y/g)\,] \end{array}$$

This notation guarantees that no auxiliary function will be analysed twice within the same subtree. At the moment an auxiliary function gets analysed, the information of the variables gets updated. Further analysis of the main expression will then be based on this updated information.

A.2.3 Memo-table

The entries of the memo-table are implemented as dynamic predicates which are asserted and retracted when needed. The structure of such an entry is as follows:

memo(Annot, Goal, Bin, R, D, Bout, FixP).

Annot This argument can have two different values: "user" or "system". If it is set to "user" then the memo-entry concerns a user-defined function, which can be manipulated freely[1]. This is the most general case. The "system" annotation denotes functions for which determinism information is provided by hand. They cannot be retracted, as they are always valid.

Goal This is the actual goal for which the determinism information is recorded.

[1]For each new function to be analysed, these memo-entries are retracted.

Bin Associated to each goal that is analysed, we have the corresponding abstract substitution.

R This argument determines the abstract result of the goal. The initial value of this parameter is `bottom`. This value is needed when a fix-point computation should take place.

D The value of this parameter concerns the determinism annotation for the given goal. As "R", it is initially set to `bottom`

Bout After the analysis of some goal, the abstract success substitution must be recorded. This is the role of this argument. Just as the preceding two arguments, initially it's value is set to `bottom`.

FixP The information recorded in this argument is the central information needed for the fix-point computation to be performed correctly. It can have the following values:

- "analysing": For each new function to be analysed, a corresponding memo-entry is made, with **R**,**D**, and **Bout** set to `bottom`, and **FixP** set to analysing. Whenever further in the analysis the same function is encountered, the memo-table is consulted, and a fix-point computation may start.
- "complete": This simply states that the analysis is completed.
- "fixpoint": A memo-entry set to "fixpoint" signals that a fix-point computation is being performed, centred on the goal specified.

In fact one could think of a fourth value for this parameter, "a-nalysing", which would signal that the results obtained for some function are to be handled with care as these results depend on the consecutive approximations of the recursive functions they rely on. But whenever a new approximation is made, these functions must be re-analysed from scratch. Thus, whenever an analysis of a function yields an approximate results, without the function itself being directly recursive, the memo-entry may just be retracted, as the particular goal will have to be re-analysed anyway.

A.2.4 A note on the efficiency

The main intention of the code was to provide correct results. No concerns were made about user-interface, optimisation, or efficiency. Yet, it is worth spending a few words on the efficiency obtained.

The efficiency is directly related to two aspects of the goal to be analysed:

1. the size of the abstract AND-OR tree that will have to be analysed.

2. the size of the data flow that has to be managed along this tree.

The size of the data flow, i.e. the amount of information that has to be carried along the tree, cannot directly be influenced. It simply depends on the definition of the function that is to be analysed.

But the size of the abstract AND-OR tree can in some cases be reduced, if more information is available.

There is mainly one aspect of the analysis framework that has an enormous impact on the size of the constructed abstract tree. We've seen that whenever lazy function positions occur, an OR-node must be created such that all possible combinations, all possible orders for the analysis of the remaining functions must be considered.

In a first implementation of this analysis tool, I mistakenly considered only the permutations of the remaining functions at such OR-nodes (instead of considering all the permutations of all the subsets of these positions). Having corrected the error, the abstract AND-OR trees that had to be considered, had literally grown exponentially.

The solution to this is to pre-analyse such lazy positions, and to reduce all the possible permutations, to some few realistic ones. This is also the solution proposed to yield preciser results (chapter 8).

Appendix B

Manual to the analysis tool

The analysis tool is based on the latest version of TasteCurry (June 25, 1997). No real trouble should occur when using the system on newer versions of Curry, as long as the internal representation by means of definitional trees remains the same.

B.1 Getting started

The first step consists of starting SICStus Prolog, and compiling the analysis tool:

```
SICStus 3  #3: Tue Dec 10 11:31:16 MET 1996
| ?- compile(analyse).
```

In order to analyse a specific function from a specific Curry program, this program must first be translated in terms of definitional trees. This can be done by first calling the TasteCurry prompt, and reading in the program files.

```
| ?- cs.

TasteCurry Interpreter (Version of 25/07/97 by
Informatik II, RWTH Aachen)

Type "help." for list of available commands
|: read myfile.
```

The analysis tool is only accessible from outside TasteCurry itself. So you must leave the curry prompt (exit.). Herewith, the SICStus prompt reappears. All the definitional trees of the needed functions should now be in the Prolog database and the actual analysis can be started.

B.2 Performing an analysis

There are basically two ways of starting the determinism analysis of particular functions, depending on the way the user wants to specify these functions.

B.2.1 Explicit groundness information

The user can give groundness information over the variables occuring in the goal explicitly. This can be done using the `analysemode` procedure.

Suppose a function $f/2$ has to be analysed, with it's first argument ground, and it's second argument being anything. The correct way to start the analysis is as follows:

```
| ?- analysemode(f(g,a)).
```

If mistakenly one of the arguments of f has some other value then the expected ones (`g` or `a`), then an error is produced.

If everything works fine, the tool prompts by repeating the goal to be analysed, in the way it is represented internally (a goal with free logic variables, and an abstract entry substitution accompanying this).

B.2.2 Implicit groundness information

Another approach consists of calling the analysis tool with a possible concrete function call by using the procedure `analyse`. The concrete function call must be simple though, which means that no nested expressions are allowed.

The following inputs for the analysis of the same f in the same situation as above are all valid. They will all be abstracted to the same abstract goal, namely $f(X,Y)$ with an abstract entry substitution: $\{(X/g),(Y/a)\}$.

```
| ?- analyse(f(1,X)).
| ?- analyse(f(s(1),s(X))).
| ?- analyse(f(true,_)).
```

Note that the types of the given goal are not checked. The only thing that matters here is to see whether the arguments are ground or any.

As above, the tool will prompt by repeating the initial goal in its abstract version.

B.3 Interpreting the results

An analysis session may look like this:

```
| ?- analysemode(h(g,g)).
Goal: h(_213,_224)
Bin:  [_213/g,_224/g]
Analysing with given modes......done.
Results of the analysis:
        Result: g
        Det: det
        Bout: [_213/g,_224/g]

yes
| ?-
```

The first line is of course the analysis-command itself. As said, the analysis starts by repeating the initial goal, with its abstract entry substitution. After a while, the results are shown.

The result information of the analysis is made up of three parts:

1. **Result**: shows the calculated abstract result of the goal (ground, or any).

2. **Det**: shows the determinism information deduced. It can have the values: det (deterministic), nondet (non-deterministic), or top (unable to analyse).

3. **Bout**: represents the abstract success substitution. The variables can be identified by looking at the abstract goal, and the abstract entry substitution.

B.4 More results

Sometimes it is useful to see the determinism results of the functions that had to be analysed in order to deduce the results for the main function. This can be done by consulting the memo-table with the command `showmemo`.

A possible output of `showmemo` could be:

```
memo(user, generate(A), [A/a], a, nondet, top, complete).
memo(user, B=C, [B/generate(A),A/a,C/g], ...
```

These are simply the memo-entries. The interpretation of each of the fields in a memo-entry was already provided in appendix A, at page iii[1].

It is possible that some functions, that actually had to be analysed, do not occur in the memo-table. Functions for which the abstract results are only approximations, and which themselves are not recursive, are deleted

[1]Note that the abstract substitution (third field) may contain an auxiliary function (appendix A).

from the memo-table. The reason why this is done is that, as long as the fix-point of the function on which they depend is not reached, they will have to be re-analysed anyway (see appendix A).

Appendix C

Determinism Analysis - Examples

In this last appendix we'll review some examples. Some of these examples will yield fine results, other will be more imprecise.

C.1 nondet

We'll start with the classic example of a non-deterministic I/O function. The code of this function is given as follows:

```
nondet if generate(X) = putChar(X).
generate(98) = true.
generate(99) = true.
```

The transcript of the analysis session is:

```
| ?- analyse(nondet).
     Goal: nondet
     Analysing.....done.
     Results of the analysis:
             Result: a
             Det: nondet
             Bout: top

     yes
```

The result of this analysis clearly corresponds with what we wanted: non-determinism.

The functions that had to be analysed yielded as results (showmemo):

```
| ?- showmemo.
```

```
memo(user, generate(A), [A/a], a, nondet, top, complete).
memo(user, B=C, [B/generate(A),A/a,C/g], a, nondet,
              top, complete).
memo(user, cond(D,E), [D/(B=C),B/generate(A),A/a,C/g,
              E/putChar(A)], a, nondet, top, complete).
memo(user, nondet, [], a, nondet, top, complete).
```

```
yes
```

C.2 append

The definition of append is:

```
append([ ],Y) = Y.
append([X|Xs],Y) = [X|append(Xs,Y)].
```

This right-hand side of the second rule is a constructor-based expression, for which we know that the analysis yields bad results.

```
| ?- analysemode(append(g,g)).
Goal: append(_135,_146)
Bin:  [_135/g,_146/g]
Analysing with given modes..done.
Results of the analysis:
        Result: a
        Det: top
        Bout: top

yes
```

Here the determinism analysis yields `top` as determinism annotation. This is what we could have expected, as the meaning of `top` is that the analysis tool could not analyse the given function.

C.3 det

Let's view a deterministic version of the first example. In this example the derived abstract result will be "any" although the result will always be ground. We'll explain why.

```
det if generate(X) = putChar(X).
generate(98) = true.
```

Only one possible value of `X` exists. Hence the computation of `det` will be deterministic. The analysis tool correctly derives this information, as the transcript shows:

```
| ?- analyse(det).
Goal: det
Analysing......done.
Results of the analysis:
        Result: a
        Det: det
        Bout: []

yes
```

But why is the abstract result set to "any"? The reason is very simple. The `if`-construct is in fact syntactic sugar replacing the function `cond/2` defined as follows:

```
cond :: constraint -> A -> A.
cond eval 1:rigid.
cond({},X) = X.
```

This function is rigid in its first argument. It's second argument is not annotated, which in our terminology means that the second position is lazy. The analysis tool analysis such situations eagerly. But a priori it does not know whether the lazy positions will ever be evaluated or not. So it also considers the case where the second argument is just discarded (set to "any").

So the analysis of `cond/2` will produce an OR-node. The first branch will analyse `generate(X)`, which will be ground. And therefore this first branch will yield that the result of this function is ground. But the second branch of this OR-node will discard the function `generate(X)` and simply replace it with the abstract value "any". As the right-hand side of `cond` is equal to the second argument, in this case this argument will be "any", the result of the analysis of this branch will yield "any" as abstract result. The final abstract result is obtained by calculating the least upper bound of each of the abstract results obtained in the branches of the OR-node, yielding "any".

Bibliography

[1] H. Aït-Kaci, P. Lincoln, and R. Nasr. Le fun: Logic, equations and functions. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 17–23, San Francisco, August - September 1987. IEEE, Computer Society Press.

[2] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–279, Portland, Oregon, January 1994.

[3] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, 1991. revised version of K.U.L. technical report CW 62, 1987.

[4] M. Bruynooghe and D. De Schreye. Tutorial on abstract interpretation of logic programs. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seatle, 1988. ALP, IEEE, The MIT Press.

[5] M. Bruynooghe and G. Janssens. Propagation: a New Operation in a Framework for Abstract Interpretation of Logic Programs. Research Report CW 146, Department of Computer Science, KU Leuven, Belgium, March 1992.

[6] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL, Los Angeles, CA*, pages 238–252, January 1977.

[7] W. Hans and S. Winkler. Aliasing and Groundness Analysis of Logic Programs through Abstract Interpretation and its Safety. Technical Report 92-27, Technical University of Aachen (RWTH Aachen), 1992.

[8] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[9] M. Hanus. Teaching functional and logic programming with a single computation model. In *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*. Springer LNCS (to appear), 1997.

[10] M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.

[11] M. Hanus, S. Antoy, H. Kuchen, and F.J. López-Fraguas. Curry: an Integrated Functional Logic Language. Technical report, Technical University of Aachen (RWTH Aachen), June 1997. Available at http://www-i2.informatik.rwth-aachen.de/hanus/curry/report.html.

[12] M. Hanus and F. Zartmann. Mode Analysis of Functional Logic Programs. In Baudouin Le Charlier, editor, *1st International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 26–42, Namur, Belgium, September 1994. Springer Verlag.

[13] G. Janssens and W. Simoens. On the Implementation of Abstract Interpretation Systems for (Constraint) Logic Programs. In Y. Deville, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'93*, Workshops In Computing, Louvain-La-Neuve, Belgium, 1993. Springer-Verlag.

[14] J. W. Lloyd. Declarative programming in escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[15] K. Marriott and H. Søndergaard. On Propagation-Based Analysis of Logic Programs. In *ILPS Workshop on Global Compilation*, pages 45–65, Vancouver Canada, November 1993.

[16] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[17] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, 1990.

[18] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[19] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(1, 2, 3 and 4):315–347, 1992.

[20] J. Peterson[editor], K. Hammond[editor], L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, S. Peyton Jones, A. Reid, and P. Wadler. Haskell 1.3, A non-strict, purely functional language. Report YALEU / DCS / RR-1106, Department of Computer Science, Yale University, May 1996.

[21] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. Technical report, University of Glasgow, October 1992.

[22] Philip Wadler. Comprehending monads. Technical report, University of Glasgow, April 1992.

[23] Philip Wadler. The Essence of Functional Programming. In *19th Annual Symposium on Principles of Programming Languages*, January 1992.

[24] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*, NATO ASI Series F: Computer and Systems Sciences. Springer-Verlag, 1993.

[25] Philip Wadler. How to declare an imperative (invited talk). In *International Logic Programming Symposium*, Portland, Oregon, December 1995. MIT Press.