Two Non-Determinism Analyses in Eden

Ricardo Peña Clara Segura

03 October 2000

Technical Report nº 108-00
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid

Abstract

Non-determinism may affect the referential transparency of the programs written in Eden: If a non-deterministic expression is evaluated in different processes, the variable it is bound to it will denote possibly different values. It would be desirable to warn the programmer about this situation, or to force the evaluation of such an expression so that all the occurrences of the variable have the same value. Additionally there exist sequential transformations that are incorrect when non-determinism is involved. Such transformations should be applied only to those parts of the program that are sure to be deterministic. In this paper several analyses of different efficiency and power are presented. Several techniques are used: A types annotation system and abstract interpretation.

1 Introduction

The parallel-functional language Eden extends the lazy functional language Haskell by syntactic constructs to explicitly define and communicate processes. The three main new concepts are *process abstractions*, *process instantiations* and the non-deterministic process abstraction `merge`. Process abstractions of type `Process a b` can be compared to functions of type `a -> b`, and process instantiations can be compared to function applications. An instantiation is achieved by using the predefined infix operator `(#) :: Process a b -> a -> b`. Each time an expression `e1 # e2` is evaluated, a new parallel process is created to evaluate `(e1 e2)`.

Non-determinism is introduced in Eden by means of a predefined process abstraction `merge :: Process [[a]] [a]` which *fairly* interleaves a set of input lists, to produce a single non-deterministic list. Its implementation immediately copies to the output list any value appearing at any of the input lists. So, `merge` can profitably be used to quickly react to requests coming in an unpredictable order from a set of processes. This feature is essential in reactive systems and very useful in some deterministic parallel algorithms [KPR00]. Eden is aimed at both types of applications.

Eden has been implemented by modifying the *Glasgow Haskell Compiler* (GHC) [JHH+93]. GHC translates Haskell into a minimal functional language called Core where a lot of optimizations [San95, PS98] are performed. Some of them are incorrect in a non-deterministic environment. So, a non-determinism analysis is carried out at Core level and, as a result, variables are annotated as deterministic or (possibly) non-deterministic. After that, the dangerous transformations are disallowed if non-determinism is present.

The plan of the paper is as follows: In Section 2, we review some non-determinism approaches in functional languages. In Section 3 we study the non-determinism in Eden. In Section 4 the intermediate language CoreEden being analised is presented. In Section 5 a first analysis is presented. It is a type based analysis. This type annotation system corresponds directly to an abstract interpretation, presented in Section 6. At the end of this section we explain some limitations of this analysis and the reasons for them. In Section 7 a new abstract interpretation based analysis is presented. It pretends to overcome the limitations of the first analysis. It is more powerful but it is also more expensive. In Section 8 we prove that the first analysis is an upper approximation to a widening of the second analysis. In Section 9 some related work is given and also some ideas about a third analysis are presented. It pretends to be an intermediate analysis, both efficient and powerful.

2 Non-determinism in the functional languages

The introduction of non-determinism in functional languages has a long tradition and has been a source of strong controversy. John McCarthy [McC63] introduced the operator `amb :: a -> a -> a` which non-deterministically chooses between two values. Henderson [Hen82] introduced instead `merge :: [a] -> [a] -> [a]` which non-deterministically interleaves two lists into a single list. Both operators violate referential transparency in the sense that it is no longer possible to replace equals by equals. For instance,

```
let x = amb 0 1 in x + x  ≠  amb 0 1 + amb 0 1
```

as the first expression may only evaluate to 0 or to 2, while the second one may also evaluate to 1.

Hughes and O'Donnell proposed in [HO90] a functional language in which non-determinism is compatible with referential transparency. The idea is the introduction of the type `Set a` of sets of values to denote the result of non-deterministic expressions. The programmer explicitly uses this type whenever an expression may return one value chosen from a set of possible values. The implementation represents a set by a single value belonging to the set. Once a set is created, the programmer cannot come back to single values. So, if a deterministic function `f` is applied to a non-deterministic value (a set `S`), this must be expressed as `f * S` where `(*) :: (a -> b) -> Set a -> Set b` is the `map` function for sets. A limited amount of set operations are allowed. The most important one is $\cup$ (set union) that allows the creation of non-deterministic sets and can be used to simulate `amb`. Other, such as `choose :: Set a -> a` or $\cap$ (set intersection) are disallowed either because they violate referential transparency or because they cannot be implemented by 'remembering' one value per set. In the paper, a denotational semantics based on Hoare powerdomains is given for the language and a number of useful equational laws are presented so that the programmer can formally reason about the (partial) correctness of programs.

But the controversy goes further. In [SS90, SS92], the authors claim that what is really missing is an appropriate definition of *referential transparency*. They show that several apparently equivalent definitions (replacing equals by equals, unfoldability of definitions, absence of side effects, definiteness of variables, determinism, and others) have been around in different contexts and that they are not in fact equivalent in the presence of non-determinism. To situate Eden in perspective, we reproduce here their main concepts:

Referential transparency Expression e is purely referential in position p iff

$$\forall e_1, e_2.[\![e_1]\!]\ \rho = [\![e_2]\!]\ \rho \Rightarrow [\![e[e_1/p]]\!]\ \rho = [\![e[e_2/p]]\!]\ \rho$$

Operator $op :: t_1 \rightarrow \cdots t_n \rightarrow t$ is referentially transparent if for all expressions $e \stackrel{\text{def}}{=} op\ e_1 \cdots e_n$, whenever expression $e_i, 1 \leq i \leq n$ is purely referential in position p, expression e is purely referential in position $i.p$. A language is referentially transparent if so do all its operators.

Definiteness Definiteness property holds if a variable denotes the same single value in all its occurrences. For instance, if variables are definite, the expression $(\lambda x.x - x)(amb\ 0\ 1)$ evaluates always to 0. If they are not, it may also evaluate to 1 and -1.

Unfoldability Unfoldability property holds if $[\![(\lambda x.e)\ e']\!]\ \rho = [\![e[e'/x]]\!]\ \rho$ for all e, e'. In presence of non-determinism, unfoldability is not compatible with definiteness. For instance, if variables are definite $[\![(\lambda x.x - x)(amb\ 0\ 1)]\!]\ \rho \neq [\![(amb\ 0\ 1) - (amb\ 0\ 1)]\!]\ \rho$.

In the above definitions, the semantics of an expression is a set of values in the appropriate powerdomain. However, the environment ρ maps a variable into a single value in the case variables are definite (also called *singular semantics*), and to a set of values in the case they are indefinite (also called *plural semantics*).

3 Non-determinism in Eden

In Eden, the only source of non-determinism is the predefined process `merge`. When instantiating a new process by evaluating the expression `e1 # e2`, closure `e1`, together with the closures of all the free variables referenced there, are copied (possibly unevaluated) to another processor where the new process is instantiated. However,

within the same processor, a variable is evaluated at most once and its value is shared thereafter. We are still developing a denotational semantics for the language but, for the purpose of this discussion, we will assume that the denotation of an expression of type `a` is a (probably downwards and limit closed) set of values of type `a` representing the set of possible values returned by the expression. If the expression is deterministic, its denotation is a singleton. Under these premises, we can characterize Eden as follows with respect to the previously defined concepts:

Referential transparency Eden is referentially transparent. The only difference with respect to Haskell is that now, in a given environment ρ, an expression denotes a set of values instead of a single one. Inside an expression, a non-deterministic subexpression can always been replaced by its denotation without affecting the resulting set of values.

Definiteness Variables are definite within the same process and are not definite within different processes. When an unevaluated non-deterministic free variable is duplicated in two different processes, it may happen that the actual value computed by each process is different. However, denotationally both variables represent the same set of values, so the semantics of the enclosing expressions will not change by the fact that the variable is evaluated twice.

Unfoldability In general, in Eden we do not have the unfoldability property except in the case that the unfolded expression is deterministic. This is a consequence of having definite variables within a process.

The motivation for a non-determinism analysis in Eden comes from the following two facts:

- In future, Eden's programmers may wish to have definite variables in all situations. It is sensible to think of having a compiler flag to select this semantic option. In this case, the analysis will detect the (possibly) non-deterministic variables and the compiler will force their evaluation to normal form before being copied to a different processor.
- At present, some transformations carried out by the compiler in the optimization phases are semantically incorrect for non-deterministic expressions. In this case, they will be deactivated. They are *full laziness* [JPS96], the *static argument transformation* [San95] and the *specialization*. The general reason for all of them is the increasing of closure sharing: Before the transformation, several evaluation of a non-deterministic expression can produce several different values; after the transformation, a shared non-deterministic expression is once evaluated, yielding a unique value.

Lets see now in more detail the transformations mentioned above. In [PS00] a full study of the efects of all the transformations done in GHC over Eden constructions is done, not only from the point of view of the semantic correctness but also from the point of view of the efficiency. Here we only talk bout those affecting the semantics when non-determinism is involved.

Full laziness This transformation floats a *let* binding outside a λ-abstraction to share its evaluation between all the applications of the function:

$$\begin{array}{l}\textbf{let} \\ \quad g = \lambda y.\textbf{let } x = e \\ \qquad\qquad \textbf{in } e' \\ \textbf{in } \ldots\end{array} \quad \Rightarrow \quad \begin{array}{l}\textbf{let} \\ \quad x = e \\ \textbf{in let} \\ \quad g = \lambda y.e' \\ \textbf{in } \ldots\end{array}$$

This transformation is correct only if e does not depend on y.

From the non-determinism point of view, the problem arises when the floated binding is non-deterministic. As an example, lets consider the following definitions, that represent a function before and after the transformation:

```
f = λy.let x = e1          f' = let x = e1
    in x + y                    in λy. x + y
```

If `e1` is non-deterministic, the semantics of `f` is a non-deterministic function, that is, it returns a non-single set of values So $[\![f\ 5 - f\ 5]\!]\ \rho$ will deliver a non-single set of values as `x` is evaluated each time the lambda is applied. The semantics of the expression bound to f' is instead a set of deterministic functions and, due to the definiteness of variables `x` and `f'`, $[\![f'\ 5 - f'\ 5]\!]\ \rho$ evaluates always to $\{0\}$. So, the semantics has changed after the full laziness transformation.

The compiler would detect the non-deterministic bindings and disallow the floating of a `let` out of a lambda in these situations (see [PPRS00] for more details).

Static argument transformation This transformation is applied to recursive definitions. If there is an argument such that in all the recursive calls of the function its value is always the same, it is called a static argument. Then we can define a new function having the static argument as a free variable that behaves as the original function. Lets see an example:

$$
\begin{array}{l}
foldr\ f\ z\ l = \\
\quad \textbf{case}\ l\ \textbf{of} \\
\qquad [\,] \to z \\
\qquad (a : as) \to \textbf{let}\ v = foldr\ f\ z\ as \\
\qquad\qquad \textbf{in}\ f\ a\ v
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
foldr\ f\ z\ l = \\
\quad \textbf{let}\ foldr'\ l = \\
\qquad \textbf{case}\ l\ \textbf{of} \\
\qquad\quad [\,] \to z \\
\qquad\quad (a : as) \to \textbf{let}\ v = foldr\ f\ z\ as \\
\qquad\qquad\qquad \textbf{in}\ f\ a\ v \\
\quad \textbf{in}\ foldr'\ l
\end{array}
$$

The problem arises when the non-static part of the function, that is, the partial application of the function to its static arguments (they must be explicit so that the transformation can be applied) is not a weak head normal form. If it is, there is no problem as the new function is also a λ-abstraction so its behaviour and that of the partial application of the original function to its static arguments are exactly the same. But if it is necessary to evaluate a non-deterministic expression to get a weak head normal form, then the new function and the original may not have the same behaviour: Before the transformation, the non-deterministic expression is evaluated in each recursive call, while after the transformation it is only evaluated once.

Specialization It transforms partial applications to types and dictionaries into *let* bindings:

$$
\begin{array}{l}
g = \Lambda ty.\lambda dict.\lambda y. \to \\
\quad \textbf{let}\ f = \Lambda ty.\lambda dict. \to e \\
\quad \textbf{in}\ f\ ty\ dict\ (f\ ty\ dict\ y)
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
g = \Lambda ty.\lambda dict.\lambda y. \to \\
\quad \textbf{let}\ f = \Lambda ty.\lambda dict. \to e \\
\quad \textbf{in} \\
\qquad \textbf{let}\ f' = f\ ty\ dict \\
\qquad \textbf{in}\ f'\ (f'\ y)
\end{array}
$$

Lets assume in this example that e is an expression that generates non-deterministically a function. This means that the two partial applications $f\ ty\ dict$ appearing in g's body denote potentially different functions. Once the transformation has been applied, the first time f' is evaluated, it will be updated with its weak head normal form, so the two ocurrences of f' do denote necessarily the same function. This means that we have lost non-determinism.

4 Language definition

As Eden is implemented by modifying GHC, the language being analysed, CoreEden, is an extension of the intermediate language Core of GHC. This is a simple functional language with second-order polymorphism, so the language includes type abstraction and type application. The type information is available in Core at the binders positions (that is, bound variables in *let* expressions, variables in the left hand side of a *case* alternative, and in the arguments of a function). In Figure 1 the syntax of the language and of the type expressions are shown. There, v denotes a variable, k denotes a literal and x denotes an atom (a variable or a literal).

A program (*prog*) is a list of possibly recursive bindings (*bind*) from variables to expressions. Such expressions (*expr*) may be Core expressions or any of the new Eden constructions. Core expressions include variables, lambda abstractions, applications of a functional expression to an atom, constructor applications, primitive operators

prog	$\rightarrow$	$bind_1; \ldots; bind_m$		
bind	$\rightarrow$	$v = expr$		{non-recursive binding}
		| **rec** $v_1 = expr_1; \ldots; v_m = expr_m$		{recursive binding}
expr	$\rightarrow$	$expr\ x$		{application to an atom}
		| $\lambda v.expr$		{lambda abstraction}
		| **case** *expr* **of** *alts*		{*case* expression}
		| **let** *bind* **in** *expr*		{*let* expression}
		| $C\ x_1 \ldots x_m$		{saturated constructor application}
		| $op\ x_1 \ldots x_m$		{saturated primitive operator application}
		| x		{atom}
		| $\Lambda\alpha.expr$		{type abstraction}
		| $expr\ type$		{type application}
		| $v\ \#\ x$		{process instantiation}
		| **process** $v \rightarrow expr$		{process abstraction}
alts	$\rightarrow$	$Calt_1; \ldots; Calt_m; [Deft]$	$m \geq 0$	
		| $Lalt_1; \ldots; Lalt_m; [Deft]$	$m \geq 0$	
Calt	$\rightarrow$	$C\ v_1 \ldots v_m \rightarrow expr$	$n \geq 0$	{algebraic alternative}
Lalt	$\rightarrow$	$k \rightarrow expr$		{primitive alternative}
Deft	$\rightarrow$	$v \rightarrow expr$		{default alternative}
type	$\rightarrow$	K		{basic types: Integers, characters}
		| α		{type variables}
		| $T\ type_1 \ldots type_m$		{type constructor application}
		| $type_1 \rightarrow type_2$		{function type}
		| $Process\ type_1\ type_2$		{process type}
		| $\forall\alpha.type$		{polymorphic type}

Figure 1: Language definition and type expressions

applications, and also *case* and *let* expressions. Constructor and primitive operators applications are saturated. The variables contain type information, so we will not write it explicitly in the expressions.

The new Eden expressions are a process abstraction **process** $v \rightarrow e$, and a process instantiation $v\ \#\ x$. Each process abstraction has as input a single variable, as the λ-abstractions have. In Core the applications are made over atoms (literals or variables), so in a process instantiation the process must be a variable and the input an atom.

There is also a new type $Process\ t_1\ t_2$, see Figure 1, representing the type of a process abstraction **process** $v \rightarrow e$ where v has type t_1 and e has type t_2. Frequently t_1 and t_2 are tuple types, where each tuple element represents an input or an output channel of the process. The rest of the types are the Core types: Basic types, type variables, type constructor applications, function types and polymorphic types.

The program written in CoreEden is obtained by a translation from Core. In Core, the Eden constructions are hidden: The process abstraction is hidden as a λ-abstraction and the instantiation is hidden as the application of a special instantiation function. For more details, see [PPRS00].

5 The annotations

In this section a types annotation system is presented. As type information is already available in the language, we just need to annotate types.

5.1 Introduction

The analysis attaches non-determinism annotations to types. These are basic annotations n or d, or tuples of basic annotations. A basic annotation d in the type of an expression means that such expression is sure to be deterministic. For example, an integer constant is deterministic. A basic annotation n means that the

expression may be non-deterministic. As process `merge` is the only source of non-determinism, we say that an expression may be non-deterministic if it 'contains´ an instantiation of `merge`, where this 'contains' has to be precisely explained.

Tuples of basic annotations correspond to expressions of tuple type (or processes/functions returning tuples, see below) where each component carries its own annotation. The tuple type is treated in a special way; the rest of data types just carry a basic annotation (the motivations and consequences of this are explained in Section 6.4). Processes usually have several input/output channels, and this fact is represented by using tuples. In the implementation, an independent concurrent thread is provided for every output channel of a process. We would like to express which ones are deterministic and which ones may be non-deterministic. For example, in the following process abstraction

$$
\begin{array}{l}
\textbf{process } v \rightarrow \textbf{case } v \textbf{ of} \\
\quad (v_1, v_2) \rightarrow \textbf{let} \\
\qquad\qquad y_1 = v_1 \\
\qquad\qquad y_2 = \mathit{merge}\ \#\ v_2 \\
\qquad \textbf{in} \\
\qquad\qquad (y_1, y_2)
\end{array}
$$

we say that the first output is deterministic and that the second one may be non-deterministic. The same happens to functions returning tuples. No nested tuples of annotations are generated. As the internal tuples do not represent output channels only one level of tupling is maintained, they are treated as the rest of algebraic types.

Lets see now what means to be deterministics or possibly non-determinstic. We have already said that the constants are deterministic. A function/process is deterministic when it maintains the determinism level. For example, the identity function/process is deterministic: When applied to a deterministic argument it returns a deterministic result, and when applied to a non-deterministic argument, it returns a non-deterministic result. The constant functions/processes are also deterministic, as they always return a deterministic argument, independently from the input. It seems there are several levels of determinism that perhaps we should distinguish. But at this moment we do not do it.

The consequences of this decision are not insignificant and they are explained in more detail in Section 6.4. Lets say now that this decision makes the analysis simpler and efficient. It is simpler because it allows us to consider that the determinism of a function/process depends exclusively on its body but not on the arguments it is called with. So we can assume the arguments are deterministic: A function/process will be deterministic if given deterministic arguments we obtain a deterministic result. If the result is non-deterministic for deterministic arguments then the function/process will be considered as non-deterministic. This means we forget about the dependency between the arguments and the result of the function/process, that is, we are losing information.

5.2 Some notation

In Figures 3 and 4 the types annotation system is shown. There b is used to denote a basic annotation and a to denote a basic annotation or a tuple of basic annotations:

$$
\begin{array}{lcl}
b & \rightarrow & n \\
 & & \mid d \\
a & \rightarrow & b \\
 & & \mid (b_1, \ldots, b_m)
\end{array}
$$

Regarding the types, t is used to denote the unannotated ones, see Figure 1, and τ or t^a to denote the annotated ones. So, if we write $A \vdash e :: t^a$ we are making explicit the fact that a is the annotation of t. In the type environments, $A + [v :: t^a]$ denotes the extension of environment A with the annotated typing for v. In the typing rules of Figures 3 and 4, i ranges from 1 to m and j from 1 to l_i. Overlining is used to indicate an indexed sequence. For example, $A + \overline{[v_i :: \tau_i]}$ represents the extension of A with new typings for the variables $v_1, \ldots, v_m$.

$$
\begin{array}{l}
(b_1,\ldots,b_m)_t = (b_1,\ldots,b_m) \\
b_{(t_1,\ldots,t_m)} = (\overset{1}{b},\ldots,\overset{m}{b}) \\
b_{t_1 \to t_2} = b_{t_2} \\
b_{Process\ t_1\ t_2} = b_{t_2} \\
b_{\forall \alpha.t} = b_t \\
b_t = b \ e.o.c
\end{array}
$$

Figure 2: Adaptation function definition

An ordering between the annotations is stablished, $d \sqsubseteq n$ (naturally extended to tuples), and we define a least upper bound operator (lub) $\sqcup$:

$$
\begin{array}{rcl}
n \sqcup b & = & n \\
d \sqcup b & = & b
\end{array}
$$

This operator is overloaded as we use it also to distribute a basic annotation along a tuple:

$$b \sqcup (b_1,\ldots,b_m) = (b_1 \sqcup b,\ldots,b_m \sqcup b)$$

We use $\bigsqcup_i$ or $\bigsqcup_j$ to denote several lub operations.

We need an operator $\widehat{\sqcup}$ to flatten the internal tuples so that nested tuples do not appear:

$$
\begin{array}{rcl}
\widehat{\sqcup} b & = & b \\
\widehat{\sqcup}(b_1,\ldots,b_m) & = & \bigsqcup_i b_i
\end{array}
$$

In the type rules it is necessary to adapt an annotation a to a type t in some places. This adaptation is represented as a_t and it is defined in Figure 2.

5.3 The type system

Lets see now the types annotation rules, shown in Figures 3 and 4. Rule [*VAR*] is trivial. Rule [*LIT*] specifies that constants of basic types are deterministic. There are two rules for constructors: One for tuples [*TUP*] and another one [*CONS*] for the rest. In the first case, we obtain the annotation of each component, flatten them (if they are tuples, nesting must be eliminated) and give back the resulting tuple. In the second case, we also obtain the components' annotations, flatten them and finally apply the lub operator, so that a basic annotation is obtained. This implies that, if any component of the construction may be non-deterministic, the whole expression will be considered as possibly non-deterministic; the information about the components is lost.

We have already said that we are only interested in the result of the functions when they are applied to deterministic arguments. So, in the rule [*ABS*] the annotation attached to the function is the one obtained for the body when in the environment the argument is assigned a deterministic annotation. If the body gets a deterministic annotation, the function is deterministic; but if the body may be non-deterministic then the function may be non-deterministic. The deterministic annotation given to the argument is an adaptation of the basic annotation d to the type of the argument, see Figure 2. For example, if it is a n-tuple, the annotation should be an n-tuple $(d,\ldots,d)$. If the argument were non-deterministic we are always assuming that the result may be non-deterministic. This means that we are not expressing how the output depends on the input. In Section 6.4 we will see that this leads to some limitations of the analysis. The lack of such information is reflected in the [*APPLY*] rule. The result of the application may be non-deterministic either the function is annotated as non-deterministic or the argument is annotated as non-deterministic. This is expressed by using a lub operator. If the argument's annotation is a tuple, then we have to previously flatten it as we cannot use the information that its components provide. Such information (independent annotations) is used when

$$\frac{}{A + [v :: \tau] \vdash v :: \tau}\ VAR \qquad \frac{}{A \vdash k :: K^d}\ LIT$$

$$\textbf{data}\ \ T\ \overline{\alpha_k} = \overline{C_i\ \overline{t_{ij}}} \qquad op :: t_1 \to (t_2 \to \ldots (t_m \to t))$$

$$\frac{A \vdash x_j :: [t_{ij}[\overline{\alpha_k := t_k}]]^{a_j}}{A \vdash C_i\ \overline{x_j} :: (T\ \overline{t_k})^{\bigsqcup_j (\widehat{\sqcup} a_j)}}\ CONS \qquad \frac{A \vdash x_i :: {t_i}^{a_i}}{A \vdash op\ x_1 \ldots x_m :: t^{(\bigsqcup_i (\widehat{\sqcup} a_i))_t}}\ PRIM$$

$$\frac{A \vdash x_i :: {t_i}^{a_i}}{A \vdash (x_1, \ldots, x_m) :: (t_1, \ldots, t_m)^{(\widehat{\sqcup} a_1, \ldots, \widehat{\sqcup} a_m)}}\ TUP \qquad \frac{A \vdash e :: t_1 \xrightarrow{a_1} t_2 \quad A \vdash x :: {t_1}^{a_2}}{A \vdash (e\ x) :: {t_2}^{(\widehat{\sqcup} a_2) \sqcup a_1}}\ APPLY$$

$$\frac{A + [v :: t^{d_t}] \vdash e :: {t'}^a}{A \vdash (\lambda v.e) :: t \xrightarrow{a} t'}\ ABS \qquad \frac{A + [v :: t^{d_t}] \vdash e :: {t'}^a}{A \vdash \textbf{process}\ v \to e :: Process^a\ t\ t'}\ PABS$$

Figure 3: Types annotation rules (I)

the components are separatedly used in different parts of the program, and this is what usually happens with processes: Each output channel feeds a different process. Rules [*PABS*] and [*PINST*] are identical to [*ABS*] and [*APPLY*].

In [*PRIM*] rule, primitive operators are considered as deterministic, so we just flatten the annotations of the arguments and apply a lub operator to them. Finally, the annotation is adapted to the type of the result.

The [*MERGE*] rule specifies that `merge` may be a non-deterministic process (in fact, it is the source of non-determinism). The [*LETNONREC*] and [*LETREC*] rules are the expected ones: The binders are added to the environment with the annotations of the right hand sides of their bindings.

An algebraic *case* expression may be non-deterministic if either the discriminant expression (the choice between the alternatives could be non-deterministic) or any of the expressions in the alternatives may be non-deterministic. This is expressed in the [*CASEALG*] rule. However if the discriminant is a tuple, there is no non-deterministic choice between the alternatives. This information is just passed to the right hand side of the alternative, so that only if the non-deterministic variables are used there, the result will be annotated as non-deterministic. This is reflected in the [*CASETUP*] rule. In general, the same applies to those types with only one constructor, so it could be extended to all such types. In these two *case* rules, the annotation obtained from the discriminant has to be adapted to the types of the variables in the left hand side of the alternatives. In the [*CASEALG*] rule the discriminant annotation is just a basic annotation that represents the whole structure. If it is deterministic, then we can say that each of the components of the value is deterministic; and in case it is non-deterministic, we have to say that each component is non-deterministic, as we have lost information when annotating the discriminant. In the [*CASETUP*] rule each component has its own annotation, so we don't lose so much information, but, as there are no nested tuples, we still have to adapt each annotation to the component's type. The optional default alternative has not been included in the figure for clarity but it is easy to do. For example the [*CASEALG*] rule would be:

$$\frac{\begin{array}{c}\textbf{data}\ T\ \overline{\alpha_k} = \overline{C_i\ \overline{t_{ij}}} \\ A \vdash e :: (T\ \overline{t_k})^b \\ A + A_i \vdash e_i :: t^{a_i}\ \text{where}\ A_i = \overline{[v_{ij} :: {tv_{ij}}^{b_{tv_{ij}}}]},\ tv_{ij} = t_{ij}\ \overline{[\alpha_k := t_k]} \\ A + [v :: (T\ \overline{t_k})^b] \vdash e_v :: t^{a_v}\end{array}}{A \vdash \textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}; v \to e_v :: t^{\,b \sqcup (\bigsqcup_i a_i) \sqcup a_v}}\ CASEALG$$

We have type polymorphism but not annotation polymorphism. In [*TYABS*] rule A, α means that α is a type variable not free in A. When the instantiation of a polymorphic type takes place, see rule [*TYAPP*], it is necessary to adapt the annotation of the polymorphic type to the instantiated type. This is necessary when new structure arises from the instantiation. In the following section some examples are shown.

5.4 Some examples

Lets see how the types annotation system works for some examples.

The identity process The identity process is defined as follows:

$$id = \Lambda\alpha.\mathbf{process}\ v \to v$$

We have already said it is a deterministic process:

$$\dfrac{\dfrac{\dfrac{}{A + [v :: \alpha^d] \vdash v :: \alpha^d}\ VAR}{A \vdash \mathbf{process}\ v \to v :: Process^d\ \alpha\ \alpha}\ PABS}{A \vdash \Lambda\alpha.\mathbf{process}\ v \to v :: (\forall\alpha.Process\ \alpha\ \alpha)^d}\ TYABS$$

An application of the identity process This example illustrates the need for adapting an annotation when a polymorphic type is instatiated. Lets apply the identity process to a tuple of 5:

$$(id\ (Int, Int))\ \#\ (5, 5)$$

We need to adapt d to the instantiated type $Process\ (Int, Int)\ (Int, Int)$, which produces (d, d), see Figure 2. In Figure 5 the place where adaptation is made appears in bold face. The dots represent the previous derivation for the identity process.

If the external structure was already a tuple the annotation is maintained. For example, in $\Lambda\alpha.process\ x \to (x, x)$, with annotated type $(\forall\alpha.Process\ \alpha\ (\alpha, \alpha))^{(d,d)}$, the adaptation gives back the same annotation (d, d).

A *case* expression In a *case* expression it is also necessary to adapt the discriminant's annotation to the types of the components, in case these are tuples. This is reflected in the following example.

$$\begin{array}{l}\Lambda\alpha.\Lambda\beta.\Lambda\gamma.\mathbf{process}\ v \to \mathbf{case}\ v\ \mathbf{of} \\ \qquad (v_1, v_2) \to \mathbf{case}\ v_1\ \mathbf{of} \\ \qquad\qquad (v_{11}, v_{12}) \to ((v_2, v_{11}), v_{12})\end{array}$$

Lets call p this process. In Figure 6 each of the outputs of p is shown to be deterministic. The place where the adaptation of the annotation is made is shown in bold face: When [*CASETUP*] rule is applied, the annotation d is adapted to the type of v_1, that is a tuple, so it is added to the type environment with (d, d) as annotation.

6 Abstract interpretation

The analysis of the previous section has several limitations, explained in Section 6.4. In this section an abstract interpretation version of the analysis is presented. This version will lead us to develop a more powerful analysis, also abstract interpretation based, in which we will be able to overcome these limitations. Such extension does not seem so evident in the types annotation system.

The type system is directly related to an abstract interpretation where the domains corresponding to functions/processes are identified with their range domains. This means there are no functional domains, so the fixpoint calculation is less expensive. This abstract interpretation leads directly to an algorithm we have implemented in Haskell, and we have executed with the examples of Section 5.4, and some more, one of which is shown below. This algorithm uses syntax-driven recursive calls that accumulate variables in the environment as necessary. This is equivalent to a bottom-to-top pass in the type annotation rules where only the type environments are built. When recursive calls finish, lub operations are carried out. This is equivalent to the application of the types annotation rules from top to bottom, once we have the appropriate environment.

6.1 The abstract domains

Figure 7 shows the abstract domains. There is a basic domain *Basic* that corresponds to the annotations d and n in the previous section, with the same ordering. This is the abstract domain corresponding to basic types and algebraic types (except tuples). Tuples are again specially treated, as tuples of basic abstract values. The abstract domain corresponding to functions and processes is the abstract domain corresponding to the type of the result. The abstract domain of a polymorphic type is that of its smallest instance, i.e. that one in which K is substituted for the type variable. So the domain corresponding to a type variable is *Basic*.

$$\frac{A + [v :: \tau_1] \vdash e :: \tau_2 \quad A \vdash e_1 :: \tau_1}{A \vdash \textbf{let } v = e_1 \textbf{ in } e :: \tau_2} \; \mathit{LETNONREC}$$

$$\frac{A + \overline{[v_i :: \tau_i]} \vdash e_i :: \tau_i \quad A + \overline{[v_i :: \tau_i]} \vdash e :: \tau'}{A \vdash \textbf{let rec } \overline{v_i = e_i} \textbf{ in } e :: \tau'} \; \mathit{LETREC}$$

$$\frac{A \vdash v :: \mathit{Process}^a \; t \; t' \quad A \vdash x :: t^{a'}}{A \vdash v\#x :: t^{b \sqcup a} \text{ where } b = \widehat{\sqcup} a'} \; \mathit{PINST} \qquad \frac{}{A \vdash \mathit{merge} :: \mathit{Process}^n \; [[\alpha]] \; [\alpha]} \; \mathit{MERGE}$$

$$\frac{A \vdash e :: (t_1, \ldots, t_m)^{(b_1, \ldots, b_m)} \quad A + \overline{[v_i :: t_i^{\,b_i \iota_i}]} \vdash e' :: \tau'}{A \vdash \textbf{case } e \textbf{ of } (v_1, \ldots, v_m) \to e' :: \tau'} \; \mathit{CASETUP}$$

$$\textbf{data } T \; \overline{\alpha_k} = \overline{C_i \; \overline{t_{ij}}}$$

$$A \vdash e :: (T \; \overline{t_k})^b$$

$$\frac{A + A_i \vdash e_i :: t^{a_i} \text{ where } A_i = \overline{[v_{ij} :: tv_{ij}^{\,b_{\iota v_{ij}}}]}, tv_{ij} = t_{ij} \; \overline{[\alpha_k := t_k]}}{A \vdash \textbf{case } e \textbf{ of } \overline{C_i \; \overline{v_{ij}} \to e_i} :: t^{\,b \sqcup (\bigsqcup_i a_i)}} \; \mathit{CASEALG}$$

$$\frac{A \vdash e :: K^b \quad A \vdash e_i :: t^{a_i}}{A \vdash \textbf{case } e \textbf{ of } \overline{k_i \to e_i} :: t^{\,b \sqcup (\bigsqcup_i a_i)}} \; \mathit{CASEPRIM}$$

$$\frac{A, \alpha \vdash e :: t^a}{A \vdash \Lambda\alpha.e :: (\forall\alpha.t)^a} \; \mathit{TYABS} \qquad \frac{A \vdash e :: (\forall\alpha.t)^a}{A \vdash (e \; t') :: \mathit{tinst}^{a_{\mathit{tinst}}} \text{ where } \mathit{tinst} = t[\alpha := t']} \; \mathit{TYAPP}$$

Figure 4: Types annotation rules (II)

$$\dfrac{\dfrac{\dfrac{}{A \vdash 5 :: Int^{d}}\ LIT \quad \dfrac{}{A \vdash 5 :: Int^{d}}\ LIT}{A \vdash (5,5) :: (Int, Int)^{(d,d)}}\ TUP \quad \dfrac{\vdots}{A \vdash id :: \forall\alpha.Process^{d}\ \alpha\ \alpha}}{A \vdash id\ (Int, Int) :: Process^{(\mathbf{d},\mathbf{d})}\ (Int, Int)\ (Int, Int)}\ \mathbf{TYAPP}}{A \vdash (id\ (Int, Int))\ \#\ (5,5) :: (Int, Int)^{(d,d)}}\ PINST$$

Figure 5: An application of the identity process

$$\dfrac{\dfrac{\dfrac{}{A_2 \vdash v_1 :: (\alpha,\beta)^{(d,d)}}\ VAR \quad \dfrac{\dfrac{\dfrac{}{A_3 \vdash v_2 :: \gamma^{d}}\ VAR \quad \dfrac{}{A_3 \vdash v_{11} :: \alpha^{d}}\ VAR}{A_3 \vdash (v_2, v_{11}) :: (\gamma,\alpha)^{(d,d)}}\ TUP \quad \dfrac{}{A_3 \vdash v_{12} :: \beta^{d}}\ VAR}{A_3 = A_2 + [v_{11} :: \alpha^{d}, v_{12} :: \beta^{d}] \vdash ((v_2, v_{11}), v_{12}) :: ((\gamma,\alpha),\beta)^{(d,d)}}\ TUP}{A_2 = A_1 + [v_1 :: (\alpha,\beta)^{(\mathbf{d},\mathbf{d})}, v_2 :: \gamma^{d}] \vdash \mathbf{case}\ v_1\ \mathbf{of}\ (v_{11}, v_{12}) \to ((v_2, v_{11}), v_{12}) :: ((\gamma,\alpha),\beta)^{(d,d)}}\ CASETUP \quad \dfrac{}{A_1 \vdash v :: ((\alpha,\beta),\gamma)^{(d,d)}}\ VAR}{A_1 = A + [v :: ((\alpha,\beta),\gamma)^{(d,d)}] \vdash \mathbf{case}\ v\ \mathbf{of}\ (v_1, v_2) \to \mathbf{case}\ v_1\ \mathbf{of}\ (v_{11}, v_{12}) \to ((v_2, v_{11}), v_{12}) :: ((\gamma,\alpha),\beta)^{(d,d)}}\ \mathbf{CASETUP}}{A \vdash \mathbf{process}\ v \to \mathbf{case}\ v\ \mathbf{of}\ (v_1, v_2) \to \mathbf{case}\ v_1\ \mathbf{of}\ (v_{11}, v_{12}) \to ((v_2, v_{11}), v_{12}) :: Process^{(d,d)}\ ((\alpha,\beta),\gamma)\ ((\gamma,\alpha),\beta)}\ PABS}{A \vdash p :: \forall\alpha.\forall\beta.\forall\gamma.Process^{(d,d)}\ ((\alpha,\beta),\gamma)\ ((\gamma,\alpha),\beta)}\ TYABS*$$

Figure 6: Annotation of the *case* expression

$$
\begin{array}{l}
Basic = \{d, n\} \text{ where } d \sqsubseteq n \\
D_K = Basic \\
D_{(t_1,\ldots,t_m)} = \{(b_1, \ldots, b_m) \mid b_i \in \{d, n\}\} \\
D_{T\ t_1 \ldots t_m} = Basic \\
D_{t_1 \to t_2} = D_{t_2} \\
D_{Process\ t_1\ t_2} = D_{t_2} \\
D_\alpha = Basic \\
D_{\forall \alpha.t} = D_t
\end{array}
$$

Figure 7: Abstract domains

6.2 A simple abstract interpretation

In Figure 8 the abstract interpretation is shown. It is very similar to the type annotation system, so we just outline some specific details.

In the recursive *let* we have to calculate a fixpoint, which can be obtained by using the Kleene's ascending chain:

$$[\![\textbf{let rec } \overline{\{v_i = e_i\}} \textbf{ in } e']\!]\ \rho = [\![e']\!]\ (\bigsqcup_{n \in \mathbb{N}} (\lambda \rho'.\rho\ \overline{[v_i \to [\![e_i]\!]\ \rho']})^n (\rho_0))$$

where ρ_0 is an environment in which all variables have as abstract value the infimum $\bot_t$ of its corresponding abstract domain. :

$$
\begin{array}{l}
\bot_K = \bot_{T\ t_1 \ldots t_n} = \bot_\alpha = d \\
\bot_{(t_1,\ldots,t_n)} = (d, \ldots, d) \\
\bot_{t_1 \to t_2} = \bot_{Process\ t_1\ t_2} = \bot_{t_2} \\
\bot_{\forall \beta.t'} = \bot_{t'}
\end{array}
$$

Notice that for each type t, $d_t = \bot_t$ (this can be proved easily by structural induction over t). At each iteration, the abstract values of the bindings' right hand sides are computed and the environment is updated until no changes are found. Termination is assured, as the abstract domains corresponding to each type are finite. The number of iterations are $O(N)$, being N the total number of 'components' in the bindings: One for each non-tuple variable and one for each component of a tuple variable.

The interpretation of the algebraic and primitive *case* can be expressed in a more intuitive way as:

$$
[\![\textbf{case } e \textbf{ of } \overline{C_i\ \overline{v_{ij}} \to e_i}]\!]\ \rho = \begin{cases} n_t \text{ if } [\![e]\!]\ \rho = n \\ \bigsqcup\limits_i [\![e_i]\!]\ \rho_i \text{ otherwise} \end{cases}
$$

$$
\text{where } \rho_i = \rho\ \overline{[v_{ij} \to d_{t_{ij}}]}, v_{ij} :: t_{ij}, e_i :: t
$$

$$
[\![\textbf{case } e \textbf{ of } \overline{k_i \to e_i}]\!]\ \rho = \begin{cases} n_t \text{ if } [\![e]\!]\ \rho = n \\ \bigsqcup\limits_i [\![e_i]\!]\ \rho \text{ otherwise} \end{cases}
$$

$$
\text{where } e_i :: t
$$

In the *case* expression where the discriminant is of tuple type, π_i represents the ith projection.

6.3 An example: Replicated workers

This example shows a simplified version of a replicated workers topology [KPR00]. We have a manager process and n worker processes. The manager provides the workers with tasks. When any of the workers finishes its task, it sends a message to the manager including the obtained results and asking for a new task. In order the manager can receive the answers from the workers in any order, and inmediately assign new tasks to idle processes, a `merge` process is needed.

The function `rw` representing this scheme when $n = 2$ is shown in Figure 9, where `worker` is the worker process and `ts` is an initial list of tasks to be done by the workers. The output of the manager process `manager` usually depends on both input lists, the initial one `ts` and that produced by the workers `os`. However, in order

$$
\begin{array}{l}
[\![v]\!]\ \rho = \rho\ v \\
[\![k]\!]\ \rho = d \\
[\![(x_1, \ldots, x_m)]\!]\ \rho = (\widehat{\sqcup}([\![x_1]\!]\ \rho), \ldots, \widehat{\sqcup}([\![x_m]\!]\ \rho)) \\
[\![C\ x_1 \ldots x_m]\!]\ \rho = \bigsqcup_i \widehat{\sqcup}([\![x_i]\!]\ \rho) \\
[\![e\ x]\!]\ \rho = (\widehat{\sqcup}([\![x]\!]\ \rho)) \sqcup [\![e]\!]\ \rho \\
[\![op\ x_1 \ldots x_m]\!]\ \rho = (\bigsqcup_i \widehat{\sqcup}([\![x_i]\!]\ \rho))_t \text{ where } op :: t_1 \to (t_2 \to \ldots (t_m \to t)) \\
[\![p\#x]\!]\ \rho = \widehat{\sqcup}([\![x]\!]\ \rho) \sqcup [\![p]\!]\ \rho \\
[\![\lambda v.e]\!]\ \rho = [\![e]\!]\ \rho\ [v \to d_t] \text{ where } v :: t \\
[\![\textbf{process}\ v \to e]\!]\ \rho = [\![e]\!]\ \rho\ [v \to d_t] \text{ where } v :: t \\
[\![merge]\!]\ \rho = n \\
[\![\textbf{let}\ v = e\ \textbf{in}\ e']\!]\ \rho = [\![e']\!]\ \rho\ [v \to [\![e]\!]\ \rho] \\
[\![\textbf{let rec}\ \overline{\{v_i = e_i\}}\ \textbf{in}\ e']\!]\ \rho = [\![e']\!]\ (fix\ (\lambda\rho'.\rho\ \overline{[v_i \to [\![e_i]\!]\ \rho']})) \\
[\![\textbf{case}\ e\ \textbf{of}\ (v_1, \ldots, v_m) \to e']\!]\ \rho = [\![e']\!]\ \rho\ \overline{[v_i \to \pi_i([\![e]\!]\ \rho)_{t_i}]} \text{ where } v_i :: t_i \\
[\![\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}]\!]\ \rho = b \sqcup (\bigsqcup_i [\![e_i]\!]\ \rho_i) \\
\qquad\qquad \text{where } b = [\![e]\!]\ \rho \\
\qquad\qquad\qquad \rho_i = \rho\ \overline{[v_{ij} \to b_{t_{ij}}]}, v_{ij} :: t_{ij} \\
[\![\textbf{case}\ e\ \textbf{of}\ \overline{k_i \to e_i}]\!]\ \rho = [\![e]\!]\ \rho \sqcup (\bigsqcup_i [\![e_i]\!]\ \rho) \\
[\![\Lambda\alpha.e]\!]\ \rho = [\![e]\!]\ \rho \\
[\![e\ t]\!]\ \rho = ([\![e]\!]\ \rho)_{tinst} \text{ where } (e\ t) :: tinst
\end{array}
$$

Figure 8: Abstract interpretation

to compare the power of this analysis with that one presented in Section 7 we are assuming that `manager` is defined as shown in Figure 9, where `g`, `h` and `r` are deterministic functions (i.e they have d as abstract value). So, `manager`'s abstract value is (d, d, d). With this definition, the third component of the process output only depends on the first input list. This means that the final result of the function `rw` only depends on the initial tasks list. In Figure 9 the process topology with the annotations in each channel is shown. As there are mutually recursive definitions, all of them get the n annotation. However, we know that that when `ts` is deterministic, the result of the function is also deterministic, as it only depends on that initial list, and not on the one coming from the `merge` process. The analysis answer is safe but just approximate. In Section 7 a more powerful analysis is presented. There the results are more accurate.

6.4 Limitations

As we have previously said, there are no functional abstract domains in this analysis. This means that the fixpoint calculation is not expensive, but it also imposes some limitations to the analysis. For instance, we cannot express the dependency of the result with respect to the argument. As we have said before, in a function application or process instantiation, we cannot fully use the information provided by the argument.

This happens, for example, when the function does not depend on any of its arguments. For example, if we define the function $f\ v = 5$, the analysis tells us that the function is deterministic, but when we apply f to a possibly non-deterministic value, the result of the application is stablished as possibly non-deterministic. This is not true, as we will always obtain the same unique value. Of course this a safe approximation, but not a very accurate one. The function $g\ v = v$ would have the same abstract behaviour. Both f and g are deterministic functions, but they have different levels of determinism: f does not depend on its argument, but g does. If functions and processes were interpreted as functions, this limitation would be over. The abstract function $f^\#$ corresponding to f would be $f^\# = \lambda z.d$, while that of g, $g^\#$, would be $\lambda z.z$. Now, if we applied $f^\#$ to n, we would obtain d, but n in the case of $g^\#$. The same happens when tuples are involved. The abstract value of

$$h\ v_1\ v_2\ v_3 = (v_1, v_2, merge\ \#\ v_3)$$

is (d, d, n). But when we apply it, if any of the arguments has n as abstract value, the result will be (n, n, n).

```
rw = λ worker.λ ts.
       let rec
        t = (ts, is)
        ys = manager # t
        y1 = case ys of (z1,z2,z3) → z1
        y2 = case ys of (u1,u2,u3) → u2
        y3 = case ys of (v1,v2,v3) → v3
        o1 = worker # y1
        o2 = worker # y2
        os = [o1, o2]
        is = merge # os
       in y3

manager = process ts → case ts of
            (t1, t2) → (g t1 t2,h t1 t2,r t1)
```

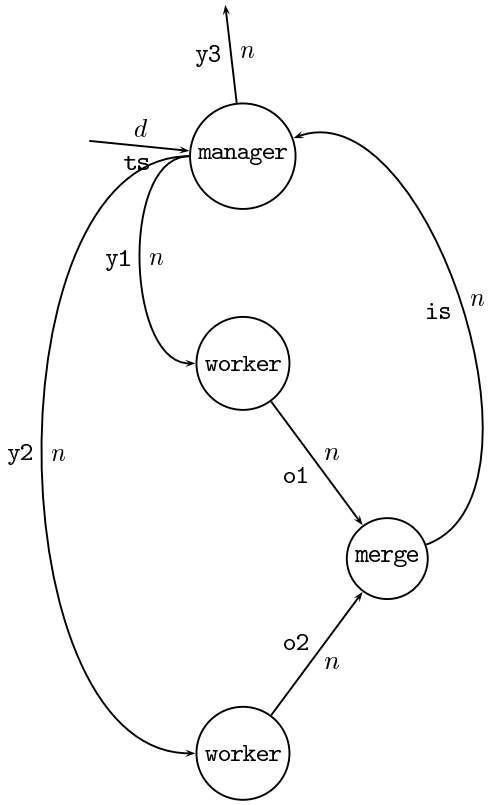


Figure 9: Replicated workers process structure where $n = 2$.

If the abstract value were

$$h^{\#}\ v_1{}^{\#}\ v_2{}^{\#}\ v_3{}^{\#} = (v_1{}^{\#}, v_2{}^{\#}, n)$$

then we would not lose so much information, for example $h^{\#}\ d\ n\ d = (d, n, n)$.

Although this analysis will be more precise, it will be also more expensive, as we need to represent functions and also to calculate the fixpoint, which implies it is necessary to compare functions. So it is exponential. It has all the problems of the typical strictness analysis [BHA86, MH87, Hun91]. In the simple analysis of the previous sections, the fixpoint calculation is linear in the number of components in the *let* bindings.

With respect to the algebraic types we have represented the whole structure with a single basic abstract value (except in the tuples). This means for example, that if any element of a list is non-deterministic, then the whole list is considered as non-deterministic. Is much information lost in this way? Lets concentrate in the integer lists *List Int*. In the type system we could annotate the type of the elements to indicate if these have been generated in a deterministic or non-deterministic way, and also the external list to indicate if the list itself has been generated in a deterministic way or not.

For example, $List^n\ Int^d$ would represent a list of deterministic integers that has been generated in a non-deterministic way. This would be the case of $merge\#[[0, 0..], [1, 1..]]$. But in this case when an element is taken from the list, for example the head of the list, we have to forget about the way in which the elements have been generated as we do not know which value appears in that position. So it does not seem very useful to know how the elements have been generated when the list itself is non-deterministic.

$List^d\ Int^n$ would represent a list whose elements are non-deterministic that has been generated in a deterministic way (for example just by applying the list constructor). This kind of lists would admit some deterministic queries; for example, the function *length* applied to it should return d as result.

Finally we have $List^d\ Int^d$ and $List^n\ Int^n$. These are in fact the cases we are considering in the analysis. In the analysis we are identifying all the cases in which an n appears in any position with $List^n\ Int^n$. It seems we are not losing so much that it is worthwhile to introduce such level of detail, but perhaps a more detailed study would be useful.

$Basic = \{d, n\}$ where $d \sqsubseteq n$
$D_{2K} = Basic$
$D_{2T\ t_1 \ldots t_m} = Basic$
$D_{2(t_1,\ldots,t_m)} = D_{2t_1} \times \ldots \times D_{2t_m}$
$D_{2t_1 \to t_2} = [D_{2t_1} \to D_{2t_2}]$
$D_{2Process\ t_1\ t_2} = [D_{2t_1} \to D_{2t_2}]$
$D_{2\beta} = Basic$
$D_{2\forall\beta.t} = D_{2t}$

Figure 10: Abstract domains for the refined analysis

7 A refinement of the analysis

To overcome the limitations of the first analysis, we define a new abstract interpretation based analysis, where the dependency between the argument and the result in a function can be expressed by interpreting the functions and processes as abstract functions. As we have previously said, this allows us to express several levels of determinism and non-determinism.

7.1 The abstract domains

In Figure 10 the abstract domains for the new analysis are shown. An index 2 is used to distinguish them from the previous analysis, that will be indexed with a 1.

As in the previous analysis, the abstract domain corresponding to the basic types is the basic domain *Basic*. For the algebraic data types (except for tuples) we maintain *Basic* as abstract domain, following the same previous ideas. The abstract domain of a tuple type is the cartesian product of the domains corresponding to the types of the components. So now we can have nested tuples. In this way we can maintain more information than in the previous analysis; as now functions are interpreted as abstract functions, we would like that those appearing inside a tuple (for example, if a process produces a function as result) are not lost because of a flattening. The domains corresponding to functions and processes are the domains of continuous functions between the domains of the argument and the result.

7.2 Abstract interpretation

In Figure 11 the abstract interpretation is shown. The interpretation of a variable is the abstract value assigned in the environment ρ_2. Literals are deterministic. The interpretation of a tuple is now a tuple of the abstract values of the components. The interpretation of a constructor belongs to *Basic*, so we have to take the lub of the abstract values of the components. But now, each $x_i :: t_i$ has an abstract value belonging to D_{2t_i}, so we can not apply directly the lub. First we have to flatten the information of each component. The function responsible for doing this is called the *abstraction function* α_t and it is defined in the following section. For each type t, α_t transforms an abstract value belonging to D_{2t} in an abstract value belonging to *Basic*. This is done not only in a safe way but also in a careful way (if α_t takes any value in D_{2t} to n, it would also be a safe function).

Functions and process abstractions are interpreted as abstract functions taking an abstract value z and returning the abstract value of the body of the function in an extension of the original environment where the argument v is bound to the abstract value z. So the abstract interpretation of an application $e\ x$ is the application of the abstract funtion (abstract interpretation of e) to the abstract value of the argument x. The interpretation of a process instantiation $v\#x$ is similar.

The interpretation of a non-recursive *let* expression is the interpretation of the main expression in an extension of the original environment where the variable in the binding has as abstract value the interpretation of the expression on the right hand side of the binding.

In a recursive *let* expression the fixpoint can be calculated using Kleene's ascending chain:

$$[\![\textbf{let rec } \overline{\{v_i = e_i\}} \textbf{ in } e']\!]_2\ \rho_2 = [\![e']\!]_2\ (\bigsqcup_{n\in\mathbb{N}}(\lambda\rho_2'.\rho_2\ \overline{[v_i \to [\![e_i]\!]_2\ \rho_2']})^n(\rho_{02}))$$

$$
\begin{array}{l}
[\![v]\!]_2\ \rho_2 = \rho_2(v) \\
[\![k]\!]_2\ \rho_2 = d \\
[\![(x_1,\ldots,x_m)]\!]_2\ \rho_2 = ([\![x_1]\!]_2\ \rho_2,\ldots,[\![x_m]\!]_2\ \rho_2) \\
[\![C\ x_1 \ldots x_m]\!]_2\ \rho_2 = \bigsqcup_i \alpha_{t_i}([\![x_i]\!]_2\ \rho_2) \text{ where } x_i :: t_i \\
[\![e\ x]\!]_2\ \rho_2 = ([\![e]\!]_2\ \rho_2)\ ([\![x]\!]_2\ \rho_2) \\
[\![op\ x_1 \ldots x_m]\!]_2\ \rho_2 = (\gamma_{t_{op}}(d))\ ([\![x_1]\!]_2\ \rho_2) \ldots ([\![x_m]\!]_2\ \rho_2) \text{ where } op :: t_{op} \\
[\![v\#x]\!]_2\ \rho_2 = ([\![v]\!]_2\ \rho_2)\ ([\![x]\!]_2\ \rho_2) \\
[\![\lambda v.e]\!]_2\ \rho_2 = \lambda z.[\![e]\!]_2\ \rho_2\ [v \to z] \\
[\![\textbf{process}\ v \to e]\!]_2\ \rho_2 = \lambda z.[\![e]\!]_2\ \rho_2\ [v \to z] \\
[\![merge]\!]_2\ \rho_2 = \lambda z.n \\
[\![\textbf{let}\ v = e\ \textbf{in}\ e']\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ [v \to [\![e]\!]_2\ \rho_2] \\
[\![\textbf{let rec}\ \overline{\{v_i = e_i\}}\ \textbf{in}\ e']\!]_2\ \rho_2 = [\![e']\!]_2\ (fix\ (\lambda\rho_2'.\rho_2\ \overline{[v_i \to [\![e_i]\!]_2\ \rho_2']})) \\
[\![\textbf{case}\ e\ \textbf{of}\ (v_1,\ldots,v_m) \to e']\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ \overline{[v_i \to \pi_i([\![e]\!]_2\ \rho_2)]} \\
[\![\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}]\!]_2\ \rho_2 = \begin{cases} \gamma_t(n) \text{ if } [\![e]\!]_2\ \rho_2 = n \\ \bigsqcup_i [\![e_i]\!]_2\ \rho_{2i} \text{ otherwise} \end{cases} \\
\qquad \text{where } \rho_{2i} = \rho_2\ \overline{[v_{ij} \to \gamma_{t_{ij}}(d)]}, v_{ij} :: t_{ij}, e_i :: t \\
[\![\textbf{case}\ e\ \textbf{of}\ \overline{k_i \to e_i}]\!]_2\ \rho_2 = \begin{cases} \gamma_t(n) \text{ if } [\![e]\!]_2\ \rho_2 = n \\ \bigsqcup_i [\![e_i]\!]_2\ \rho_2 \text{ otherwise} \end{cases} \\
\qquad \text{where } e_i :: t
\end{array}
$$

Figure 11: Abstract interpretation of the refined analysis

where ρ_{02} is the initial environment where each variable $v :: t$ has $\bot_{2t}$ as abstract value, that is the infimum of D_{2t}. Such infimum can be written in the following way:

$$
\begin{array}{l}
\bot_{2K} = \bot_{2T\ t_1 \ldots t_m} = d \\
\bot_{2(t_1,\ldots,t_m)} = (\bot_{2t_1},\ldots,\bot_{2t_m}) \\
\bot_{2t_1 \to t_2} = \bot_{2\,Process\ t_1\ t_2} = \lambda z.\bot_{2t_2}
\end{array}
$$

We have three different kinds of *case* expressions. If the discriminant e is of tuple type, its abstract value is a tuple of abstract values. The abstract value of the *case* expression is the abstract value of the alternative's right hand side e' in an extension of the original environment where each variable on the left hand side has as abstract value the corresponding component of e's abstract value.

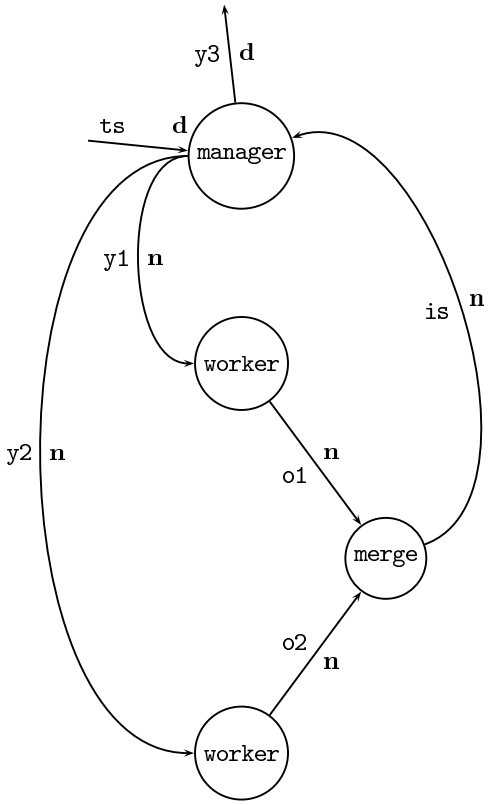
If e is of another algebraic type, things are more complex. Recall that e's abstract value, let us call it b, belongs to *Basic*. That is, when it was interpreted we lost the information about the components. We want now to interpret each alternative's right hand side in an extended environment with abstract values for the variables in the left hand side of the alternative. But we do not have such information, but a basic abstract value that represents all of them. We have to find a safe representative of b in each domain $D_{2t_{ij}}$.

This is obtained with a function γ_t, which we will call a *concretisation function*. Given a value in *Basic*, it returns a value in D_{2t}, that represents it. In particular n is represented by the top of the domain D_{2t}, and d, by a value in D_{2t} ($\gamma_t(d)$) that reflects our original idea of determinism: the maintaining of determinism. All the values below it will have different levels of determinism. So, we will see that $\gamma_t(d)$ is the biggest value in D_{2t} that has the property of maintaining the determinism. As an example, in $Int \to Int$, $\gamma_t(d) = \lambda z.z$, while $\lambda z.d$ is below it and it also has the property. This function and α_t have several properties we will study forwards.

To conclude, if the discriminant has as abstract value n, then the whole *case* expression is non-deterministic and consequently it has as abstract value $\gamma_t(n)$ (it is like the adaptation of n to the type t). If it has d as abstract value, the right hand sides of the alternatives are interpreted in extended environments where each v_{ij} has $\gamma_{t_{ij}}(d)$ as abstract value, and the lub of the resulting abstract values is calculated.

The primitive *case* expression is similar but then it is not necessary to extend the environments.

We have again considered that all the primitive operators are deterministic. So we choose $\gamma_{t_{op}}(d)$ as abstract value of an operator $op :: t_{op}$. Another, more accurate option, would be to provide an initial environment with



$g^\# = h^\# = \lambda t.\lambda s.t \sqcup s$
$r^\# = \lambda t.t$
$manager^\# = \lambda t.(\pi_1(t) \sqcup \pi_2(t),$
$\pi_1(t) \sqcup \pi_2(t), \pi_1(t))$
$rw^\# = \lambda w.\lambda l.l$
$ys^\# = (n, n, l)$

Figure 12: Results of the refined analysis for the example of Figure 9

the abstract values of all the primitive operators.

Replicated workers topology Using this analysis in the example of Figure 9 we obtain more accurate information than in the first analysis. We have assumed that `g`, `h` and `r` are deterministic functions. But now we have to provide abstract functions as their abstract values. We assume that their abstract values are: $g^\# = h^\# = \lambda t.\lambda s.t \sqcup s$ and $r^\# = \lambda t.t$. These are $\gamma_{Int\to Int\to Int}(d)$ and $\gamma_{Int\to Int}(d)$, that is, the 'biggest' deterministic functions of the corresponding types. Then, the abstract value of `manager` is $manager^\# = \lambda t.(\pi_1(t) \sqcup \pi_2(t), \pi_1(t) \sqcup \pi_2(t), \pi_1(t))$. This means that `ys` has as abstract value $ys^\# = (n, n, l)$ where l is the abstract value of the argument `ts`. So the abstract value of `rw` is $rw^\# = \lambda w.\lambda l.l$. This result tells us that the abstract value of the `worker` process is ignored and that if `ts` is deterministic ($l = d$), then the result of the function is deterministic as well. In Figure 12 the process topology with the new abstract values in each channel is shown together with the abstract values of the functions involved.

We have previously said that in this scheme the third component of the `manager` process usually depends not only on the initial list of tasks but also on the list produced by the `worker` processes. In such case also in the second analysis we would obtain an n abstract value in channel `y3`.

However in the applications of this scheme (for example, Mandelbrot's algorithm [Rub99]), an ordering process is used to produce the final ouput of the topology, so that it is deterministic. So, the topology happens to be deterministic although the internal part is non-deterministic. This fact can not be detected with an analysis. It would be necessary that the user marked in any way that process as an 'antimerge', that undoes the efect of the `merge` process.

From now on, to avoid confusion, we will use e_i to represent the values belonging to the domains D_{2t_i} and β or β' to represent type variables, so that we can distinguish them from the abstraction function α_t.

7.3 Polymorphism (I)

Polymorphism is now incorporated to the refined analysis. In Figure 13 the abstract interpretation of a type abstraction and a type application is shown.

$$[\![\Lambda\beta.e]\!]_2\ \rho_2 = [\![e]\!]_2\ \rho_2$$

$$[\![e\ t]\!]_2\ \rho_2 = \gamma_{t'\,tinst}([\![e]\!]_2\ \rho_2) \text{ where } e :: \forall\beta.t', tinst = t'[\beta := t]$$

Figure 13: Abstract interpretation of the type abstraction and application

The abstract interpretation of a polymorphic expression is the abstract interpretation of its 'smallest instance', i.e. that instance where K (the basic type) is substituted for the type variables. This is the reason why the abstract domain corresponding to a type variable β is *Basic*, and the abstract domain corresponding a polymorphic type is the abstract domain corresponding to the type without qualifier. Abstract domains are shown in Figure 10. The infimum in these domains are:

$$\bot_{2\beta} = d$$
$$\bot_{2\forall\beta.t'} = \bot_{2t'}$$

So the abstract interpretation of a type abstraction $\Lambda\beta.e :: \forall\beta.t'$ is the abstract interpretation of its body e, what represents the abstract interpretation of the smallest instance of its (polymorphic) type. When an application to a type t is done, the abstract value of the appropriate instance must be calculated. Such abstract value is in fact obtained as an approximation from the abstract value of the smallest instance. From now on, the instance type $t'[\beta := t]$ will be denoted as *tinst*. The approximation to the instance's abstract value is obtained by using a new function $\gamma_{t'\,tinst}$, which we will call the *polymorphic concretisation funcion*. It is defined and studied in detail in Section 7.5. This function *adapts* an abstract value belonging to D_{2t} to an abstract value belonging to D_{2tinst}. That is, it transforms the abstract value of the smallest instance in another abstract value, taken as approximation to the abstract value of the instance corresponding to t. Another function, $\alpha_{tinst\,t'}$, which we will call the *polymorphic abstraction function*, will also be defined below. Together they will be shown to be a Galois insertion.

But first we are going to define the (plain) abstraction and concretisation functions, and also to study some useful properties they have.

7.4 Abstraction and concretisation functions

Given a type t, the abstraction function takes an abstract value in D_{2t} and *flattens* it to a value in *Basic*. We have already seen that this is necessary in constructor applications, as a single basic abstract value represents the whole structure.

Given a type t, the concretisation function γ_t *unflattens* a basic abstract value and produces an abstract value in D_{2t}. This function is used in an algebraic *case* expression. The discriminant has a basic abstract value, as we have flattened all the values of the components. We have to recover the values of the components in order to analyse the right hand sides. But we have lost such information, so the only thing we can do is to give a safe approximation of those values. This is what γ_t does: Given a basic abstract value, it gives a safe approximation to any abstract value that the component could have had, considering how the flattening has been done.

The functions are mutually recursive. The idea of the abstraction function is to come back to the first analysis, that is to flatten the tuples and apply the functions to the unflattening of d for the argument's type. The abstraction function loses information. As an example, if $t = Int \to Int$, $\alpha_t(\lambda z.z) = \alpha_t(\lambda z.d) = d$. In Figure 14 we show the abstraction function for the type $(Int \to Int) \to Int \to Int$.

The idea of the concretisation function is to obtain the best safe approximation to determinism and non-determinism. It tries to recover the information that the abstraction function lost. The function type needs explanation, the rest of them are inmediate. As we have said before, a function is deterministic if it produces deterministic results from deterministic arguments. If the argument is non-deterministic, the safer we can produce is a non-deterministic result: It is like an 'identity' function. So, the unflattening of d for a function type is a function that takes an argument, flattens it to see whether it is deterministic or not and again applies the concretisation function with the type of the result. As an example, if $t = Int \to Int$, $\gamma_t(d) = \lambda z.z$. In Figure 14 we show the concretisation function for the type $(Int \to Int) \to Int \to Int$. The unflattening of n

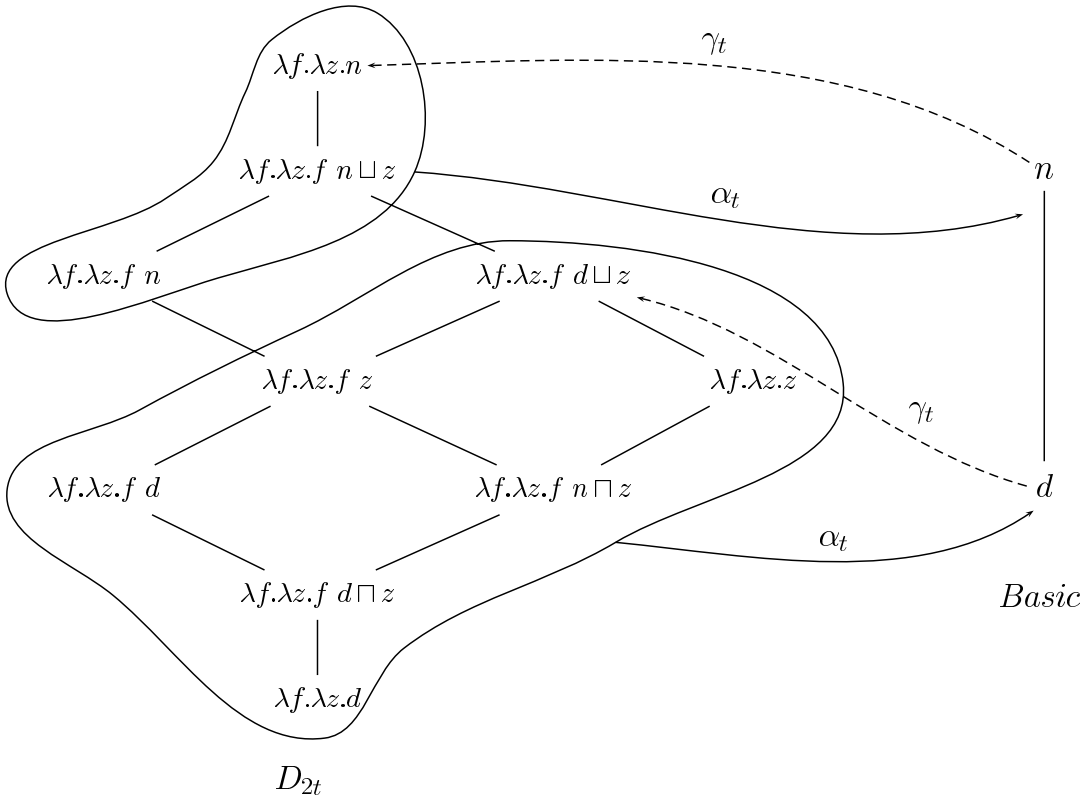


Figure 14: Abstraction and concretisation functions for $t = (Int \to Int) \to Int \to Int$

for a function type is the function that returns a non-deterministic result independently of the argument (it is the top of the abstract domain). For example, if $t = Int \to Int$, $\gamma_t(n) = \lambda z.n$.

Let us see now the formal definitions of the abstraction and concretisation functions.

Definition 7.1 (Abstraction function) *The abstraction function α_t is defined as follows:*

$$
\begin{array}{l}
\alpha_t : D_{2t} \to Basic \\
\alpha_K = \alpha_{T\ t_1 \ldots t_m} = \alpha_\beta = id_{Basic} \\
\alpha_{(t_1,\ldots,t_m)}(e_1,\ldots,e_m) = \bigsqcup_i \alpha_{t_i}(e_i) \\
\alpha_{t_1 \to t_2}(f) = \alpha_{Process\ t_1\ t_2}(f) = \alpha_{t_2}(f(\gamma_{t_1}(d))) \\
\alpha_{\forall \beta.t} = \alpha_t
\end{array}
$$

Definition 7.2 (Concretisation function) *The concretisation function γ_t is defined as follows:*

$$
\begin{array}{l}
\gamma_t : Basic \to D_{2t} \\
\gamma_K = \gamma_{T\ t_1 \ldots t_m} = \gamma_\beta = id_{Basic} \\
\gamma_{(t_1,\ldots,t_m)}(b) = (\gamma_{t_1}(b),\ldots,\gamma_{t_m}(b)) \\
\gamma_{t_1 \to t_2}(b) = \gamma_{Process\ t_1\ t_2}(b) = \left\{ \begin{array}{l} \lambda z.\gamma_{t_2}(n) \text{ if } b = n \\ \lambda z.\gamma_{t_2}(\alpha_{t_1}(z)) \text{ if } b = d \end{array} \right. \\
\gamma_{\forall \beta.t} = \gamma_t
\end{array}
$$

Both functions are monotone and continuous for each type t, as the following proposition asserts.

Proposition 7.3 *For each type t, the functions α_t and γ_t are monotone and continuous.*

Proof (Proposition 7.3) If we prove that for each t, α_t and γ_t are monotone, then we will also have proved they are continuous, as their domains are finite (they satisfy the Ascending Chain Condition) [NNH99]. Monotonicity can be proved by structural induction on t.

We first prove that α_t is monotone:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\alpha_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. By i.h., for each $i = 1, \ldots, m$, α_{t_i} is monotone. Let $e_i, e_i' \in D_{2t_i}$. Let us assume that $(e_1, \ldots, e_m) \sqsubseteq (e_1', \ldots, e_m')$, that is, for each i, $e_i \sqsubseteq e_i'$. We have to prove that $\alpha_t(e_1, \ldots, e_m) \sqsubseteq \alpha_t(e_1', \ldots, e_m')$:

$$\begin{array}{rcll} \alpha_t(e_1, \ldots, e_m) & = & \bigsqcup_i \alpha_{t_i}(e_i) & \text{by definition of } \alpha_t \\ & \sqsubseteq & \bigsqcup_i \alpha_{t_i}(e_i') & \text{by i.h. and } e_i \sqsubseteq e_i' \\ & = & \alpha_t(e_1', \ldots, e_m') & \text{by definition of } \alpha_t \end{array}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. By i.h. each α_{t_i} is monotone, for $i = 1, 2$. We have to prove that α_t is monotone. Let $f, f' \in [D_{2t_1} \to D_{2t_2}]$. Let us assume that $f \sqsubseteq f'$, that is, for each $e \in D_{2t_1}$, $f(e) \sqsubseteq f'(e)$. We have to prove that $\alpha_t(f) \sqsubseteq \alpha_t(f')$:

$$\begin{array}{rcll} \alpha_t(f) & = & \alpha_{t_2}(f(\gamma_{t_1}(d))) & \text{by definition of } \alpha_t \\ & \sqsubseteq & \alpha_{t_2}(f'(\gamma_{t_1}(d))) & \text{by i.h. and } f \sqsubseteq f' \\ & = & \alpha_t(f') & \text{by definition of } \alpha_t \end{array}$$

- $t = \forall \beta.t'$. As $\alpha_t = \alpha_{t'}$, it trivially holds by i.h.

Let us prove now that γ_t is monotone:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These case are trivial as $\gamma_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. By i.h. for each $i = 1, \ldots, m$, γ_{t_i} is monotone. Let $b, b' \in \mathit{Basic}$. Let us assume $b \sqsubseteq b'$ and let us see that $\gamma_t(b) \sqsubseteq \gamma_t(b')$:

$$\begin{array}{rcll} \gamma_t(b) & = & (\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \text{by definition of } \gamma_t \\ & \sqsubseteq & (\gamma_{t_1}(b'), \ldots, \gamma_{t_m}(b')) & \text{by i.h. and } b \sqsubseteq b' \\ & = & \gamma_t(b') & \text{by definition of } \gamma_t \end{array}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. By i.h. each γ_{t_i} is monotone, for $i = 1, 2$. Let $b, b' \in \mathit{Basic}$. Let us assume that $b \sqsubseteq b'$ and let us prove that $\gamma_t(b) \sqsubseteq \gamma_t(b')$.

  When $b = b'$ this is trivial to prove. The non-trivial case is $b = d$ and $b' = n$. In such a case $\gamma_t(b) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$ and $\gamma_t(b') = \lambda z.\gamma_{t_2}(n)$. As n is the top in *Basic*, we have that $\alpha_{t_1}(e) \sqsubseteq n$ for each $e \in D_{2t_1}$. So, by i.h.: $\forall e \in D_{2t_1}.\ \gamma_{t_2}(\alpha_{t_1}(e)) \sqsubseteq \gamma_{t_2}(n)$, from which we can obtain that $\gamma_t(d) \sqsubseteq \gamma_t(n)$.
- $t = \forall \beta.t'$. As $\gamma_t = \gamma_{t'}$, it trivially holds by i.h.

Functions α_t and γ_t are a Galois insertion [NNH99], see Proposition 7.5 (or equivalently a Galois surjection [CC92], or an embedding-closure pair [Bar93]). This means that α_t loses information but just all at once; and that γ_t recovers as much as possible of it, so that another application of α_t does not lose more information. A useful lemma is first presented. It confirms that $\gamma_t(n)$ is the top in D_{2t}.

Lemma 7.4 *For each type t:*

$$\forall e \in D_{2t}.e \sqsubseteq \gamma_t(n)$$

Proof (Lemma 7.4) It can be proved by structural induction on t:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\gamma_t = id_{Basic}$, and n is the top in *Basic*.
- $t = (t_1, \ldots, t_m)$. If $e \in D_{2t}$, then $e = (e_1, \ldots, e_m)$, where $e_i \in D_{2t_i}$ for $i = 1, \ldots, m$. By i.h., for each $i = 1, \ldots, m$ we have that $e_i \sqsubseteq \gamma_{t_i}(n)$, so $e = (e_1, \ldots, e_m) \sqsubseteq (\gamma_{t_1}(n), \ldots, \gamma_{t_m}(n)) = \gamma_t(n)$.
- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $f \in D_{2t}$. And let $e \in D_{2t_1}$. Then $f(e) \in D_{2t_2}$, and by i.h. $f(e) \sqsubseteq \gamma_{t_2}(n)$. On the other hand $\gamma_t(n) = \lambda z.\gamma_{t_2}(n)$. That is, for each $e \in D_{2t_1}$, $f(e) \sqsubseteq (\gamma_t(n))(e)$, so $f \sqsubseteq \gamma_t(n)$.

- $t = \forall\beta.t'$. As $\gamma_t = \gamma'_t$, it holds by i.h..

Let us prove now that α_t and γ_t are a Galois insertion.

Proposition 7.5 *For each type t, α_t and γ_t are a Galois insertion, or equivalently a embedding-closure where γ_t is the embedding and α_t is the closure; that is:*

- $\alpha_t \cdot \gamma_t = id_{Basic}$
- $\gamma_t \cdot \alpha_t \sqsupseteq id_{D_{2t}}$

Proof (Proposition 7.5) This proposition can be proved by structural induction on t.
Let us prove first that $\alpha_t \cdot \gamma_t = id_{Basic}$.

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial, as $\alpha_t = \gamma_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $b \in Basic$. Then:

$$\begin{array}{rcll} \alpha_t(\gamma_t(b)) & = & \alpha_t(\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \text{by definition of } \gamma_t \\ & = & \bigsqcup_i \alpha_{t_i}(\gamma_{t_i}(b)) & \text{by definition of } \alpha_t \\ & = & \bigsqcup_i b & \text{by i.h.} \\ & = & b & \end{array}$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Let $b \in Basic$. We distinguish two cases: $b = d$ and $b = n$.
  If $b = d$, then:

$$\begin{array}{rcll} \alpha_t(\gamma_t(d)) & = & \alpha_t(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) & \text{by definition of } \gamma_t \\ & = & \alpha_{t_2}(\gamma_{t_2}(\alpha_{t_1}(\gamma_{t_1}(d)))) & \text{by definition of } \alpha_t \\ & = & d & \text{by i.h.} \end{array}$$

  If $b = n$, then:

$$\begin{array}{rcll} \alpha_t(\gamma_t(n)) & = & \alpha_t(\lambda z.\gamma_{t_2}(n)) & \text{by definition of } \gamma_t \\ & = & \alpha_{t_2}(\gamma_{t_2}(n)) & \text{by definition of } \alpha_t \\ & = & n & \text{by i.h.} \end{array}$$

- $t = \forall\beta.t'$. It directly holds by i.h. as $\alpha_t = \alpha_{t'}$ y $\gamma_t = \gamma_{t'}$.

It remains to prove that $\gamma_t \cdot \alpha_t \sqsupseteq id_{D_{2t}}$.

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial, as $\alpha_t = \gamma_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $e_i \in D_{2t_i}$ for each $i = 1, \ldots, m$. Then:

$$\begin{array}{rcll} \gamma_t(\alpha_t(e_1, \ldots, e_m)) & = & \gamma_t(\bigsqcup_i \alpha_{t_i}(e_i)) & \text{by definition of } \alpha_t \\ & = & (\gamma_{t_1}(\bigsqcup_i \alpha_{t_i}(e_i)), \ldots, \gamma_{t_m}(\bigsqcup_i \alpha_{t_i}(e_i))) & \text{by definition of } \gamma_t \\ & \sqsupseteq & (\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_m))) & \text{by monotonicity of } \gamma_t \\ & \sqsupseteq & (e_1, \ldots, e_m) & \text{by i.h.} \end{array}$$

- $t = t_1 \to t_2$ o $t = Process\ t_1\ t_2$. Let $f \in [D_{2t_1} \to D_{2t_2}]$. Then:

$$\gamma_t(\alpha_t(f)) \quad = \quad \gamma_t(\alpha_{t_2}(f(\gamma_{t_1}(d)))) \quad \text{by definition of } \alpha_t$$

  We distinguish two cases:

  - If $\alpha_{t_2}(f(\gamma_{t_1}(d)) = d$ then by definition of γ_t we have that $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$. Let $e \in D_{2t_1}$. We want to prove that $f(e) \sqsubseteq (\lambda z.\gamma_{t_2}(\alpha_{t_1}(z)))(e)$. That is, $f(e) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e))$. We distinguish again two cases:

* $e \sqsubseteq \gamma_{t_1}(d)$. By monotonicity of f, $f(e) \sqsubseteq f(\gamma_{t_1}(d))$. So, by monotonicity of α_t, $\alpha_{t_2}(f(e)) \sqsubseteq \alpha_{t_2}(f(\gamma_{t_1}(d)))$ (which is equal to d), that is,

$$\alpha_{t_2}(f(e)) \sqsubseteq d \sqsubseteq \alpha_{t_1}(e)$$

The last inequality holds because d is the infimum in *Basic*. So, by monotonicity of γ_t, $\gamma_{t_2}(\alpha_{t_2}(f(e))) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e))$, and by i.h. we have

$$f(e) \sqsubseteq \gamma_{t_2}(\alpha_{t_2}(f(e))) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e))$$

* $e \not\sqsubseteq \gamma_{t_1}(d)$. In this case $\alpha_{t_1}(e) = n$. Let us prove this by contradiction. Let us assume that $\alpha_{t_1}(e) \neq n$, i.e. that $\alpha_{t_1}(e) = d$. Then $\gamma_{t_1}(\alpha_{t_1}(e)) = \gamma_{t_1}(d)$. By i.h. we know that $e \sqsubseteq \gamma_{t_1}(\alpha_{t_1}(e))$, so $e \sqsubseteq \gamma_{t_1}(d)$, which contradicts the case in which we are.
So, we have to prove that $f(e) \sqsubseteq \gamma_{t_2}(n)$, which holds by Lemma 7.4, as $f(e) \in D_{2t_2}$.

– If $\alpha_{t_2}(f(\gamma_{t_1}(d))) = n$, then $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(n)$, so we have to prove that given $e \in D_{2t_1}$, $f(e) \sqsubseteq \gamma_{t_2}(n)$, which in fact holds by Lemma 7.4, as $f(e) \in D_{2t_2}$.

In this proof some properties have been discovered that may be useful below. The two following lemmas describe the relation between *Basic* and D_{2t} for each t. The first one asserts that all the values below $\gamma_t(d)$ are flattened to d. That is, as we have said before $\gamma_t(d)$ represents d. The second one says that any other value, bigger that $\gamma_t(d)$ o uncomparable with it, is flattened to n. This confirms that $\gamma_t(d)$ is the biggest value that represents d.

Lemma 7.6 *For each type t:*

$$\forall e \in D_{2t}.e \sqsubseteq \gamma_t(d) \Rightarrow \alpha_t(e) = d$$

Proof (Lemma 7.6)
If $e \sqsubseteq \gamma_t(d)$ then, as α_t is monotone $\alpha_t(e) \sqsubseteq \alpha_t(\gamma_t(d))$. By Proposition 7.5, $\alpha_t(\gamma_t(d)) = d$, and as d is the infimum in *Basic*, we have that $\alpha_t(e) = d$.

Lemma 7.7 *For each type t:*

$$\forall e \in D_{2t}.e \not\sqsubseteq \gamma_t(d) \Rightarrow \alpha_t(e) = n$$

Proof (Lemma 7.7)
Let $e \in D_{2t}$ such that $e \not\sqsubseteq \gamma_t(d)$. Then $\alpha_t(e) = n$ necessarily. If $\alpha_t(e) = d$ then by Proposition 7.5 we would have that $e \sqsubseteq \gamma_t(\alpha_t(e)) = \gamma_t(d)$, which would contradict the hypothesis.

From Lemma 7.6 and Lemma 7.7 we obtain that

$$\forall e \in D_{2t}.e \sqsubseteq \gamma_t(d) \Leftrightarrow \alpha_t(e) = d$$

or equivalently

$$\forall e \in D_{2t}.e \not\sqsubseteq \gamma_t(d) \Leftrightarrow \alpha_t(e) = n$$

It has been previously said that $\gamma_t(d)$ is the biggest representative of d, that is, every value less or equal than it flattens to d, and the rest of values flatten to n. It has also been said that a function/process is considered as deterministic when applied to a deterministic argument its result is also deterministic. This means that when a function (resp. process) of type $t_1 \to t_2$ (resp. *Process* t_1 t_2) is applied to a value lesss or equal than $\gamma_{t_1}(d)$, the result is less or equal than $\gamma_{t_2}(d)$:

$$\forall e \sqsubseteq \gamma_{t_1}(d).f(e) \sqsubseteq \gamma_{t_2}(d)$$

The following proposition says that a function of type $t_1 \to t_2$ is deterministic if and only if it is less or equal than $\gamma_{t_1 \to t_2}(d)$, i.e. $\gamma_{t_1 \to t_2}(d)$ is the biggest representative of the deterministic functions of that type. The same happens for a process of type *Process* t_1 t_2: $\gamma_{Process\ t_1\ t_2}(d)$ is the biggest representative of the deterministic processes of that type.

Proposition 7.8 *Given a function $f \in D_{2t}$, where $t = t_1 \to t_2$ or $t =$ Process t_1 t_2, the following holds:*

$$f \sqsubseteq \gamma_t(d) \Leftrightarrow \forall e \sqsubseteq \gamma_{t_1}(d).f(e) \sqsubseteq \gamma_{t_2}(d)$$

Proof (Proposition 7.8)

($\Rightarrow$) Let $f \sqsubseteq \gamma_t(d)$ and $e \sqsubseteq \gamma_{t_1}(d)$. By Lemma 7.6 we have that $\alpha_{t_1}(e) = d$. On the other hand $\gamma_t(d) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$ by definition of γ_t.

As $f \sqsubseteq \gamma_t(d)$ then:

$$f(e) \sqsubseteq \gamma_{t_2}(\alpha_{t_1}(e)) = \gamma_{t_2}(d)$$

($\Leftarrow$) Let us assume that for all $e \sqsubseteq \gamma_{t_1}(d)$ we have that $f(e) \sqsubseteq \gamma_{t_2}(d)$. We have to prove that $f \sqsubseteq \gamma_t(d)$. Let $e \in D_{2t_1}$. We distinguish two cases:

- $e \sqsubseteq \gamma_{t_1}(d)$. In this case, by Lemma 7.6 we have that $\alpha_{t_1}(e) = d$ and:

$$\begin{array}{lcll} f(e) & \sqsubseteq & \gamma_{t_2}(d) & \text{by hypothesis} \\ & = & \gamma_{t_2}(\alpha_{t_1}(e)) & \text{by Lemma 7.6} \\ & = & (\gamma_t(d))(e) & \text{by definition of } \gamma_t \end{array}$$

- Otherwise, by Lemma 7.7 we have that $\alpha_{t_1}(e) = n$. So:

$$\begin{array}{lcll} f(e) & \sqsubseteq & \gamma_{t_2}(n) & \text{by Lemma 7.4} \\ & = & \gamma_{t_2}(\alpha_{t_1}(e)) & \text{by Lemma 7.7} \\ & = & (\gamma_t(d))(e) & \text{by definition of } \gamma_t \end{array}$$

7.5 Polymorphism (II)

In this section, functions $\gamma_{t'\,tinst}$ and $\alpha_{tinst\,t'}$ are formally defined. These functions are similar to α_t and γ_t. In fact they are a generalisation of the latter. The functions α_t and γ_t operated with values in *Basic* and D_{2t}. Now we want to operate with values in the domains $D_{2t'}$ and D_{2tinst}, where $tinst = t'[\beta := t]$, that is between the domain corresponding to the polymorphic type and each of its concrete instances. So in the particular case where $t' = \beta$ they will be equal to α_t and γ_t. This is the reason why we call them in a similar way. From now on, we will call abstraction function indistinctly to α_t and $\alpha_{tinst\,t'}$, and concretisation function to γ_t and $\gamma_{t'\,tinst}$. When we need to specifically refer to $\alpha_{tinst\,t'}$ and $\gamma_{t'\,tinst}$, we will use the 'polymorphic' qualification.

We now define the polymorphic concretisation function $\gamma_{t'\,tinst}$.

Definition 7.9 *(**Polymorphic concretisation function**) Given two types t', t and a type variable β, the polymorphic concretisation function from t' to $tinst = t'[\beta := t]$ is defined by cases on t' as follows ($tinst_i$ represents $t_i[\beta := t]$).*

$$\begin{array}{l} \gamma_{t'\,tinst} : D_{2t'} \to D_{2tinst} \\ \begin{array}{ll} t' = K & \gamma_{t'\,tinst} = id_{Basic} \\ t' = T\ t_1 \ldots t_m & \gamma_{t'\,tinst} = id_{Basic} \\ t' = (t_1, \ldots, t_m) & \gamma_{t'\,tinst}(e_1, \ldots, e_m) = (\gamma_{t_1 tinst_1}(e_1), \ldots, \gamma_{t_m tinst_m}(e_m)) \\ t' = t_1 \to t_2 & \gamma_{t'\,tinst}(f) = \lambda z.\gamma_{t_2 tinst_2}(f(\alpha_{tinst_1\,t_1}(z))) \text{ where } z \in D_{2tinst_1} \\ t' = Process\ t_1\ t_2 & \gamma_{t'\,tinst}(f) = \lambda z.\gamma_{t_2 tinst_2}(f(\alpha_{tinst_1\,t_1}(z))) \text{ where } z \in D_{2tinst_1} \\ t' = \beta & \gamma_{t'\,tinst} = \gamma_t \\ t' = \beta'\ (\neq \beta) & \gamma_{t'\,tinst} = id_{Basic} \\ t' = \forall \beta'.t_1 & \gamma_{t'\,tinst} = \gamma_{t_1 tinst_1} \end{array} \end{array}$$

And also the polymorphic abstraction function.

Definition 7.10 *(**Polymorphic abstraction function**) Given two types t', t and a type variable β, the polymorphic abstraction function from $tinst = t'[\beta := t]$ to t' is defined by cases as follows ($tinst_i$ represents*

$t_i[\beta := t]$).

$$
\begin{array}{ll}
\alpha_{tinst\,t'} : D_{2\,tinst} \to D_{2t'} & \\
t' = K & \alpha_{tinst\,t'} = id_{Basic} \\
t' = T\ t_1 \ldots t_m & \alpha_{tinst\,t'} = id_{Basic} \\
t' = (t_1, \ldots, t_m) & \alpha_{tinst\,t'}(e_1, \ldots, e_m) = (\alpha_{tinst_1\,t_1}(e_1), \ldots, \alpha_{tinst_m\,t_m}(e_m)) \\
t' = t_1 \to t_2 & \alpha_{tinst\,t'}(f) = \lambda z.\alpha_{tinst_2\,t_2}(f(\gamma_{t_1\,tinst_1}(z))) \text{ where } z \in D_{2t_1} \\
t' = Process\ t_1\ t_2 & \alpha_{tinst\,t'}(f) = \lambda z.\alpha_{tinst_2\,t_2}(f(\gamma_{t_1\,tinst_1}(z))) \text{ where } z \in D_{2t_1} \\
t' = \beta & \alpha_{tinst\,t'} = \alpha_t \\
t' = \beta'\ (\neq \beta) & \alpha_{tinst\,t'} = id_{Basic} \\
t' = \forall \beta'.t_1 & \alpha_{tinst\,t'} = \alpha_{tinst_1\,t_1}
\end{array}
$$

Both functions are monotone and continuous, as the following proposition says, and they are a Galois insertion.

Proposition 7.11 *Given two types t, t' and a type variable β, $\gamma_{t'\,tinst}$ and $\alpha_{tinst\,t'}$ are monotone and continuous, where $tinst = t'[\beta := t]$.*

Proof (Proposition 7.11) Let us prove first that they are monotone. Then, as their domains of definition are finite, they are also continuous. Monotonicity can be proved by structural induction on t'. Let us prove it for $\gamma_{t'\,tinst}$, being the proof analogous for $\alpha_{tinst\,t'}$:

- $t' = K$ or $t' = T\ t1 \ldots t_m$ or $t' = \beta'\ (\neq \beta)$. In such cases $\gamma_{t'\,tinst} = id_{Basic}$, so monotonicity holds trivially.
- $t' = (t_1, \ldots, t_m)$. Let $e_i, e'_i \in D_{2t_i}$, where $(e_1, \ldots, e_m) \sqsubseteq (e'_1, \ldots, e'_m)$, that is, $e_i \sqsubseteq e'_i$, for each $i = 1, \ldots, m$. We have to prove that $\gamma_{t'\,tinst}(e_1, \ldots, e_m) \sqsubseteq \gamma_{t'\,tinst}(e'_1, \ldots, e'_m)$. In this case $tinst = (tinst_1, \ldots, tinst_m)$, where $tinst_i = t_i[\beta := t]$ for each $i = 1, \ldots, m$. By i.h. we have that for each $i = 1, \ldots, m$, $\gamma_{t_i\,tinst_i}(e_i) \sqsubseteq \gamma_{t_i\,tinst_i}(e'_i)$. Then, we have

$$
\begin{array}{lcll}
\gamma_{t'\,tinst}(e_1, \ldots, e_m) & = & (\gamma_{t_1\,tinst_1}(e_1), \ldots, \gamma_{t_m\,tinst_m}(e_m)) & \text{by definition of } \gamma_{t'\,tinst} \\
 & \sqsubseteq & (\gamma_{t_1\,tinst_1}(e'_1), \ldots, \gamma_{t_m\,tinst_m}(e'_m)) & \text{by i.h.} \\
 & = & \gamma_{t'\,tinst}(e'_1, \ldots, e'_m) & \text{by definition of } \gamma_{t'\,tinst}
\end{array}
$$

- $t' = t_1 \to t_2$ or $t' = Process\ t_1\ t_2$. Let $f, g \in D_{2t'}$, where $f \sqsubseteq g$. We have to prove that $\gamma_{t'\,tinst}(f) \sqsubseteq \gamma_{t'\,tinst}(g)$. In this case $tinst = tinst_1 \to tinst_2$ (or $tinst = Process\ tinst_1\ tinst_2$) where $tinst_i = t_i[\beta := t]$ for each $i = 1, 2$. Let us prove that $\gamma_{t'\,tinst}(f) \sqsubseteq \gamma_{t'\,tinst}(g)$, that is, that for each $e \in D_{2\,tinst_1}$, $(\gamma_{t'\,tinst}(f))(e) \sqsubseteq (\gamma_{t'\,tinst}(g))(e)$:

$$
\begin{array}{lcll}
(\gamma_{t'\,tinst}(f))(e) & = & \gamma_{t_2\,tinst_2}(f(\alpha_{tinst_1\,t_1}(e))) & \text{by definition of } \gamma_{t'\,tinst} \\
 & \sqsubseteq & \gamma_{t_2\,tinst_2}(g(\alpha_{tinst_1\,t_1}(e))) & \text{by i.h. } \gamma_{t_2\,tinst_2} \text{ is monotone, and by hyp. } f \sqsubseteq g \\
 & = & (\gamma_{t'\,tinst}(g))(e) & \text{by definition of } \gamma_{t'\,tinst}
\end{array}
$$

- $t' = \beta$. Now $tinst = t$, and $\gamma_{t'\,tinst} = \gamma_t$, that is monotone by Proposition 7.3.
- $t' = \forall \beta'.t_1$. It trivially holds by i.h. as $\gamma_{t'\,tinst} = \gamma_{t_1\,tinst_1}$.

In [Bar93] the category of domains and embedding-closure pairs is presented. Two functors, $\times$ and $\to$, can be defined in this category. They build a new embedding-closure pair from two (or more in the case of the cartesian product) embedding-closure pairs:

$$
\begin{array}{l}
\times((f^e, f^c), (g^e, g^c)) = (f^e \times g^e, f^c \times g^c) \\
\to ((f^e, f^c), (g^e, g^c)) = (\lambda h.g^e \cdot h \cdot f^c, \lambda h'.g^c \cdot h' \cdot f^e)
\end{array}
$$

We can rewrite the functions $\alpha_{tinst\,t'}$ and $\gamma_{t'\,tinst}$ by using these functors in the following way:

$$
\begin{array}{ll}
t' = K & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (id_{Basic}, id_{Basic}) \\
t' = T\ t_1 \ldots t_m & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (id_{Basic}, id_{Basic}) \\
t' = (t_1, \ldots, t_m) & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = \times^m((\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1}), \ldots, (\gamma_{t_m\,tinst_m}, \alpha_{tinst_m\,t_m})) \\
t' = t_1 \to t_2 & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = \ \to ((\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1}), (\gamma_{t_2\,tinst_2}, \alpha_{tinst_2\,t_2})) \\
t' = Process\ t_1\ t_2 & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = \ \to ((\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1}), (\gamma_{t_2\,tinst_2}, \alpha_{tinst_2\,t_2})) \\
t' = \beta & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (\gamma_t, \alpha_t) \\
t' = \beta'\ (\neq \beta) & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (id_{Basic}, id_{Basic}) \\
t' = \forall \beta'.t_1 & (\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1})
\end{array}
$$

Now it is very easy to prove the following proposition:

Proposition 7.12 *Given two types t, t' and a type variable β, $\gamma_{t'\,tinst}$ and $\alpha_{tinst\,t'}$ are a embedding-closure pair, that is:*

- $\alpha_{tinst\,t'} \cdot \gamma_{t'\,tinst} = id_{D_{2_{t'}}}$
- $\gamma_{t'\,tinst} \cdot \alpha_{tinst\,t'} \sqsupseteq id_{D_{2_{tinst}}}$

where $tinst = t'[\beta := t]$.

Proof (Proposition 7.12) Both can be proved by structural induction on t' at the same time.

- $t' = K$ or $t' = T\ t_1 \ldots t_m$ or $t' = \beta' (\neq \beta)$. These cases are trivial to prove as $(\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (id_{Basic}, id_{Basic})$.
- $t' = (t_1, \ldots, t_m)$. By i.h., for each $i = 1, \ldots, m$, $(\gamma_{t_i\,tinst_i}, \alpha_{tinst_i\,t_i})$ is an embedding-closure pair, so, as $\times^m$ is a functor in the category of domains and embedding-closure pairs, $(\gamma_{t'\,tinst}, \alpha_{tinst\,t'})$ is an embedding-closure pair as well, as in this case $(\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = \times^m((\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1}), \ldots, (\gamma_{t_m\,tinst_m}, \alpha_{tinst_m\,t_m}))$.
- $t' = t_1 \to t_2$ or $t' = Process\ t_1\ t_2$. By i.h. we have that for each $i = 1, 2$, $(\gamma_{t_i\,tinst_i}, \alpha_{tinst_i\,t_i})$ is an isertion-closure pair. As $\to$ is a functor in the category of domains and embedding-closure pairs, we have that $(\gamma_{t'\,tinst}, \alpha_{tinst\,t'})$ is an embedding-closure pair as well, as in this cases $(\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = \ \to ((\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1}), (\gamma_{t_2\,tinst_2}, \alpha_{tinst_2\,t_2}))$.
- $t' = \beta$. Now $(\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (\gamma_t, \alpha_t)$, which is an embedding-closure pair by Proposition 7.5.
- $t' = \forall \beta'.t_1$. It trivially holds by i.h. as $(\gamma_{t'\,tinst}, \alpha_{tinst\,t'}) = (\gamma_{t_1\,tinst_1}, \alpha_{tinst_1\,t_1})$.

This pair also have some more properties that will be very useful below. They are mainly commutativity properties, shown in Figure 15. There are eight possible ways of combining the arrows in the figure. Six of them are presented in propositions 7.13, 7.14 and Lemma 7.15. The other two are circular combinations (so they represent inequalities or equalities with respect to the identity) easily obtained from the first ones.

The following proposition says first that it is the same to adapt from *Basic* to t' and then to *tinst* than to directly adapt from *Basic* to *tinst*. It also says that it is the same to abstract from *tinst* to t' and then abstract to *Basic* than to directly abstract from *tinst* to *Basic*.

Proposition 7.13 *Given two types t, t' and a type variable β, the following equalities hold:*

- $\gamma_{t'\,tinst} \cdot \gamma_{t'} = \gamma_{tinst}$
- $\alpha_{t'} \cdot \alpha_{tinst\,t'} = \alpha_{tinst}$

where $tinst = t'[\beta := t]$.

Proof (Proposition 7.13) They can be proved simultaneously by structural induction on t':

- $t' = K$ or $t' = T\ t_1 \ldots t_m$ or $t' = \beta'\ (\neq \beta)$. In such cases $\gamma_{t'\,tinst} = \gamma_{t'} = \gamma_{tinst} = id_{Basic} = \alpha_{tinst\,t'} = \alpha_{t'} = \alpha_{tinst}$.

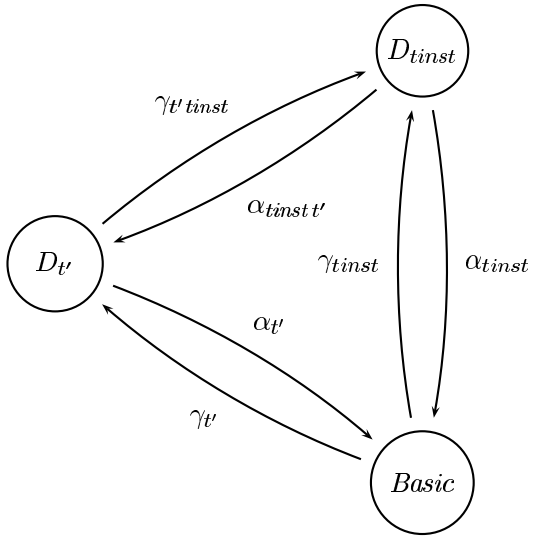


Figure 15: Conmutativity properties of the polymorphic abstraction and concretisation functions

- $t' = (t_1, \ldots, t_m)$. In this case $tinst = (tinst_1, \ldots, tinst_m)$ where $tinst_i = t_i[\beta := t]$, for each $i = 1, \ldots, m$. By i.h. we have for each $i = 1, \ldots, m$ that

  $$\gamma_{t_i\,tinst_i} \cdot \gamma_{t_i} = \gamma_{tinst_i}$$

  and

  $$\alpha_{t_i} \cdot \alpha_{tinst_i\,t_i} = \alpha_{tinst_i}$$

  Let us prove first that $\gamma_{t'\,tinst} \cdot \gamma_{t'} = \gamma_{tinst}$. Let $b \in Basic$.

  $$\begin{array}{rcll} \gamma_{t'\,tinst}(\gamma_{t'}(b)) & = & \gamma_{t'\,tinst}(\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \text{by definition of } \gamma_t \\ & = & (\gamma_{t_1\,tinst_1}(\gamma_{t_1}(b)), \ldots, \gamma_{t_m\,tinst_m}(\gamma_{t_m}(b))) & \text{by definition of } \gamma_{t'\,tinst} \\ & = & (\gamma_{tinst_1}(b), \ldots, \gamma_{tinst_m}(b)) & \text{by i.h.} \\ & = & \gamma_{tinst}(b) & \text{by definition of } \gamma_t \end{array}$$

  Let us prove now that $\alpha_{t'} \cdot \alpha_{tinst\,t'} = \alpha_{tinst}$. Let $e_i \in D_{2tinst_i}$, for each $i = 1, \ldots, m$.

  $$\begin{array}{rcll} \alpha_{t'}(\alpha_{tinst\,t'}(e_1, \ldots, e_m)) & = & \alpha_{t'}(\alpha_{tinst_1\,t_1}(e_1), \ldots, \alpha_{tinst_m\,t_m}(e_m)) & \text{by definition of } \alpha_{tinst\,t'} \\ & = & \bigsqcup_i \alpha_{t_i}(\alpha_{tinst_i\,t_i}(e_i)) & \text{by definition of } \alpha_t \\ & = & \bigsqcup_i \alpha_{tinst_i}(e_i) & \text{by i.h.} \\ & = & \alpha_{tinst}(e_1, \ldots, e_m) & \text{by definition of } \alpha_t \end{array}$$

- $t' = t_1 \to t_2$ or $t' = Process\ t_1\ t_2$. In this case $tinst = tinst_1 \to tinst_2$ (respectively $tinst = Process\ tinst_1\ tinst_2$) where $tinst_i = t_i[\beta := t]$. By i.h. we have that for each $i = 1, 2$,

  $$\gamma_{t_i\,tinst_i} \cdot \gamma_{t_i} = \gamma_{tinst_i}$$

  and

  $$\alpha_{t_i} \cdot \alpha_{tinst_i\,t_i} = \alpha_{tinst_i}$$

  Let us prove first that $\gamma_{t'\,tinst} \cdot \gamma_{t'} = \gamma_{tinst}$. Let $b \in Basic$. We distinguish two cases: $b = d$ and $b = n$.

  If $b = d$:

  $$\begin{array}{rcll} \gamma_{t'\,tinst}(\gamma_{t'}(d)) & = & \gamma_{t'\,tinst}(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) & \text{by definition of } \gamma_t \\ & = & \lambda z.\gamma_{t_2\,tinst_2}(\gamma_{t_2}(\alpha_{t_1}(\alpha_{tinst_1\,t_1}(z)))) & \text{by definition of } \gamma_{t'\,tinst} \\ & = & \lambda z.\gamma_{tinst_2}(\alpha_{tinst_1}(z)) & \text{by i.h.} \\ & = & \gamma_{tinst}(d) & \text{by definition of } \gamma_t \end{array}$$

If $b = n$ then:

$$\begin{array}{rcll} \gamma_{t'\,tinst}(\gamma_{t'}(n)) & = & \gamma_{t'\,tinst}(\lambda z.\gamma_{t_2}(n)) & \text{by definition of } \gamma_t \\ & = & \lambda z.\gamma_{t_2\,tinst_2}(\gamma_{t_2}(n)) & \text{by definition of } \gamma_{t'\,tinst} \\ & = & \lambda z.\gamma_{tinst_2}(n) & \text{by i.h.} \\ & = & \gamma_{tinst}(n) & \text{by definition of } \gamma_t \end{array}$$

Let us prove now that $\alpha_{t'} \cdot \alpha_{tinst\,t'} = \alpha_{tinst}$. Let $f \in D_{2\,tinst}$:

$$\begin{array}{rcll} \alpha_{t'}(\alpha_{tinst\,t'}(f)) & = & \alpha_{t'}(\lambda z.\alpha_{tinst_2\,t_2}(f(\gamma_{t_1\,tinst_1}(z)))) & \text{by definition of } \alpha_{tinst\,t'} \\ & = & \alpha_{t_2}(\alpha_{tinst_2\,t_2}(f(\gamma_{t_1\,tinst_1}(\gamma_{t_1}(d))))) & \text{by definition of } \alpha_t \\ & = & \alpha_{tinst_2}(f(\gamma_{tinst_1}(d))) & \text{by i.h.} \\ & = & \alpha_{tinst}(f) & \text{by definition of } \alpha_t \end{array}$$

- $t' = \beta$. In this case $tinst = t$ and by definition we have that $\gamma_{t'\,tinst} = \gamma_t$, $\alpha_{tinst\,t'} = \alpha_t$, $\gamma_{t'} = \alpha_{t'} = id_{Basic}$, so

$$\begin{array}{rcll} \gamma_{t'\,tinst} \cdot \gamma_{t'} & = & \gamma_t \cdot id_{Basic} & \text{by definition of } \gamma_{t'\,tinst} \text{ and } \gamma_t \\ & = & \gamma_{tinst} & \text{as } tinst = t \end{array}$$

  and

$$\begin{array}{rcll} \alpha_{t'} \cdot \alpha_{tinst\,t'} & = & id_{Basic} \cdot \alpha_t & \text{by definition of } \alpha_{tinst\,t'} \text{ and } \alpha_t \\ & = & \alpha_{tinst} & \text{as } tinst = t \end{array}$$

- $t' = \forall\beta'.t_1$. In this case $tinst = tinst_1 = t_1[\beta := t]$ and then

$$\begin{array}{rcll} \gamma_{t'\,tinst} \cdot \gamma_{t'} & = & \gamma_{t_1\,tinst_1} \cdot \gamma_{t_1} & \text{by definition of } \gamma_{t'\,tinst} \text{ and } \gamma_t \\ & = & \gamma_{tinst_1} & \text{by i.h.} \\ & = & \gamma_{tinst} & \text{by definition of } \gamma_t \end{array}$$

  and

$$\begin{array}{rcll} \alpha_{t'} \cdot \alpha_{tinst\,t'} & = & \alpha_{t_1} \cdot \alpha_{tinst_1\,t_1} & \text{by definition } \alpha_{tinst\,t'} \text{ and } \alpha_t \\ & = & \alpha_{tinst_1} & \text{by i.h.} \\ & = & \alpha_{tinst} & \text{by definition of } \alpha_t \end{array}$$

The following proposition tells us that it is the same to adapt from t' to *tinst* and then abstract to *Basic* than to directly abstract from t' to *Basic*. It also says that it is the same to adapt from *Basic* to *tinst* and then abstract to t' than to directly adapt from *Basic* to t'.

Proposition 7.14 *Given two types t, t' and a type variable β, the following equalities hold:*

- $\alpha_{tinst} \cdot \gamma_{t'\,tinst} = \alpha_{t'}$
- $\alpha_{tinst\,t'} \cdot \gamma_{tinst} = \gamma_{t'}$

where $tinst = t'[\beta := t]$.

Proof (Proposition 7.14) Both items can be proved easily from the previous proposition:

$$\begin{array}{rcll} \alpha_{tinst} \cdot \gamma_{t'\,tinst} & = & (\alpha_{t'} \cdot \alpha_{tinst\,t'}) \cdot \gamma_{t'\,tinst} & \text{by Proposition 7.13} \\ & = & \alpha_{t'} & \text{by associativity and Proposition 7.12} \end{array}$$

$$\begin{array}{rcll} \alpha_{tinst\,t'} \cdot \gamma_{tinst} & = & \alpha_{tinst\,t'} \cdot (\gamma_{t'\,tinst} \cdot \gamma_{t'}) & \text{by Proposition 7.13} \\ & = & \gamma_{t'} & \text{by associativity and Proposition 7.12} \end{array}$$

We have already seen that the semantics of a type application is obtained using the function $\gamma_{t'\,tinst}$. But we could also have used the semantics of the instance in another way, first abstracting to *Basic* the abstract value of the smallest instance and then adapting to *tinst*, that is:

$$[\![e\ t]\!]_2\ \rho_2 = \gamma_{tinst}(\alpha_{t'}([\![e]\!]_2\ \rho_2))$$

The first item of the following lemma tells us that this would have been a worse choice, that is, with it we would lose more information. The second one completes the conmutativity properties of interest, shown in Figure 15.

Lemma 7.15 *Given two types t, t' and a type variable β, we have:*

- $\gamma_{tinst} \cdot \alpha_{t'} \sqsupseteq \gamma_{t'\,tinst}$
- $\gamma_{t'} \cdot \alpha_{tinst} \sqsupseteq \alpha_{tinst\,t'}$

where $tinst = t'[\beta := t]$.

Proof (Lemma 7.15) It can be proved directly from previous propositions:

$$\begin{array}{lcll} \gamma_{tinst} \cdot \alpha_{t'} & = & (\gamma_{t'\,tinst} \cdot \gamma_{t'}) \cdot \alpha_{t'} & \text{by Proposition 7.13} \\ & \sqsupseteq & \gamma_{t'\,tinst} & \text{by associativity and Proposition 7.5} \\ \\ \gamma_{t'} \cdot \alpha_{tinst} & = & \gamma_{t'} \cdot (\alpha_{t'} \cdot \alpha_{tinst\,t'}) & \text{by Proposition 7.13} \\ & \sqsupseteq & \alpha_{tinst\,t'} & \text{by associativity and Proposition 7.5} \end{array}$$

A generalization of the pair $(\gamma_{t'\,tinst}, \alpha_{tinst\,t'})$ can be defined, where several type variables are instantiated in sequence. This corresponds to a type $\forall\beta_1.\ldots.\forall\beta_m.t'$ and an instantiation $tinst = t'[\beta_1 := t_1, \ldots, \beta_m := t_m]$. The analysis domains together with these pairs as morphisms form a category (by Proposition 7.13).

8 Relation between the analyses

In this section the relation between the first and the second analysis is studied. Grosso modo we are going to see that the first analysis is worse than the second one, but in a safe way, that is, it is a safe approximation to the second analysis. In particular we are going to prove that the first analysis is an upper aproximation to a widening [CC92] of the second analysis (see Theorem 8.7). In Figure 16 an scheme of the functions defined in this section and their definition domains are shown. Many propositions in this section can be easily visualized using this figure.

In the first analysis there are neither nested tuples nor abstract functions. However, the abstract value a of a function in such analysis uniquely represents an abstract function. We already know that a represents the behaviour of the function given a deterministic argument. If the argument is non-deterministic, or it has any 'dose' of non-determinism, the result of the application is always non-deterministic. What we are really doing is to simulate the behaviour of an abstract function. Let us assume that the abstract value of a function in the fist analysis is a. Which is the abstract function a represents? It is a function that takes an abstract value; if that value is non-deterministic the result is also non-deterministic, and if it is deterministic it gives as result the abstract value a. This leads us to define a function η_t, called the *expansion function* that *expands* the abstract values obtained in the first analysis into abstract values belonging to the second analysis abstract domains. As in the first analysis there are no nested tuples, it will be necessary to adapt each of the basic values to the types of the components. In the cases of basic type, algebraic type and type variables it is trivially the identity function, as in both analyses the corresponding abstract domains are *Basic*. The polymorphic type case is also easy as it is reduced to the type without qualifier.

Definition 8.1 (Expansion function) *The expansion function η_t is defined as follows:*

$$\begin{array}{l} \eta_t : D_{1t} \to D_{2t} \\ \eta_K = id_{Basic} \\ \eta_{T\ t_1 \ldots t_m} = id_{Basic} \\ \eta_{(t_1,\ldots,t_m)}(b_1, \ldots, b_m) = (\gamma_{t_1}(b_1), \ldots, \gamma_{t_m}(b_m)) \\ \\ \eta_{t_1 \to t_2}(a) = \lambda z. \begin{cases} \eta_{t_2}(a) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases} \\ \\ \eta_{Process\ t_1\ t_2}(a) = \lambda z. \begin{cases} \eta_{t_2}(a) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases} \\ \\ \eta_\beta = id_{Basic} \\ \eta_{\forall\beta.t} = \eta_t \end{array}$$

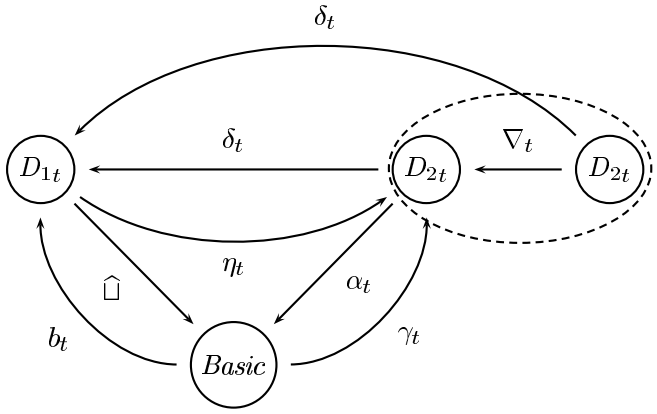


Figure 16: Representative scheme of the functions in this section

We can also look at the second analysis from the point of view of the first one. We just flatten the tuples up to the first level and apply the functions to $\gamma_t(d)$, to obtain the behaviour of the function when it is applied to a deterministic argument. This is represented by the *compression function* δ_t defined below. It *compresses* the abstract values obtained in the second analysis to abstract values of the first analysis domains. The cases of basic type, algebraic type, type variable and polymorphic type are similar to those of the expansion function.

Definition 8.2 (Compression function) *The compression function δ_t is defined ad follows:*

$$
\begin{array}{l}
\delta_t : D_{2t} \to D_{1t} \\
\delta_K = id_{Basic} \\
\delta_{T\ t_1 \ldots t_m} = id_{Basic} \\
\delta_{(t_1,\ldots,t_m)}(e_1,\ldots,e_m) = (\alpha_{t_1}(e_1),\ldots,\alpha_{t_m}(e_m)) \\
\delta_{t_1 \to t_2}(f) = \delta_{t_2}(f(\gamma_{t_1}(d))) \\
\delta_{Process\ t_1\ t_2}(f) = \delta_{t_2}(f(\gamma_{t_1}(d))) \\
\delta_\beta = id_{Basic} \\
\delta_{\forall \beta.t} = \delta_t
\end{array}
$$

Proposition 8.3 *For each type t, the functions δ_t and η_t are monotone and continuous.*

Proof (Proposition 8.3) If we prove that they are monotone, as their domains of definition are finite, then they will also be continuous. The monotonicity can be proved by structural induction on t.

Let us prove now that δ_t is monotone:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\delta_t = id_{Basic}$.
- $t = (t_1,\ldots,t_m)$. Let $e_i, e_i' \in D_{2t_i}$. Let us assume that $(e_1,\ldots,e_m) \sqsubseteq (e_1',\ldots,e_m')$, that is, for each $i = 1,\ldots,m$, $e_i \sqsubseteq e_i'$. We have to prove that $\delta_t(e_1,\ldots,e_m) \sqsubseteq \delta_t(e_1',\ldots,e_m')$:

$$
\begin{array}{rcll}
\delta_t(e_1,\ldots,e_m) & = & (\alpha_{t_1}(e_1),\ldots,\alpha_{t_m}(e_m)) & \text{by definition of } \delta_t \\
& \sqsubseteq & (\alpha_{t_1}(e_1'),\ldots,\alpha_{t_m}(e_m')) & \text{by monotonicity of } \alpha_t \\
& = & \delta_t(e_1',\ldots,e_m') & \text{by definition of } \delta_t
\end{array}
$$

- $t = t_1 \to t_2$ or $t = Process\ t_1\ t_2$. Let $f, f' \in [D_{2t_1} \to D_{2t_2}]$. Let us assume that $f \sqsubseteq f'$, that is, for each $e \in D_{2t_1}$, $f(e) \sqsubseteq f'(e)$. We have to prove that $\delta_t(f) \sqsubseteq \delta_t(f')$.

$$
\begin{array}{rcll}
\delta_t(f) & = & \delta_{t_2}(f(\gamma_{t_1}(d))) & \text{by definition of } \delta_t \\
& \sqsubseteq & \delta_{t_2}(f'(\gamma_{t_1}(d))) & \text{by i.h. and } f \sqsubseteq f' \\
& = & \delta_t(f') & \text{by definition of } \delta_t
\end{array}
$$

- $t = \forall \beta.t'$. As $\delta_t = \delta_{t'}$, it trivially holds by i.h.

And now let us prove the monotonicity of η_t.

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\eta_t = id_{Basic}$.

- $t = (t_1, \ldots, t_m)$. Let $b_i, b'_i \in \mathit{Basic}$. Let us assume that $(b_1, \ldots, b_m) \sqsubseteq (b'_1, \ldots, b'_m)$, that is, for each $i = 1, \ldots, m$, $b_i \sqsubseteq b'_i$. We have to prove that $\eta_t(b_1, \ldots, b_m) \sqsubseteq \eta_t(b'_1, \ldots, b'_m)$:

$$\begin{array}{lcll} \eta_t(b_1, \ldots, b_m) & = & (\gamma_{t_1}(b_1), \ldots, \gamma_{t_m}(b_m)) & \\ & \sqsubseteq & (\gamma_{t_1}(b'_1), \ldots, \gamma_{t_m}(b'_m)) & \text{by monotonicity of } \gamma_t \\ & = & \eta_t(b'_1, \ldots, b'_m) & \end{array}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $a, a' \in D_{1t} = D_{1t_2}$. Let us assume that $a \sqsubseteq a'$. We have to prove that for each $e \in D_{2t_1}$, $(\eta_t(a))(e) \sqsubseteq (\eta_t(a'))(e)$. Let $e \in D_{2t_1}$. We distinguish two cases: $e \sqsubseteq \gamma_{t_1}(d)$ or $e \not\sqsubseteq \gamma_{t_1}(d)$.

  If $e \sqsubseteq \gamma_{t_1}(d)$, then

$$\begin{array}{lcll} (\eta_t(a))(e) & = & \eta_{t_2}(a) & \text{by definition of } \eta_t \\ & \sqsubseteq & \eta_{t_2}(a') & \text{by i.h. and } a \sqsubseteq a' \\ & = & (\eta_t(a'))(e) & \text{by definition of } \eta_t \end{array}$$

  Otherwise, $e \not\sqsubseteq \gamma_{t_1}(d)$, and by definition of η_t: $(\eta_t(a))(e) = \gamma_{t_2}(n) = (\eta_t(a'))(e)$.

- $t = \forall \beta.t'$. As $\eta_t = \eta_{t'}$, it trivially holds by i.h.

The following proposition asserts that given an abstract value in D_{2t}, if we apply δ_t to it to obtain a value in D_{1t} and then we apply $\widehat{\sqcup}$ to the result (it could be a tuple of basic abstract values) we obtain the same as applying α_t to it directly to obtain a value in Basic. This shows a similarity between the abstraction process used in the constructors applications and the compression function δ_t. In the end the ideas are the same.

Proposition 8.4 *For each type t the following holds:*

$$\widehat{\sqcup} \cdot \delta_t = \alpha_t$$

Proof (Proposition 8.4) It can be proved by structural induction on t:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\delta_t = \alpha_t = id_{\mathit{Basic}}$.
- $t = (t_1, \ldots, t_m)$. Let $e_i \in D_{2t_i}$, where $i = 1, \ldots, m$. Then:

$$\begin{array}{lcll} \widehat{\sqcup}(\delta_t(e_1, \ldots, e_m)) & = & \widehat{\sqcup}(\alpha_{t_1}(e_1), \ldots, \alpha_{t_m}(e_m)) & \text{by definition of } \delta \\ & = & \bigsqcup_i \alpha_{t_i}(e_i) & \text{by definition of } \widehat{\sqcup} \\ & = & \alpha_t(e_1, \ldots, e_m) & \text{by definition of } \alpha_t \end{array}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $f \in D_{2t}$. Then:

$$\begin{array}{lcll} \widehat{\sqcup}(\delta_t(f)) & = & \widehat{\sqcup}(\delta_{t_2}(f(\gamma_{t_1}(d)))) & \text{by definition of } \delta_t \\ & = & \alpha_{t_2}(f(\gamma_{t_1}(d))) & \text{by i.h.} \\ & = & \alpha_t(f) & \text{by definition of } \alpha_t \end{array}$$

- $t = \forall \beta.t'$. In this case $\delta_t = \delta'_t$ and $\alpha_t = \alpha_{t'}$, so it trivially holds by i.h..

Let us prove now that δ_t and η_t are an embedding-closure pair.

Proposition 8.5 *For each type t, the following holds:*

- $\delta_t \cdot \eta_t = id_{D_{1t}}$
- $\eta_t \cdot \delta_t \sqsupseteq id_{D_{2t}}$

that is, they are an embedding-closure pair, or a Galois insertion.

Proof (Proposition 8.5) Both things can be proved by structural induction on t. Let us prove first that $\delta_t \cdot \eta_t = id_{D_{1t}}$:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $D_{1t} = \mathit{Basic}$ and $\eta_t = \delta_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $b_i \in \mathit{Basic}$, where $i = 1, \ldots, m$. Then:

$$\begin{array}{lcll} \delta_t(\eta_t(b_1, \ldots, b_m)) & = & \delta_t(\gamma_{t_1}(b_1), \ldots, \gamma_{t_m}(b_m)) & \text{by definition of } \eta_t \\ & = & (\alpha_{t_1}(\gamma_{t_1}(b_1)), \ldots, \alpha_{t_m}(\gamma_{t_m}(b_m))) & \text{by definition of } \delta_t \\ & = & (b_1, \ldots, b_m) & \text{by Proposition 7.5 in each component} \end{array}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $a \in D_{1t}$. Then

$$\begin{array}{lcll} \delta_t(\eta_t(a)) & = & \delta_t\left(\lambda z. \left\{ \begin{array}{l} \eta_{t_2}(a) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{array} \right.\right) & \text{by definition of } \eta_t \\ & = & \delta_{t_2}(\eta_{t_2}(a)) & \text{by definition of } \delta_t \\ & = & a & \text{by i.h.} \end{array}$$

- $t = \forall \beta.t'$. In this case $D_{1t} = D_{1t'}$, $\eta_t = \eta_{t'}$ and $\delta_t = \delta_{t'}$, so it trivially holds by i.h..

Let us prove now that $\eta_t \cdot \delta_t \sqsupseteq id_{D_{2t}}$:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $D_{2t} = \mathit{Basic}$ and $\eta_t = \delta_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $e_i \in D_{2t_i}$, where $i = 1, \ldots, m$. Entonces, tenemos:

$$\begin{array}{lcll} \eta_t(\delta_t(e_1, \ldots, e_m)) & = & \eta_t(\alpha_{t_1}(e_1), \ldots, \alpha_{t_m}(e_m)) & \text{by definition of } \delta_t \\ & = & (\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_m))) & \text{by definition of } \eta_t \\ & \sqsupseteq & (e_1, \ldots, e_m) & \text{by Proposition 7.5} \end{array}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $f \in D_{2t}$. Then

$$\begin{array}{lcll} \eta_t(\delta_t(f)) & = & \eta_t(\delta_{t_2}(f(\gamma_{t_1}(d)))) & \text{by definition of } \delta_t \\ & = & \lambda z. \left\{ \begin{array}{l} \eta_{t_2}(\delta_{t_2}(f(\gamma_{t_1}(d)))) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{array} \right. & \text{by definition of } \eta_t \end{array}$$

  Let $e \in D_{2t_1}$. We have to prove that $f(e) \sqsubseteq (\eta_t(\delta_t(f)))(e)$. We distinguish two cases::

  - $e \sqsubseteq \gamma_{t_1}(d)$. In this case $(\eta_t(\delta_t(f)))(e) = \eta_{t_2}(\delta_{t_2}(f(\gamma_{t_1}(d))))$. As f is continuous and consequently monotone, we have that $f(e) \sqsubseteq f(\gamma_{t_1}(d))$. So

$$\begin{array}{lcll} (\eta_t(\delta_t(f)))(e) & = & & \\ \eta_{t_2}(\delta_{t_2}(f(\gamma_{t_1}(d)))) & \sqsupseteq & f(\gamma_{t_1}(d)) & \text{by i.h.} \\ & \sqsupseteq & f(e) & \text{as } f \text{ is monotone} \end{array}$$

  - $e \not\sqsubseteq \gamma_{t_1}(d)$. In this case $(\eta_t(\delta_t(f)))(e) = \gamma_{t_2}(n)$. By Lemma 7.4 $\gamma_{t_2}(n) \sqsupseteq f(e)$, as $f(e) \in D_{2t_2}$.

- $t = \forall \beta.t'$. In this case $D_{2t} = D_{2t'}$, $\eta_t = \eta_{t'}$ and $\delta_t = \delta_{t'}$, so it trivially holds by i.h..

This last proposition tells us that $\eta_t \cdot \delta_t$ is a widening operator in D_{2t}. This composition will be widely used in the following, so we define $\nabla_t = \eta_t \cdot \delta_t$. We will call it the *widening function*. By unfolding its definition we obtain the following:

$$\begin{array}{l} \nabla_t : D_{2t} \to D_{2t} \\ \nabla_K = id_{Basic} \\ \nabla_{T\ t_1 \ldots t_m} = id_{Basic} \\ \nabla_{(t_1,\ldots,t_m)}(e_1, \ldots, e_m) = (\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_m))) \end{array}$$

$$\nabla_{t_1 \to t_2}(f) = \lambda z. \begin{cases} \nabla_{t_2}(f(\gamma_{t_1}(d))) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases}$$

$$\nabla_{Process\ t_1\ t_2}(f) = \lambda z. \begin{cases} \nabla_{t_2}(f(\gamma_{t_1}(d))) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases}$$

$$\nabla_{\beta} = id_{Basic}$$

$$\nabla_{\forall \beta.t} = \nabla_t$$

The last proposition tell us that for each type t, the range of ∇_t is isomorphic to D_{1t}: $\nabla_t(D_{2t}) \simeq D_{1t}$ (as ∇_t is idempotent). This means that the range of ∇_t is a subdomain of D_{2t}, where we have lost the additional information provided by the second analysis. For example, we still have nested tuples, but they are maintained in a fictitious way, that is, they have been flattened up to the first level and then unflattened again. So all the internal tuples will be formed by only n or only d. For example both $((n,d),d,(d,d))$ and $((d,n),d,(d,d))$ are transformed by ∇_t into $((n,n),d,(d,d))$. We also have abstract functions, but only some of them; those that can be represented with an abstract value in the domains of the first analysis: these are the functions such that for all the values below $\gamma_{t_1}(d)$, the result of the function is the same as the result obtained for $\gamma_{t_1}(d)$ and for the rest of the values the result is the top of the corresponding domain, $\gamma_{t_2}(n)$.

We have already seen that $\gamma_t \cdot \alpha_t \sqsupseteq id_{D_{2t}}$ and also that $\nabla_t \sqsupseteq id_{D_{2t}}$. But, how are they related? The following propositions tells us that ∇_t is smaller (i.e. better) than $\gamma_t \cdot \alpha_t$.

Proposition 8.6 *For each type t the following holds:*

$$\nabla_t \sqsubseteq \gamma_t \cdot \alpha_t$$

Proof (Proposition 8.6) It can be proved by structural induction on t:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\nabla_t = \alpha_t = \gamma_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $e_i \in D_{2t_i}$ for $i = 1, \ldots, m$. Then:

$$\begin{array}{rcll} \nabla_t(e_1,\ldots,e_m) & = & (\gamma_{t_1}(\alpha_{t_1}(e_1)),\ldots,\gamma_{t_m}(\alpha_{t_m}(e_m))) & \text{by definition of } \nabla_t \\ & \sqsubseteq & (\gamma_{t_1}(\bigsqcup_i \alpha_{t_i}(e_i)),\ldots,\gamma_{t_m}(\bigsqcup_i \alpha_{t_i}(e_i))) & \text{as } \alpha_{t_j}(e_j) \sqsubseteq \bigsqcup_i \alpha_{t_i}(e_i) \text{ for each } j = 1,\ldots,m \\ & & & \text{and } \gamma_{t_i} \text{ is monotone by Proposition 7.3} \\ & = & \gamma_t(\alpha_t(e_1,\ldots,e_m)) & \text{by definition of } \alpha_t \text{ and } \gamma_t \end{array}$$

- $t = t_1 \to t_2$ or $t = Process\ t_1\ t_2$. Let $f \in [D_{2t_1} \to D_{2t_2}]$ By definition of ∇_t

$$\nabla_t(f) = \lambda z. \begin{cases} \nabla_{t_2}(f(\gamma_{t_1}(d))) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases}$$

  By definition of γ_t and α_t

$$\gamma_t(\alpha_t(f)) = \lambda z. \begin{cases} \gamma_{t_2}(\alpha_{t_1}(z)) \text{ if } \alpha_{t_2}(f(\gamma_{t_1}(d))) = d \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases}$$

  We distinguish two cases:

  - $\alpha_{t_2}(f(\gamma_{t_1}(d))) = n$. In this case $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(n)$. Let $e \in D_{2t_1}$. We have that $(\gamma_t(\alpha_t(f)))(e) = \gamma_{t_2}(n)$. So, by Lemma 7.4 we have that $\gamma_{t_2}(n) \sqsupseteq (\nabla_t(f))(e)$ as $(\nabla_t(f))(e) \in D_{2t_2}$.
  - $\alpha_{t_2}(f(\gamma_{t_1}(d))) = d$. Now $\gamma_t(\alpha_t(f)) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$. Let $e \in D_{2t_1}$. We distinguish again two cases: $e \sqsubseteq \gamma_{t_1}(d)$ or not.
    If $e \sqsubseteq \gamma_{t_1}(d)$ then $(\nabla_t(f))(e) = \nabla_{t_2}(f(\gamma_{t_1}(d)))$. By Lemma 7.6 $\alpha_{t_1}(e) = d$. So, what we really want to prove is that $\nabla_{t_2}(f(\gamma_{t_1}(d))) \sqsubseteq \gamma_{t_2}(d)$. By i.h. we know that $\nabla_{t_2} \sqsubseteq \gamma_{t_2} \cdot \alpha_{t_2}$, so:

$$\nabla_{t_2}(f(\gamma_{t_1}(d))) \sqsubseteq \gamma_{t_2}(\alpha_{t_2}(f(\gamma_{t_1}(d)))) = \gamma_{t_2}(d)$$

    In case $e \not\sqsubseteq \gamma_{t_1}(d)$, then $(\nabla_t(f))(e) = \gamma_{t_2}(n)$. By Lemma 7.7 $\alpha_{t_1}(e) = n$, so $\gamma_{t_2}(\alpha_{t_1}(e)) = \gamma_{t_2}(n)$, and we have the equality.

- $t = \forall\beta.t'$. As $\alpha_t = \alpha_{t'}$, $\gamma_t = \gamma_{t'}$ and $\nabla_t = \nabla_{t'}$ by definition, it trivially holds by i.h.

The following theorem stablishes that the first analysis is a safe (upper) approximation to the widening of the second analysis. As a corollary of this theorem we obtain the correctness of the first analysis with respect to the second one: If the first analysis tells us that an expression is deterministic then the second analysis also tells us that it is deterministic, probably with some additional detail as the independence of the output with respect to the input in a function/process.

Theorem 8.7 *If for each variable $v :: t_v$, we have that $\rho_1(v) \sqsupseteq \delta_{t_v}(\rho_2(v))$ then:*

$$\forall e :: t_e.[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_2\ \rho_2)$$

Or equivalently, by Proposition 8.5:

$$\forall e :: t_e.\eta_{t_e}([\![e]\!]_1\ \rho_1) \sqsupseteq \nabla_{t_e}([\![e]\!]_2\ \rho_2)$$

Corollary 8.8 *If for each variable $v :: t_v$, we have that $\rho_1(v) \sqsupseteq \delta_{t_v}(\rho_2(v))$ then:*

$$\forall e :: t_e.\ [\![e]\!]_1\ \rho_1 = d_{t_e} \Rightarrow [\![e]\!]_2\ \rho_2 \sqsubseteq \gamma_{t_e}(d)$$

To prove this theorem and its corollary we need to prove before some properties of the functions that are involved.

The following proposition relates the adaptation function in the first analysis and the concretisation function in the second one. The are made equal through the application of δ_t.

Proposition 8.9 *For each type t the following holds:*

$$\forall b \in \mathit{Basic}.\ b_t = (\delta_t \cdot \gamma_t)(b)$$

Proof (Proposition 8.9) It can be proved by structural induction on t:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $b_t = b$ and $\delta_t = \gamma_t = id_{\mathit{Basic}}$.
- $t = (t_1, \ldots, t_m)$. In this case we have:

$$\begin{array}{rcll} \delta_t(\gamma_t(b)) & = & \delta_t(\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \text{by definition of } \gamma_t \\ & = & (\alpha_{t_1}(\gamma_{t_1}(b)), \ldots, \alpha_{t_m}(\gamma_{t_m}(b))) & \text{by definition of } \delta_t \\ & = & (b, \ldots, b) & \text{by Proposition 7.5} \\ & = & b_t & \text{by definition of } b_t \end{array}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. We distinguish two cases:: $b = d$ and $b = n$.
  - $b = d$. In this case:

$$\begin{array}{rcll} \delta_t(\gamma_t(d)) & = & \delta_t(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) & \text{by definition of } \gamma_t \\ & = & \delta_{t_2}(\gamma_{t_2}(\alpha_{t_1}(\gamma_{t_1}(d)))) & \text{by definition of } \delta_t \\ & = & \delta_{t_2}(\gamma_{t_2}(d)) & \text{by Proposition 7.5} \\ & = & d_{t_2} & \text{by i.h.} \\ & = & d_t & \text{by definition of } b_t \end{array}$$

  - $b = n$. Now we have

$$\begin{array}{rcll} \delta_t(\gamma_t(n)) & = & \delta_t(\lambda z.\gamma_{t_2}(n)) & \text{by definition of } \gamma_t \\ & = & \delta_{t_2}(\gamma_{t_2}(n)) & \text{by definition of } \delta_t \\ & = & n_{t_2} & \text{by i.h.} \\ & = & n_t & \text{by definition of } b_t \end{array}$$

- $t = \forall\beta.t'$. In this case $b_t = b_{t'}$, $\delta_t = \delta'_t$ and $\gamma_t = \gamma_{t'}$, so it trivially holds by i.h..

The following proposition just tells us that δ_t is strict.

Proposition 8.10 *For each type t the following holds:*

$$\delta_t(\perp_{2t}) = d_t$$

Proof (Proposition 8.10) This is obtained as a consequence of Proposition 8.5. In an embedding-closure pair, the closure is strict, that is, the image of the infimum is the infimum. And clearly d_t is the infimum in D_{1t}, and $\perp_{2t}$ is the infimum in D_{2t}.

The two following propositions relate the widening function ∇_t with the abstraction and concretisation functions α_t and γ_t. The basic idea is that once we have gone up the domain with ∇_t, the functions α_t and γ_t have the same effect in the (subdomain of D_{2t} that is the) range of ∇_t than in the whole domain. This means that in fact α_t and γ_t are moving inside this subdomain. In Figure 16 this fact is represented by two D_{2t} domains encircled by a dashed line, where α_t and γ_t appear between the range of ∇_t and *Basic*.

Proposition 8.11 *For each type t the following holds:*

$$\alpha_t \cdot \nabla_t = \alpha_t$$

Proof (Proposition 8.11) This can be proved by structural induction on t:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\nabla_t = \alpha_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $e_i \in D_{2t_i}$ where $i = 1, \ldots, m$. Then:

$$\begin{array}{rcll}
\alpha_t(\nabla_t(e_1, \ldots, e_m)) & = & \alpha_t(\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_m))) & \text{by definition of } \nabla_t \\
& = & \bigsqcup_i \alpha_{t_i}(\gamma_{t_i}(\alpha_{t_i}(e_i))) & \text{by definition of } \alpha_t \\
& = & \bigsqcup_i \alpha_{t_i}(e_i) & \text{by Proposition 7.5} \\
& = & \alpha_t(e_1, \ldots, e_m) & \text{by definition of } \alpha_t
\end{array}$$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $f \in [D_{2t_1} \to D_{2t_2}]$. Then:

$$\begin{array}{rcll}
\alpha_t(\nabla_t(f)) & = & \alpha_{t_2}(\nabla_{t_2}(f(\gamma_{t_1}(d)))) & \text{by def. of } \alpha_t \text{ and } \nabla_t \\
& = & \alpha_{t_2}(f(\gamma_{t_1}(d))) & \text{by i.h.} \\
& = & \alpha_t(f) & \text{by definition of } \alpha_t
\end{array}$$

- $t = \forall\beta.t'$. It trivially holds by i.h. as $\nabla_t = \nabla_{t'}$ and $\alpha_t = \alpha_{t'}$.

Proposition 8.12 *For each type t the following holds:*

$$\nabla_t \cdot \gamma_t = \gamma_t$$

Proof (Proposition 8.12) This can be proved by structural induction on t:

- $t = K$ or $t = T\ t_1 \ldots t_m$ or $t = \beta$. These cases are trivial as $\nabla_t = \gamma_t = id_{Basic}$.
- $t = (t_1, \ldots, t_m)$. Let $b \in \mathit{Basic}$. In this case:

$$\begin{array}{rcll}
\nabla_t(\gamma_t(b)) & = & \nabla_t(\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \text{by definition of } \gamma_t \\
& = & (\gamma_{t_1}(\alpha_{t_1}(\gamma_{t_1}(b))), \ldots, \gamma_{t_m}(\alpha_{t_m}(\gamma_{t_m}(b)))) & \text{by definition of } \nabla_t \\
& = & (\gamma_{t_1}(b), \ldots, \gamma_{t_m}(b)) & \text{by Proposition 7.5} \\
& = & \gamma_t(b) & \text{by definition of } \gamma_t
\end{array}$$

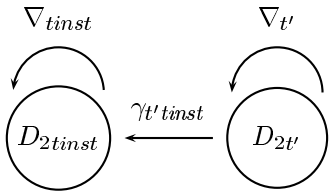


Figure 17: Proposition 8.13: $\nabla_{tinst} \cdot \gamma_{t'\,tinst} = \gamma_{t'\,tinst} \cdot \nabla_{t'}$, where $tinst = t'[\beta := t]$

- $t = t_1 \to t_2$ or $t = \mathit{Process}\ t_1\ t_2$. Let $b \in \mathit{Basic}$. We distinguish two cases: $b = d$ or $b = n$.
  Let us see first the case when $b = d$. On one side we have that:

$$\begin{array}{lcll} \nabla_t(\gamma_t(d)) & = & \nabla_t(\lambda z.\gamma_{t_2}(\alpha_{t_1}(z))) & \text{by definition of } \gamma \\ & = & \lambda z.\left\{\begin{array}{l} \nabla_{t_2}(\gamma_{t_2}(\alpha_{t_1}(\gamma_{t_1}(d)))) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{array}\right. & \text{by definition of } \nabla_t \\ & = & \lambda z.\left\{\begin{array}{l} \gamma_{t_2}(d) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{array}\right. & \text{by i.h. and by Proposition 7.5} \end{array}$$

  On the other side we have that $\gamma_t(d) = \lambda z.\gamma_{t_2}(\alpha_{t_1}(z))$. We have to prove that for each $e \in D_{2t_1}$, $(\nabla_t(\gamma_t(d)))(e) = (\gamma_t(d))(e)$. We distinguish two cases:

  - $e \sqsubseteq \gamma_{t_1}(d)$. In this case
    $$(\nabla_t(\gamma_t(d)))(e) = \gamma_{t_2}(d)$$
    By Lemma 7.6, if $e \sqsubseteq \gamma_{t_1}(d)$ then $\alpha_{t_1}(e) = d$, so
    $$(\gamma_t(d))(e) = \gamma_{t_2}(d)$$
  - Otherwise, by Lemma 7.7, $\alpha_{t_1}(e) = n$ and we have:
    $$\begin{array}{lcl} (\nabla_t(\gamma_t(d)))(e) & = & \gamma_{t_2}(n) \\ & = & \gamma_{t_2}(\alpha_{t_1}(e)) \\ & = & (\gamma_t(d))(e) \end{array}$$

  If $b = n$:

$$\begin{array}{lcll} \nabla_t(\gamma_t(n)) & = & \nabla_t(\lambda z.\gamma_{t_2}(n)) & \text{by definition of } \gamma_t \\ & = & \lambda z.\left\{\begin{array}{l} \nabla_{t_2}(\gamma_{t_2}(n)) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{array}\right. & \text{by definition of } \nabla_t \\ & = & \lambda z.\gamma_{t_2}(n) & \text{by i.h. } \nabla_{t_2} \cdot \gamma_{t_2} = \gamma_{t_2} \\ & = & \gamma_t(n) & \text{by definition of } \gamma_t \end{array}$$

- $t = \forall\beta.t'$. It trivially holds by i.h. as $\nabla_t = \nabla_{t'}$ and $\gamma_t = \gamma_{t'}$.

The two following propositions add some results about the polymorphic functions. The first one tells us that we obtain the same result if we adapt an abstract value belonging to $D_{2t'}$ to the type *tinst* and then widen the result that if we first widen it and then adapt the result. In Figure 17 a diagram is shown.

Proposition 8.13 *Given two types t, t' and a type variable β, the following holds:*

$$\nabla_{tinst} \cdot \gamma_{t'\,tinst} = \gamma_{t'\,tinst} \cdot \nabla_{t'}$$

where $tinst = t'[\beta := t]$.

Proof (Proposition 8.13) This can be proved by structural induction on t':

- $t' = K$ or $t' = T\ t_1 \ldots t_m$ or $t' = \beta'$ $(\neq \beta)$. In these cases $D_{2\,tinst} = D_{2t'} = \mathit{Basic}$ and $\nabla_{tinst} = \gamma_{t'\,tinst} = \nabla_{t'} = id_{\mathit{Basic}}$, so it trivially holds.

- $t' = (t_1, \ldots, t_m)$. In this case $tinst = (tinst_1, \ldots, tinst_m)$ where $tinst_i = t_i[\beta := t]$, for $i = 1, \ldots, m$. By i.h. we have that for each $i = 1, \ldots, m$, $\nabla_{tinst_i} \cdot \gamma_{t_i tinst_i} = \gamma_{t_i tinst_i} \cdot \nabla_{t_i}$. Let $e_i \in D_{2t_i}$ where $i = 1, \ldots, m$. Then:

$$
\begin{array}{lll}
\nabla_{tinst}(\gamma_{t' tinst}(e_1, \ldots, e_m)) & & \\
= & \nabla_{tinst}(\gamma_{t_1 tinst_1}(e_1), \ldots, \gamma_{t_m tinst_m}(e_m)) & \text{by definition of } \gamma_{t' tinst} \\
= & (\gamma_{tinst_1}(\alpha_{tinst_1}(\gamma_{t_1 tinst_1}(e_1))), \ldots, \gamma_{tinst_m}(\alpha_{tinst_m}(\gamma_{t_m tinst_m}(e_m)))) & \text{by definition of } \nabla_t \\
= & (\gamma_{tinst_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{tinst_m}(\alpha_{t_m}(e_m))) & \text{by Proposition 7.14} \\
= & (\gamma_{t_1 tinst_1}(\gamma_{t_1}(\alpha_{t_1}(e_1))), \ldots, \gamma_{t_m tinst_m}(\gamma_{t_m}(\alpha_{t_m}(e_m)))) & \text{by Proposition 7.13} \\
= & \gamma_{t' tinst}(\gamma_{t_1}(\alpha_{t_1}(e_1)), \ldots, \gamma_{t_m}(\alpha_{t_m}(e_1))) & \text{by definition of } \gamma_{t' tinst} \\
= & \gamma_{t' tinst}(\nabla_{t'}(e_1, \ldots, e_m)) & \text{by definition of } \nabla_t
\end{array}
$$

- $t' = t_1 \to t_2$ or $t' = Process\ t_1\ t_2$. In this case $tinst = tinst_1 \to tinst_2$, respectively $tinst = Process\ tinst_1\ tinst_2$, where $tinst_i = t_i[\beta := t]$, for $i = 1, 2$. Let $f \in [D_{2t_1} \to D_{2t_2}]$. On one side we have:

$$
\begin{array}{lll}
\nabla_{tinst}(\gamma_{t' tinst}(f)) & & \\
= & \nabla_{tinst}(\lambda z.\gamma_{t_2 tinst_2}(f(\alpha_{tinst_1 t_1}(z)))) & \text{by definition of } \gamma_{t' tinst} \\
= & \lambda z. \begin{cases} \nabla_{tinst_2}(\gamma_{t_2 tinst_2}(f(\alpha_{tinst_1 t_1}(\gamma_{tinst_1}(d))))) \text{ if } z \sqsubseteq \gamma_{tinst_1}(d) \\ \gamma_{tinst_2}(n) \text{ otherwise} \end{cases} & \text{by definition of } \nabla_t
\end{array}
$$

On the other side:

$$
\begin{array}{lll}
\gamma_{t' tinst}(\nabla_{t'}(f)) & = & \gamma_{t' tinst}\left(\lambda z. \begin{cases} \nabla_{t_2}(f(\gamma_{t_1}(d))) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise} \end{cases}\right) \quad \text{by definition of } \nabla_t \\
& = & \lambda z. \begin{cases} \gamma_{t_2 tinst_2}(\nabla_{t_2}(f(\gamma_{t_1}(d)))) \text{ if } \alpha_{tinst_1 t_1}(z) \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2 tinst_2}(\gamma_{t_2}(n)) \text{ otherwise} \end{cases} \quad \text{by definition of } \gamma_{t' tinst}
\end{array}
$$

We want to prove the equality of these two functions. Let $e \in D_{2tinst_1}$. We distinguish two cases::

- $e \sqsubseteq \gamma_{tinst_1}(d)$. Then by monotonicity of $\alpha_{tinst\, t'}$ (Proposition 7.11), $\alpha_{tinst_1 t_1}(e) \sqsubseteq \alpha_{tinst_1 t_1}(\gamma_{tinst_1}(d))$ and by Proposition 7.14 $\alpha_{tinst_1 t_1}(\gamma_{tinst_1}(d)) = \gamma_{t_1}(d)$, so

$$\alpha_{tinst_1 t_1}(e) \sqsubseteq \gamma_{t_1}(d)$$

  So, in this case:

$$(\nabla_{tinst}(\gamma_{t' tinst}(f)))(e) = \nabla_{tinst_2}(\gamma_{t_2 tinst_2}(f(\alpha_{tinst_1 t_1}(\gamma_{tinst_1}(d)))))$$

  and

$$(\gamma_{t' tinst}(\nabla_{t'}(f)))(e) = \gamma_{t_2 tinst_2}(\nabla_{t_2}(f(\gamma_{t_1}(d))))$$

  So:

$$
\begin{array}{lll}
(\nabla_{tinst}(\gamma_{t' tinst}(f)))(e) & = & \nabla_{tinst_2}(\gamma_{t_2 tinst_2}(f(\alpha_{tinst_1 t_1}(\gamma_{tinst_1}(d))))) \\
& = & \gamma_{t_2 tinst_2}(\nabla_{t_2}(f(\alpha_{tinst_1 t_1}(\gamma_{tinst_1}(d))))) \quad \text{by i.h.} \\
& = & \gamma_{t_2 tinst_2}(\nabla_{t_2}(f(\gamma_{t_1}(d)))) \quad \text{by Proposition 7.14} \\
& = & \gamma_{t' tinst}(\nabla_{t'}(f)))(e)
\end{array}
$$

- $e \not\sqsubseteq \gamma_{tinst_1}(d)$. In this case by Lemma 7.7 $\alpha_{tinst_1}(e) = n$. Additionally, $(\nabla_{tinst}(\gamma_{t' tinst}(f)))(e) = \gamma_{tinst_2}(n)$. And also we have that $\alpha_{tinst_1 t_1}(e) \not\sqsubseteq \gamma_{t_1}(d)$. This is true because otherwise by Proposition 7.13 $\alpha_{tinst_1}(e) = \alpha_{t_1}(\alpha_{tinst_1 t_1}(e))$, we would have that $\alpha_{tinst_1}(e) \sqsubseteq \alpha_{t_1}(\gamma_{t_1}(d))$ by monotonicity of α_t (Proposition 7.3), what would mean that $\alpha_{tinst_1}(e) = d$ (by Proposition 7.5), which is false as we have just seen that it is equal to n. So $(\gamma_{t' tinst}(\nabla_{t'}(f)))(e) = \gamma_{t_2 tinst_2}(\gamma_{t_2}(n))$. And:

$$
\begin{array}{lll}
(\nabla_{tinst}(\gamma_{t' tinst}(f)))(e) & = & \gamma_{tinst_2}(n) \\
& = & \gamma_{t_2 tinst_2}(\gamma_{t_2}(n)) \quad \text{by Proposition 7.13} \\
& = & (\gamma_{t' tinst}(\nabla_{t'}(f)))(e)
\end{array}
$$

- $t' = \beta$. In this case:

$$\begin{array}{lcll} \nabla_{tinst} \cdot \gamma_{t'\,tinst} & = & \nabla_t \cdot \gamma_t & \text{by definition of } \nabla_t \text{ and } \gamma_t \\ & = & \gamma_t & \text{by Proposition 8.12} \\ & = & \gamma_{t'\,tinst} \cdot \nabla_{t'} & \text{by definition of } \gamma_{t'\,tinst} \text{ and } \nabla_t \end{array}$$

- $t' = \forall\beta'.t_1$. It holds directly by i.h., as $\nabla_{tinst} = \nabla_{tinst_1}$, $\gamma_{t'\,tinst} = \gamma_{t_1\,tinst_1}$ and $\nabla_{t'} = \nabla_{t_1}$.

The following proposition tells us basically that the adaptation of an abstract value in $D_{1t'}$ to obtain an aproximation to the abstract value of an instance *tinst*, is basically equal to the adaptation made with $\gamma_{t'\,tinst}$ in the domains of the second analysis.

Proposition 8.14 *Given two types t, t' and a type variable β, the following holds:*

$$\forall a \in D_{1t'}.\ a_{tinst} = \delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(a)))$$

where $tinst = t'[\beta := t]$.

Proof (Proposition 8.14) It can be proved by structural induction on t':

- $t' = K$ or $t' = T\ t_1 \ldots t_m$ or $t' = \beta'$ $(\neq \beta)$. In this case $D_{1\,tinst} = D_{1t'} = Basic$. Let $b \in Basic$. Then $b_{tinst} = b$. On the other side $\delta_{tinst} = \gamma_{t'\,tinst} = \eta_{t'} = id_{Basic}$, so it holds trivially.
- $t' = (t_1, \ldots, t_m)$. In this case $tinst = (tinst_1, \ldots, tinst_m)$ where $tinst_i = t_i[\beta := t]$, for $i = 1, \ldots, m$. Let $b_i \in Basic$. Then

$$\begin{array}{lcll} & & \delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(b_1, \ldots, b_m))) & \\ & = & \delta_{tinst}(\gamma_{t'\,tinst}(\gamma_{t_1}(b_1), \ldots, \gamma_{t_m}(b_m))) & \text{by definition of } \eta_t \\ & = & \delta_{tinst}(\gamma_{t_1\,tinst_1}(\gamma_{t_1}(b_1)), \ldots, \gamma_{t_1\,tinst_1}(\gamma_{t_m}(b_m))) & \text{by definition of } \gamma_{t'\,tinst} \\ & = & \delta_{tinst}(\gamma_{tinst_1}(b_1), \ldots, \gamma_{tinst_m}(b_m)) & \text{by Proposition 7.13} \\ & = & (\alpha_{tinst_1}(\gamma_{tinst_1}(b_1)), \ldots, \alpha_{tinst_m}(\gamma_{tinst_m}(b_m))) & \text{by definition of } \delta_t \\ & = & (b_1, \ldots, b_m) & \text{by Proposition 7.5} \\ & = & (b_1, \ldots, b_m)_{tinst} & \text{by def. of the adaptation} \end{array}$$

- $t' = t_1 \to t_2$ or $t' = Process\ t_1\ t_2$. In this case $tinst = tinst_1 \to tinst_2$, respectively $tinst = Process\ tinst_1\ tinst_2$, where $tinst_i = t_i[\beta := t]$, for $i = 1, 2$. Let $a \in D_{1t'} = D_{1t_2}$. Now we have:

$$\begin{array}{lcll} & & \delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(a))) & \\ & = & \delta_{tinst}\left(\gamma_{t'\,tinst}\left(\lambda z.\left\{\begin{array}{l}\eta_{t_2}(a) \text{ if } z \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2}(n) \text{ otherwise}\end{array}\right.\right)\right) & \text{by definition of } \eta_t \\ & = & \delta_{tinst}\left(\lambda z.\left\{\begin{array}{l}\gamma_{t_2\,tinst_2}(\eta_{t_2}(a)) \text{ if } \alpha_{tinst_1\,t_1}(z) \sqsubseteq \gamma_{t_1}(d) \\ \gamma_{t_2\,tinst_2}(\gamma_{t_2}(n)) \text{ otherwise}\end{array}\right.\right) & \text{by definition of } \gamma_{t'\,tinst} \\ & = & \delta_{tinst_2}(\gamma_{t_2\,tinst_2}(\eta_{t_2}(a))) & \text{by def. of } \delta_t \text{ and Proposition 7.14} \\ & = & a_{tinst_2} & \text{by i.h.} \\ & = & a_{tinst} & \text{by definition of adaptation} \end{array}$$

- $t' = \beta$. Now $D_{1t'} = Basic$ and $D_{1\,tinst} = D_{1t}$. Let $b \in Basic$. Then:

$$\begin{array}{lcll} \delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(b))) & = & \delta_t(\gamma_t(b)) & \text{by def. of } \delta_t, \gamma_{t'\,tinst} \text{ and } \eta_t \\ & = & b_t & \text{by Proposition 8.9} \\ & = & b_{tinst} & \end{array}$$

- $t' = \forall\beta'.t_1$. It directly holds by i.h., as $\delta_{tinst} = \delta_{tinst_1}$, $\gamma_{t'\,tinst} = \gamma_{t_1\,tinst_1}$, $\eta_{tinst} = \eta_{tinst_1}$ and $b_{tinst} = b_{tinst_1}$.

A very important and useful property to prove the correctness is the semihomomorphic property of δ_t with respect to the application of a function. But as there are no functional domains in the first analysis, the property holds with respect to the pseudoapplication we use in such domains, that is, the way in which application of a function is interpreted: $f(x) = (\widehat{\sqcup} x) \sqcup f$.

Proposition 8.15 *Let $f \in [D_{2t_1} \to D_{2t_2}]$, $e \in D_{2t_1}$. The following holds:*

$$\delta_{t_2}(f(e)) \sqsubseteq (\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_1 \to t_2}(f)$$

Proof (Proposition 8.15) Let $f \in [D_{2t_1} \to D_{2t_2}]$, $e \in D_{2t_1}$. By definition $\delta_{t_1 \to t_2}(f) = \delta_{t_2}(f(\gamma_{t_1}(d)))$, so we have to prove that $\delta_{t_2}(f(e)) \sqsubseteq (\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_2}(f(\gamma_{t_1}(d)))$. We distinguish two cases::

- $e \sqsubseteq \gamma_{t_1}(d)$. As f is monotone, $f(e) \sqsubseteq f(\gamma_{t_1}(d)$. By Proposition 8.3, δ_{t_2} is monotone, so $\delta_{t_2}(f(e)) \sqsubseteq \delta_{t_2}(f(\gamma_{t_1}(d)))$, which trivially is less than or equal to $(\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_2}(f(\gamma_{t_1}(d)))$.
- $e \not\sqsubseteq \gamma_{t_1}(d)$. In this case, by Lemma 7.7 $\alpha_{t_1}(e) = n$, so by Proposition 8.4, $\widehat{\sqcup}(\delta_{t_1}(e)) = n$.

  It is obvious that $b \in \mathit{Basic}$, $n \sqcup b \sqsupseteq b'$ for any $b' \in \mathit{Basic}$. And if $a \in \mathit{Basic}^m$, then $n \sqcup a \sqsupseteq a'$, for any $a' \in \mathit{Basic}^m$. That is, the lub of any value and n takes us to the top of the corresponding domain.

  So
$$(\widehat{\sqcup}\delta_{t_1}(e)) \sqcup \delta_{t_1 \to t_2}(f) = n \sqcup \delta_{t_1 \to t_2}(f) \sqsupseteq \delta_{t_2}(f(e))$$

Let us prove now the Theorem 8.7 of correctness of the first analysis with respect to the second one. Let ρ_1 and ρ_2 such that for each variable $v :: t_v$, $\rho_1(v) \sqsupseteq \delta_{t_v}(\rho_2(v))$.

Proof (Theorem 8.7) It can be proved by structural induction on e.

- $v :: t$. In this case:

$$\begin{array}{lcll} [\![v]\!]_1\ \rho_1 & = & \rho_1(v) & \text{by definition of } [\![\ ]\!]_1 \\ & \sqsupseteq & \delta_t(\rho_2(v)) & \text{by hypothesis over the environments} \\ & = & \delta_t([\![v]\!]_2\ \rho_2) & \text{by definition of } [\![\ ]\!]_2 \end{array}$$

- $k :: K$.

$$\begin{array}{lcll} [\![k]\!]_1\ \rho_1 & = & d & \text{by definition of } [\![\ ]\!]_1 \\ & = & [\![k]\!]_2\ \rho_2 & \text{by definition of } [\![\ ]\!]_2 \\ & = & \delta_K([\![k]\!]_2\ \rho_2) & \text{by definition of } \delta_t \end{array}$$

- $C\ x_1 \ldots x_m :: T\ t'_1 \ldots t'_k$, where $x_i :: t_i$. By i.h. we know that for each $i = 1, \ldots, m$, the following holds:

$$[\![x_i]\!]_1\ \rho_1 \sqsupseteq \delta_{t_i}([\![x_i]\!]_2\ \rho_2)$$

  As the operator $\widehat{\sqcup}$ is monotone, this means that for each $i = 1, \ldots, m$ the following holds

$$\widehat{\sqcup}([\![x_i]\!]_1\ \rho_1) \sqsupseteq \widehat{\sqcup}(\delta_{t_i}([\![x_i]\!]_2\ \rho_2))$$

  By Proposition 8.4 we have that $\widehat{\sqcup}(\delta_{t_i}([\![x_i]\!]_2\ \rho_2)) = \alpha_{t_i}([\![x_i]\!]_2\ \rho_2)$, so

$$\widehat{\sqcup}([\![x_i]\!]_1\ \rho_1) \sqsupseteq \alpha_{t_i}([\![x_i]\!]_2\ \rho_2)\ (*)$$

  Then:

$$\begin{array}{lcll} [\![C\ x_1 \ldots x_m]\!]_1\ \rho_1 & = & \bigsqcup_i(\widehat{\sqcup}([\![x_i]\!]_1\ \rho_1)) & \text{by definition of } [\![\ ]\!]_1 \\ & \sqsupseteq & \bigsqcup_i (\alpha_{t_i}([\![x_i]\!]_2\ \rho_2)) & \text{by } (*) \\ & = & [\![C\ x_1 \ldots x_m]\!]_2\ \rho_2 & \text{by definition of } [\![\ ]\!]_2 \end{array}$$

- $(x_1, \ldots, x_m) :: (t_1, \ldots, t_m)$. Let us call t to $(t_1, \ldots, t_m)$. In this case we have on one side that:

$$\begin{array}{rcll} [\![(x_1, \ldots, x_m)]\!]_1\ \rho_1 & = & (\widehat{\sqcup}([\![x_1]\!]_1\ \rho_1), \ldots, \widehat{\sqcup}([\![x_m]\!]_1\ \rho_1)) & \text{by definition of } [\![\ ]\!]_1 \\ & \sqsupseteq & (\widehat{\sqcup}(\delta_{t_1}([\![x_1]\!]_2\ \rho_2)), \ldots, \widehat{\sqcup}(\delta_{t_m}([\![x_m]\!]_2\ \rho_2))) & \text{by i.h. and monotonicity of } \widehat{\sqcup} \\ & = & (\alpha_{t_1}([\![x_1]\!]_2\ \rho_2), \ldots, \alpha_{t_m}([\![x_m]\!]_2\ \rho_2)) & \text{by Proposition 8.4} \end{array}$$

  and on the other side:

$$\begin{array}{rcll} \delta_t([\![(x_1, \ldots, x_m)]\!]_2\ \rho_2) & = & \delta_t([\![x_1]\!]_2\ \rho_2, \ldots, [\![x_m]\!]_2\ \rho_2) & \text{by definition of } [\![\ ]\!]_2 \\ & = & (\alpha_{t_1}([\![x_1]\!]_2\ \rho_2), \ldots, \alpha_{t_m}([\![x_m]\!]_2\ \rho_2)) & \text{by definition of } \delta_t \end{array}$$

  So

$$[\![(x_1, \ldots, x_m)]\!]_1\ \rho_1 \sqsupseteq \delta_t([\![(x_1, \ldots, x_m)]\!]_2\ \rho_2)$$

- $\lambda v.e :: t_1 \to t_2$. The proof is exactly the same if the expression is *process* $v \to e$ :: *Process* $t_1\ t_2$.
  In this case we have on one side that:

$$[\![\lambda v.e]\!]_1\ \rho_1 = [\![e]\!]_1\ \rho_1\ [v \to d_{t_1}]$$

  Let us call ρ_1' to $\rho_1\ [v \to d_{t_1}]$.
  On the other side we have that:

$$[\![\lambda v.e]\!]_2\ \rho_2 = \lambda z.[\![e]\!]_2\ \rho_2\ [v \to z]$$

  so that:

$$\begin{array}{rcll} \delta_t([\![\lambda v.e]\!]_2\ \rho_2) & = & \delta_t(\lambda z.[\![e]\!]_2\ \rho_2\ [v \to z]) & \\ & = & \delta_{t_2}([\![e]\!]_2\ \rho_2\ [v \to \gamma_{t_1}(d)]) & \text{by definition of } \delta_t \end{array}$$

  Let us call ρ_2' to $\rho_2\ [v \to \gamma_{t_1}(d)]$.
  If we proved that for each variable $y :: t_y$, $\rho_1'(y) \sqsupseteq \delta_{t_y}(\rho_2'(y))$, then by i.h. we would have that

$$[\![e]\!]_1\ \rho_1' \sqsupseteq \delta_{t_2}([\![e]\!]_2\ \rho_2')$$

  that is what we wanted to prove.
  So let us prove this. Let a variable $y :: t_y$.

  - If $y \neq v$, it trivially holds by hypothesis over the environments ρ_1 and ρ_2.
  - If $y = v$, then we have to prove that $d_{t_1} \sqsupseteq \delta_{t_1}(\gamma_{t_1}(d))$, which holds trivially by Proposition 8.9.

- $e\ x :: t_2$, where $e :: t_1 \to t_2$ and $x :: t_1$. Let us call t to $t_1 \to t_2$. The proof is similar in case the expression is $v \# x$ where $v ::$ *Process* $t_1\ t_2$. By i.h. we know that

$$[\![x]\!]_1\ \rho_1 \sqsupseteq \delta_{t_1}([\![x]\!]_2\ \rho_2)$$

  and

$$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_t([\![e]\!]_2\ \rho_2)$$

  So we have that:

$$\begin{array}{rcll} [\![e\ x]\!]_1\ \rho_1 & = & (\widehat{\sqcup}([\![x]\!]_1\ \rho_1)) \sqcup ([\![e]\!]_1\ \rho_1) & \text{by definition of } [\![\ ]\!]_1 \\ & \sqsupseteq & (\widehat{\sqcup}(\delta_{t_1}([\![x]\!]_2\ \rho_2))) \sqcup \delta_t([\![e]\!]_2\ \rho_2) & \text{by i.h. and monotonicity of } \sqcup \text{ and } \widehat{\sqcup} \\ & \sqsupseteq & \delta_{t_2}(([\![e]\!]_2\ \rho_2)([\![x]\!]_2\ \rho_2)) & \text{by Proposition 8.15} \\ & = & \delta_{t_2}([\![e\ x]\!]_2\ \rho_2) & \text{by definition of } [\![\ ]\!]_2 \end{array}$$

- **let** $v = e$ **in** $e' :: t$, where v and e have type t_e, and $e' :: t$. In this case, we have on one side that:

$$[\![\textbf{let } v = e \textbf{ in } e']\!]_1\ \rho_1 = [\![e']\!]_1\ \rho_1\ [v \to [\![e]\!]_1\ \rho_1]$$

  Let us call ρ_1' to $\rho_1\ [v \to [\![e]\!]_1\ \rho_1]$.

On the other side we have that

$$[\![\textbf{let } v = e \textbf{ in } e']\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ [v \to [\![e]\!]_2\ \rho_2]$$

Let us call ρ_2' to $\rho_2\ [v \to [\![e]\!]_1\ \rho_2]$.

If we proved that for each variable $y :: t_y$, $\rho_1'(y) \sqsupseteq \delta_{t_y}(\rho_2'(y))$, then by i.h. we would have that

$$[\![e']\!]_1\ \rho_1' \sqsupseteq \delta_t([\![e']\!]_2\ \rho_2')$$

which is what we want to prove.

Let us then prove this. Let $y :: t_y$.

- If $y \neq v$, it trivially holds by hypothesis over the environments ρ_1 and ρ_2.
- If $y = v$, then we have to prove that

  $$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_2\ \rho_2)$$

  which holds by i.h..

- **case** e **of** $(v_1, \ldots, v_m) \to e' :: t$, where $e :: t_e = (t_1, \ldots, t_m)$, $v_i :: t_i$ and $e' :: t$. In this case, we have on one side:

  $$[\![\textbf{case } e \textbf{ of } (v_1, \ldots, v_m) \to e]\!]_1\ \rho_1 = [\![e']\!]_1\ \rho_1\ \overline{[v_i \to (\pi_i([\![e]\!]_1\ \rho_1))_{t_i}]}$$

  And on the other side that

  $$[\![\textbf{case } e \textbf{ of } (v_1, \ldots, v_m) \to e]\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ \overline{[v_i \to \pi_i([\![e]\!]_2\ \rho_2)]}$$

  If we proved that for each $i = 1, \ldots, m$ the following holds

  $$(\pi_i([\![e]\!]_1\ \rho_1))_{t_i} \sqsupseteq \delta_{t_i}(\pi_i([\![e]\!]_2\ \rho_2))$$

  then by i.h. we would have that

  $$[\![e']\!]_1\ \rho_1\ \overline{[v_i \to (\pi_i([\![e]\!]_1\ \rho_1))_{t_i}]} \sqsupseteq \delta_t([\![e']\!]_2\ \rho_2\ \overline{[v_i \to \pi_i([\![e]\!]_2\ \rho_2)]})$$

  which is what we want to prove.

  So let us prove this. By i.h. we know that the following holds:

  $$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_2\ \rho_2)$$

  This can be rewritten in the following way:

  $$\begin{array}{lcll} (\pi_1([\![e]\!]_1\ \rho_1), \ldots, \pi_m([\![e]\!]_1\ \rho_1)) & = & [\![e]\!]_1\ \rho_1 & \text{as it is of tuple type} \\ & \sqsupseteq & \delta_{t_e}([\![e]\!]_2\ \rho_2) & \text{by i.h.} \\ & = & \delta_{t_e}(\pi_1([\![e]\!]_2\ \rho_2), \ldots, \pi_m([\![e]\!]_2\ \rho_2)) & \text{as it is of tuple type} \\ & = & (\alpha_{t_1}(\pi_1([\![e]\!]_2\ \rho_2)), \ldots, \alpha_{t_m}(\pi_m([\![e]\!]_2\ \rho_2))) & \text{by definition of } \delta_t \end{array}$$

  That is, for each $i = 1, \ldots, m$ the following holds:

  $$\pi_i([\![e]\!]_1\ \rho_1) \sqsupseteq \alpha_{t_i}(\pi_i([\![e]\!]_2\ \rho_2))\ (*)$$

  So:

  $$\begin{array}{lcll} (\pi_i([\![e]\!]_1\ \rho_1))_{t_i} & = & \delta_{t_i}(\gamma_{t_i}(\pi_i([\![e]\!]_1\ \rho_1))) & \text{by Proposition 8.9} \\ & \sqsupseteq & \delta_{t_i}(\gamma_{t_i}(\alpha_{t_i}(\pi_i([\![e]\!]_2\ \rho_2)))) & \text{by (*) and monotonicity of } \delta_t \text{ and } \gamma_t \\ & \sqsupseteq & \delta_{t_i}(\pi_i([\![e]\!]_2\ \rho_2)) & \text{by Proposition 7.5 and monotonicity of } \delta_t \end{array}$$

  so we have what we wanted to prove.

- **case** e **of** $\overline{C_i\ \overline{v_{ij}} \to e_i} :: t$, where $e :: t_e = T\ t'_1 \ldots t'_k$, $v_{ij} :: t_{ij}$ and $e_i :: t$. Let us call e' to **case** e **of** $\overline{C_i\ \overline{v_{ij}} \to e_i}$. In this case we have on one side that:

$$[\![e']\!]_1\ \rho_1 = \begin{cases} n_t \text{ if } [\![e]\!]_1\ \rho_1 = n \\ \bigsqcup_i [\![e_i]\!]_1\ \rho_{1i} \text{ otherwise} \end{cases}$$

where $\rho_{1i} = \rho_1\ \overline{[v_{ij} \to d_{t_{ij}}]}, v_{ij} :: t_{ij}, e_i :: t$

and on the other side:

$$[\![e']\!]_2\ \rho_2 = \begin{cases} \gamma_t(n) \text{ if } [\![e]\!]_2\ \rho_2 = n \\ \bigsqcup_i [\![e_i]\!]_2\ \rho_{2i} \text{ otherwise} \end{cases}$$

where $\rho_{2i} = \rho_2\ \overline{[v_{ij} \to \gamma_{t_{ij}}(d)]}, v_{ij} :: t_{ij}, e_i :: t$

By i.h. we know that

$$[\![e]\!]_1\ \rho_1 \sqsupseteq \delta_{t_e}([\![e]\!]_2\ \rho_2)$$

that is, knowing that $\delta_{T\ t'_1,\ldots,t'_k} = id_{Basic}$:

$$[\![e]\!]_1\ \rho_1 \sqsupseteq [\![e]\!]_2\ \rho_2$$

We distinguish three cases:

  - $[\![e]\!]_1\ \rho_1 = [\![e]\!]_2\ \rho_2 = n$. In this case:

$$\begin{array}{lcll} [\![e']\!]_1\ \rho_1 & = & n_t & \text{by definition of } [\![\ ]\!]_1 \\ & = & \delta_t(\gamma_t(n)) & \text{by Proposition 8.9} \\ & = & \delta_t([\![e']\!]_2\ \rho_2) & \text{by definition of } [\![\ ]\!]_2 \end{array}$$

  - $[\![e]\!]_1\ \rho_1 = [\![e]\!]_2\ \rho_2 = d$. In this case:

$$[\![e']\!]_1\ \rho_1 = \bigsqcup_i [\![e_i]\!]_1\ \rho_{1i}$$

where $\rho_{1i} = \rho_1\ \overline{[v_{ij} \to d_{t_{ij}}]}$ and

$$[\![e']\!]_2\ \rho_2 = \bigsqcup_i [\![e_i]\!]_2\ \rho_{2i}$$

where $\rho_{2i} = \rho_2\ \overline{[v_{ij} \to \gamma_{t_{ij}}(d)]}$.
If we proved for each $i = 1, \ldots, m$, that for each variable $y :: t_y$, $\rho_{1i}(y) \sqsupseteq \delta_{t_y}(\rho_{2i}(y))$, then by i.h. we would have that for each i the following holds

$$[\![e_i]\!]_1\ \rho_{1i} \sqsupseteq \delta_t([\![e_i]\!]_2\ \rho_{2i})$$

so, by continuity of δ_t:

$$\bigsqcup_i [\![e_i]\!]_1\ \rho_{1i} \sqsupseteq \bigsqcup_i \delta_t([\![e_i]\!]_2\ \rho_{2i}) = \delta_t(\bigsqcup_i [\![e_i]\!]_2\ \rho_{2i})$$

which is what we wanted to prove.
So, let us prove now that for each variable $y :: t_y$, $\rho_{1i}(y) \sqsupseteq \delta_{t_y}(\rho_{2i}(y))$. If y is not any of the v_{ij}, this is trivially true by hypothesis on ρ_1 and ρ_2. If y is one of the v_{ij}, then it is also true by Proposition 8.9.

  - $[\![e]\!]_1\ \rho_1 = n \sqsupset [\![e]\!]_2\ \rho_2 = d$. In this case $[\![e']\!]_1\ \rho_1 = n_t$, and $[\![e']\!]_2\ \rho_2 = \bigsqcup_i [\![e_i]\!]_2\ \rho_{2i}$. Clearly, $n_t \sqsupseteq \bigsqcup_i [\![e_i]\!]_2\ \rho_{2i}$, as n_t is the top of D_{1t}.

- **let rec** $\overline{\{v_i = e_i\}}$ **in** $e' :: t$, where $e' :: t$, and each v_i and e_i have type t_i. Let us call e to **let rec** $\overline{\{v_i = e_i\}}$ **in** e'. In this case:

$$[\![e]\!]_1\ \rho_1 = [\![e']\!]\ (\bigsqcup_{n\in\mathbb{N}} (\lambda\rho'_1.\rho_1\ \overline{[v_i \to [\![e_i]\!]_1\ \rho'_1]})^n(\rho_{01}))$$

where ρ_{01} is the initial environment where each variable $y :: t_y$ has d_{t_y} as abstract value (that is, the infimum of the corresponding domain). Let us call F to the function between environments $\lambda\rho'_1.\rho_1\ \overline{[v_i \to [\![e_i]\!]_1\ \rho'_1]}$. Let us call ρ_1^{fix} to $\bigsqcup_{n\in\mathbb{N}} F^n(\rho_{01})$.

On the other side:

$$[\![e]\!]_2\ \rho_2 = [\![e']\!]_2\ (\bigsqcup_{n\in\mathbb{N}} (\lambda\rho'_2.\rho_2\ \overline{[v_i \to [\![e_i]\!]_2\ \rho'_2]})^n(\rho_{02}))$$

where ρ_{02} is the initial environment where each variable $y :: t_y$ has $\bot_{2t_y}$ as abstract value (that is, the infimum of the corresponding domain). Let us call G to the function between environments $\lambda\rho'_2.\rho_2\ \overline{[v_i \to [\![e_i]\!]_2\ \rho'_2]}$. Let us call ρ_2^{fix} to $\bigsqcup_{n\in\mathbb{N}} G^n(\rho_{02})$.

If we proved that for each variable $y :: t_y$, $\rho_1^{fix}(y) \sqsupseteq \delta_{t_y}(\rho_2^{fix}(y))$, then by i.h. we would have that

$$[\![e']\!]_1\ \rho_1^{fix} \sqsupseteq \delta_t([\![e']\!]_2\ \rho_2^{fix})$$

which is what we wanted to prove.

Let us see that for each $n \geq 0$, the following holds

$$\forall y :: t_y.(F^n(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((G^n(\rho_{02}))(y))$$

If this were true then

$$\forall y :: t_y.(\bigsqcup_{n\in\mathbb{N}} F^n(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((\bigsqcup_{n\in\mathbb{N}} G^n(\rho_{02}))(y))$$

by continuity of δ_t, and this is what we want to prove. It can be proved by induction on n:

- $n = 0$. This is a trivial case as $F^0(\rho_{01}) = \rho_{01}$, $G^0(\rho_{02}) = \rho_{02}$ and $d_t = \delta_t(\bot_{2t})$, by Proposition 8.10.
- $n = m + 1$. Then

$$\begin{array}{rcll} F^{m+1}(\rho_{01}) & = & F(F^m(\rho_{01})) & \\ & = & \rho_1\ \overline{[v_i \to [\![e_i]\!]_1\ (F^m(\rho_{01}))]} & \text{by definition of } F \end{array}$$

and

$$\begin{array}{rcll} G^{m+1}(\rho_{02}) & = & G(G^m(\rho_{02})) & \\ & = & \rho_2\ \overline{[v_i \to [\![e_i]\!]_2\ (G^m(\rho_{02}))]} & \text{by definition of } G \end{array}$$

Let $y :: t_y$. We want to prove that $(F^{m+1}(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((G^{m+1}(\rho_{02}))(y))$. We distinguish two cases. If y is not any of the v_i, then it holds by the hypothesis over the environments ρ_1 and ρ_2. If it is one of the v_i, then we have to prove that

$$[\![e_i]\!]_1\ (F^m(\rho_{01})) \sqsupseteq \delta_{t_i}([\![e_i]\!]_2\ (G^m(\rho_{02}))) \ (*)$$

By i.h. (internal on n)

$$\forall y :: t_y.\ (F^m(\rho_{01}))(y) \sqsupseteq \delta_{t_y}((G^m(\rho_{02}))(y))$$

we obtain (*) by i.h. (external on e).

- $\Lambda\beta.e :: \forall\beta.t$, where $e :: t$. In this case:

$$\begin{array}{rcll} [\![\Lambda\beta.e]\!]_1\ \rho_1 & = & [\![e]\!]_1\ \rho_1 & \text{by definition of } [\![\ ]\!]_1 \\ & \sqsupseteq & \delta_t([\![e]\!]_2\ \rho_2) & \text{by i.h.} \\ & = & \delta_{\forall\beta.t}([\![\Lambda\beta.e]\!]_2\ \rho_2) & \text{by definition of } [\![\ ]\!]_2 \text{ and } \delta_t \end{array}$$

- $e\ t :: tinst$ where $e :: \forall\beta.t'$ and $tinst = t'[\beta := t]$. In this case

$$\begin{array}{lcll}
[\![e\ t]\!]_1\ \rho_1 & = & ([\![e]\!]_1\ \rho_1)_{tinst} & \text{by definition of } [\![\ ]\!]_1 \\
& = & \delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}([\![e]\!]_1\ \rho_1))) & \text{by Proposition 8.14} \\
& \sqsupseteq & \delta_{tinst}(\gamma_{t'\,tinst}(\eta_{t'}(\delta_{t'}([\![e]\!]_2\ \rho_2)))) & \text{by i.h. and monotonicity of } \delta_t,\ \gamma_{t'\,tinst} \text{ and } \eta_t \\
& = & \delta_{tinst}(\gamma_{t'\,tinst}(\nabla_{t'}([\![e]\!]_2\ \rho_2))) & \text{by definition of } \nabla_t \\
& = & \delta_{tinst}(\nabla_{tinst}(\gamma_{t'\,tinst}([\![e]\!]_2\ \rho_2))) & \text{by Proposition 8.13} \\
& = & \delta_{tinst}(\gamma_{t'\,tinst}([\![e]\!]_2\ \rho_2)) & \text{by definition of } \nabla_t \text{ and Proposition 8.5} \\
& = & \delta_{tinst}([\![e\ t]\!]_2\ \rho_2) & \text{by definition of } [\![\ ]\!]_2
\end{array}$$

To prove the corollary of this theorem we need the following lemma.

Lemma 8.16 *For each type t and $b \in$ Basic the following holds:*

$$\eta_t(b_t) = \gamma_t(b)$$

Proof (Lemma 8.16) By Proposition 8.9 we know that $b_t = \delta_t(\gamma_t(b))$. So:

$$\begin{array}{lcll}
\eta_t(b_t) & = & \eta_t(\delta_t(\gamma_t(b))) & \\
& = & \nabla_t(\gamma_t(b)) & \text{by definition of } \nabla_t \\
& = & \gamma_t(b) & \text{by Proposition 8.12}
\end{array}$$

Proof (Corollary 8.8) Let us assume that for each variable $v :: t_v$, $\rho_1(v) \sqsupseteq \delta_{t_v}(\rho_2(v))$. Let $e :: t_e$ such that $[\![e]\!]_1\ \rho_1 = d_{t_e}$. Then:

$$\begin{array}{lcll}
[\![e]\!]_2\ \rho_2 & \sqsubseteq & \nabla_{t_e}([\![e]\!]_2\ \rho_2) & \text{by Proposition 8.5} \\
& \sqsubseteq & \eta_{t_e}([\![e]\!]_1\ \rho_1) & \text{by Theorem 8.7} \\
& = & \eta_{t_e}(d_{t_e}) & \text{by hypothesis} \\
& = & \gamma_{t_e}(d) & \text{by Lemma 8.16}
\end{array}$$

A more usual way of presenting the correctness of an analysis with respect to another is the following proposition, but this is more apropriate when in both analysis (or in one analysis and in the standard semantics) the functions are interpreted as functions. In our first analysis there are not functions, but we can consider as a pseudoapplication the way in which application of a function is interpreted: $f(x) = (\widehat{\sqcup}x) \sqcup f$. However, this proposition does not add anything new to the Theorem 8.7, as in the first analysis the abstract value of the function already tells us all the information we need. It is not necessary to 'pseudoapply' it to d and look at the result.

Proposition 8.17 *Let $f :: t_1 \to t_2$, $e_1 \in D_{1t_1}$ and $e_2 \in D_{2t_2}$. The following holds:*

$$(\widehat{\sqcup}e_1) \sqcup ([\![f]\!]_1\ \rho_1) \sqsubseteq e_2 \Rightarrow \forall e \sqsubseteq \eta_{t_1}(e_1).\ ([\![f]\!]_2\ \rho_2)(e) \sqsubseteq \eta_{t_2}(e_2)$$

Proof (Proposition 8.17) Let $f :: t_1 \to t_2$, $e_1 \in D_{1t_1}$ and $e_2 \in D_{2t_2}$. Let us assume that $(\widehat{\sqcup}e_1) \sqcup ([\![f]\!]_1\ \rho_1) \sqsubseteq e_2$. Let $e \sqsubseteq \eta_{t_1}(e_1)$. Then $\delta_{t_1}(e) \sqsubseteq e_1$ by Proposition 8.5. So:

$$\begin{array}{lcll}
e_2 & \sqsupseteq & (\widehat{\sqcup}e_1) \sqcup ([\![f]\!]_1\ \rho_1) & \\
& \sqsupseteq & (\widehat{\sqcup}e_1) \sqcup (\delta_{t_1 \to t_2}([\![f]\!]_2\ \rho_2)) & \text{by Theorem 8.7} \\
& \sqsupseteq & (\widehat{\sqcup}\delta_{t_1}(e)) \sqcup (\delta_{t_1 \to t_2}([\![f]\!]_2\ \rho_2)) & \\
& \sqsupseteq & \delta_{t_2}(([\![f]\!]_2\ \rho_2)(e)) & \text{by Proposition 8.15}
\end{array}$$

which implies by Proposition 8.5 that

$$([\![f]\!]_2\ \rho_2)(e) \sqsubseteq \eta_{t_2}(e_2)$$

The corollary of this proposition tells us that, given a function of type $t_1 \to t_2$, if its abstract value in the first analysis returns d_{t_2} when 'pseudoapplied' to d_{t_1}, then the abstract value in the second analysis produces a result below $\gamma_{t_2}(d)$ when applied to an argument below $\gamma_{t_1}(d)$.

Corollary 8.18 *Let $f :: t_1 \to t_2$. The following holds:*

$$(\widehat{\sqcup} d_{t_1}) \sqcup (\llbracket f \rrbracket_1\ \rho_1) = d_{t_2} \Rightarrow \forall e \sqsubseteq \gamma_{t_1}(d).\ (\llbracket f \rrbracket_2\ \rho_2)(e) \sqsubseteq \gamma_{t_2}(d))$$

Proof (Corollary 8.18) If $(\widehat{\sqcup} d_{t_1}) \sqcup (\llbracket f \rrbracket_1\ \rho_1) = d_{t_2}$, by Proposition 8.17, for each $e \sqsubseteq \eta_{t_1}(d_{t_1})$ we have that $(\llbracket f \rrbracket_2\ \rho_2)(e) \sqsubseteq \eta_{t_2}(d_{t_2})$. By Lemma 8.16 $\eta_{t_1}(d_{t_1}) = \gamma_{t_1}(d)$ and $\eta_{t_2}(d_{t_2}) = \gamma_{t_2}(d)$, so we have that for each $e \sqsubseteq \gamma_{t_1}(d)$, $(\llbracket f \rrbracket_2\ \rho_2)(e) \sqsubseteq \gamma_{t_2}(d)$.

9 Related and future work

The first analysis presented in this paper has been expressed using first a type annotation system and afterwards an abstract intepretation easily extensible to a more powerful analysis. In recent years typed based analyses have been widely used for several reasons such as their better efficiency and their adequacy when the information being looked for is preserved across transformations. Non-determinism property should not change across the transformations, so it seems natural to attach non-determinism information to the types of the expressions. In [TWM95] a type based analysis is developed to detect values that are accessed at most once. In [WJ99] type polymorphism and user-defined data types are added. The language being analysed is a second order polymorphic λ-calculus extended with some Core constructions, very similar to the one we have used in our analysis. The analysis annotates the types with usage information.

However the abstract interpretation has shown to be a more direct tool to implement a prototype of the analysis. In [BFGJ] C. Baker-Finch, K. Glynn and S. Peyton Jones present their *constructed product result* (CPR) analysis. The analysis pretends to determine which functions can return multiple results in registers, that is, which functions return an explicitly-constructed tuple. It is an abstract interpretation based analysis where the abstract domain corresponding to a function type $t_1 \to t_2$ is not the corresponding functional domain, but it is instead isomorphic to the abstract domain of the result's type t_2. Product types are interpreted as a cartesian product of a basic abstract domain, so nested tuples are not allowed. Our first analysis, expressed as an abstract interpretation, follows the same ideas but for different reasons that have been already explained.

The second analysis is a typical abstract interpretation in the style of [BHA86], where functions are interpreted as abstract functions. There, a strictness analysis is presented where the basic abstract domain is also a two-point domain ($\bot \sqsubseteq \top$). However, the analyses are rather different. As an example, let $f :: (Int \to Int) \to Int$ be a function whose abstract interpretations in the strictness analysis and in the non-determinism analysis are respectively f^s and f^n. To find out if such function is strict in its argument we apply f^s to $\bot_{Int \to Int}$, that is, to $\lambda z.\bot$: If the result is $\bot$, then it is strict in its argument; otherwise it may be non-strict. On the other hand, if we want to know whether it is is deterministic or not, we apply f^n to $\gamma_{Int \to Int}(d)$, that is, to $\lambda z.z$: If the result is less than or equal to $\gamma_{Int}(d)$ (that is, it is equal to d) then it is deterministic; otherwise it may be non-deterministic. For example, $\lambda g.g\ (head(merge\#[[0],[1]]))$ is strict in its argument but it may be non-deterministic, i.e. $f^s\ (\lambda z.\bot) = \bot$ but $f^n\ (\lambda z.z) = n$. Also, the abstract interpretation of primitive operators, constructors and *case* expressions is different in each analysis.

Correctness of the analyses has not been proved, as still there is not a formal semantics for Eden. However a simplified version where some details are abstracted could be used to prove the correctness.

We have already said that the first analysis has linear complexity, while the second one has a exponential one, as functions are involved. However the second analysis is more powerful than the first one. Following the ideas in [PP93] an intermediate analysis could be developed so that it is more powerful than the first one but less expensive than the second one. The idea is to use a probing to obtain a signature for the function. Such signature is easily comparable and represents a widening of the function. This speeds up the fixpoint calculation, as the chain of widened approximations is shorter. The first analysis is in fact a particular case of probing, where all the arguments are set to 'd'. The idea is to probe also the combinations of arguments where 'n' occupies each position. For example, in a function with three integer arguments, the additional probings would be (n,d,d), (d,n,d) and (d,d,n).

Another alternative to improve efficiency in the second analysis could be to extend the type based analysis in the style of [GS00] so that it mimicked the powerful abstract interpretation with less cost.

References

[Bar93] G. Baraki. *Abstract Interpretation of Polymorphic Higher-Order Functions.* PhD thesis, University of Glasgow, February 1993.

[BFGJ] C. Baker-Finch, K. Glynn, and S. L. Peyton Jones. Constructed Product Result Analysis for Haskell. Submitted to *International Conference on Functional Programming, ICFP'00.*

[BHA86] G. L. Burn, C. L. Hankin, and S. Abramsky. The Theory of Strictness Analysis for Higher Order Functions. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *LNCS*, pages 42–62. Springer-Verlag, October 1986.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[GS00] J. Gustavsson and J. Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 279–294, 2000.

[Hen82] P. Henderson. Purely Functional Operating Systems. In *Functional Programming and its Applications: An Advanced Course*, pages 177–191. Cambridge University Press, 1982.

[HO90] R. J. M. Hughes and J. O'Donnell. Expressing and Reasoning About Non-Deterministic Functional Programs. In *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989*, pages 308–328, London, UK, 1990. Springer-Verlag.

[Hun91] S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, October 91.

[JHH+93] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Information Technology, Keele, DTI/SERC*, pages 249–257, 1993.

[JPS96] S. L. Peyton Jones, W. Partain, and A. L. M. Santos. Let-floating: moving bindings to give faster programs. *International Conference on Functional Programming ICFP'96*, May 1996.

[KPR00] U. Klusik, R. Peña, and F. Rubio. Replicated Workers in Eden. 2nd International Workshop on Constructive Methods for Parallel Programming (CMPP 2000). To be published by Nova Science, 2000.

[McC63] J. McCarthy. Towards a Mathematical Theory of Computation. In *Proc. IFIP Congress 62*, pages 21–28, Amsterdam, 1963. North-Holland.

[MH87] J. C. Martin and C. Hankin. Finding fixed points in finite lattices. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 426–445. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[PP93] S. L. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In *Glasgow Workshop on Functional Programming 1993*, Workshops in Computing, pages 201–220. Springer-Verlag, 1993.

[PPRS00] C. Pareja, R. Peña, F. Rubio, and C. Segura. Optimizing Eden by Transformation. In *Draft Proceedings of the 2nd Scottish Functional Programming Workshop*, pages 197–212, 2000.

[PS98] S. L. Peyton Jones and A. L. M. Santos. A Transformation-based Optimiser for Haskell. *Science of Computer Programming 32(1-3):3-47*, September 1998.

[PS00] C. Pareja and C. Segura. Efecto de las Transformaciones de GHC sobre Edén. Technical Report 101-00. Dep. Sistemas Informáticos y Programación (Univ. Complutense de Madrid), 2000.

[Rub99] F. Rubio. Programación funcional paralela eficiente. Trabajo de Tercer Ciclo, Departamento de Sistemas Informáticos y Programación (Universidad Complutense de Madrid), 1999.

[San95] A. L. M. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Department of Computing Science. University of Glasgow, 1995.

[SS90] H. Søndergaard and P. Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27(6):505–517, May 1990.

[SS92] H. Søndergaard and P. Sestoft. Non-Determinism in Functional Languages. *Computer Journal*, 35(5):514–523, October 1992.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once Upon a Type. In *1995 Conf. on Functional Programming and Computer Architecture*, pages 1–11, 1995.

[WJ99] K. Wansbrough and S. L. Peyton Jones. Once upon a polymorphic type. In *The Twenty-sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1999.