Non-determinism Analyses in a Parallel-Functional Language

RICARDO PEÑA and CLARA SEGURA
Facultad de Informática, Universidad Complutense de Madrid
C/Juan del Rosal, n 8, 28040 Madrid, Spain
(*e-mail:* `ricardo@sip.ucm.es,csegura@sip.ucm.es`)

Abstract

The parallel-functional language Eden has a non-deterministic construct, the process abstraction `merge`, which interleaves a set of input lists to produce a single non-deterministic list. Its non-deterministic behaviour is a consequence of its reactivity: it immediately copies to the output list any value appearing at any of the input lists. This feature is essential in reactive systems and very useful in some deterministic parallel algorithms.

The presence of non-determinism creates some problems such that some internal transformations in the compiler must be disallowed. The paper describes several non-determinism analyses developed for Eden aimed at detecting the parts of the program that, even in the presence of a process `merge`, still exhibit a deterministic behaviour. A polynomial cost algorithm which annotates Eden expressions is described in detail.

A denotational semantics is described for Eden and the correctness of all the analyses is proved with respect to this semantics.

1 Introduction

The parallel-functional language Eden (Breitinger *et al.*, 1998b; Breitinger *et al.*, 1997; Breitinger *et al.*, 1998a) extends the lazy functional language Haskell by syntactic constructs to explicitly define processes and the communications between them. It is implemented by modifying the *Glasgow Haskell Compiler* (GHC) (Peyton Jones *et al.*, 1993). The three main new concepts are *process abstractions*, *process instantiations* and a non-deterministic process abstraction called `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 instantiation `e1 # e2` is evaluated, a new parallel process is created.

Non-determinism is introduced in Eden by means of a predefined process abstraction `merge :: Process [[a]] [a]`, which interleaves a set of input lists in a fair way to produce a single non-deterministic list. Its non-deterministic behaviour is a consequence of its reactivity: it immediately copies to the output list any value appearing at any of the input lists. In this way, `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. Eden is aimed at both types of applications.

The presence of non-determinism creates some problems in Eden such that some internal transformations in the compiler must be disallowed. In (Peña & Segura, 2001a) a solution was proposed to solve this problem: to develop a static analysis to determine when an Eden expression is sure to be deterministic and when it may be non-deterministic. Two different abstract interpretation based analyses were presented and compared with respect to expressiveness and efficiency. The first one $[\![\cdot]\!]_1$ was efficient (linear) but not very powerful, and the second one $[\![\cdot]\!]_2$ was powerful but very inefficient (exponential).

In (Peña & Segura, 2002) an intermediate analysis $[\![\cdot]\!]_3$ was presented that was a compromise between power and efficiency and its implementation was described. Its definition was based on the second analysis $[\![\cdot]\!]_2$. The improvement in efficiency was obtained by speeding up the fixpoint calculation by means of a widening operator *wop*, and by using an easily comparable representation of functions. By choosing different operators we obtained different variants of the analysis $[\![\cdot]\!]_3^{wop}$.

In (Peña & Segura, 2001b) we proved the relative correctness of these analyses showing that the less accurate ones were safe approximations to the more accurate ones. The absolute correctness of these analyses with respect to a denotational semantics for Eden has been proved in (Segura & Peña, 2003a). There, non-determinism is modelled by using Hoare powerdomains as semantic domains.

The current paper summarizes the more relevant results of the above cited papers and presents them in a uniform and organized way. It can be considered as a comprehensive and self-contained work where the reader can find all the relevant information about the analysis of non-determinism in Eden. As we will see, the semantics given and the analyses themselves abstract away the parallel and concurrent nature of Eden since processes are treated as functions. So, the paper can be also seen as a comprehensive study of non-determinism analyses in functional languages.

The analyses use conventional techniques in abstract interpretation as described in (Burn *et al.*, 1986), but the problem addressed is new in the analysis literature. The main contributions of the whole work can be summarized as follows:

- Definition of the abstract domains for the analyses, including higher-order domains and polymorphism.
- Definition of the abstract interpretations $[\![\cdot]\!]_1$, $[\![\cdot]\!]_2$ and $[\![\cdot]\!]_3^{\mathcal{W}}$.
- Implementation of $[\![\cdot]\!]_3^{\mathcal{W}}$.
- Denotational semantics for Eden using Hoare powerdomains.
- Definition of the abstraction and concretisation functions and proof of correctness.

The plan of the paper is the following: in Section 2 the language full Eden and its desugared version are summarized. A small example illustrates how to express reactive systems in Eden. Section 3 is devoted to non-determinism. After a general discussion of the problem, a denotational semantics for Eden is given. It does not exactly coincide with the one implemented in the compiler. Instead, it is an *upper*

approximation to it in the sense that the set of values denoted by an expression contains the values that may be produced by the implementation. Nevertheless, this semantics is enough for the purpose of proving the correctness of the analyses. Section 4 presents analyses $[\![\cdot]\!]_2$ and $[\![\cdot]\!]_3^{\mathcal{W}}$ and describes the Haskell implementation of the latter. Section 5, as it is typical in the abstract interpretation area, first provides the abstraction and concretisation functions for analysis $[\![\cdot]\!]_2$ and then presents its correctness proof. The proofs of the propositions can be found in a document supplementary to this paper and available through the web page of this journal. Finally, Section 6 surveys some related work and draws some conclusions.

2 The Parallel-functional Language Eden

2.1 Eden in a nutshell

The parallel-functional language Eden extends the lazy functional language Haskell by constructs to explicitly define processes and the communications between them. The three main new concepts are *process abstractions*, *process instantiations* and a non-deterministic process abstraction `merge`.

A *process abstraction* expression `process x -> e` of type `Process a b` defines the behaviour of a process having the formal parameter `x::a` as input and the expression `e::b` as output. An instantiation is achieved by using the predefined infix operator `(#) :: Process a b -> a -> b`. Process abstractions of type `Process a b` can be compared to functions of type `a -> b`, the main difference being that the former, when instantiated, are executed in parallel. Process instantiations can be compared to function applications: each time an expression `e1 # e2` is evaluated, a new parallel process is created to evaluate `(e1 e2)`.

The evaluation of an expression `e1 # e2` leads to the dynamic creation of a process together with its interconnecting communication channels. The instantiating or *parent process* will be responsible for evaluating and sending the value of `e2` via an implicitly generated channel, while the new *child process* will evaluate first the expression `e1` until a process abstraction `process x -> e` is obtained and then the application `(\ x -> e) e2`, returning the result via another implicitly generated channel. The instantiation protocol deserves some attention: (1) closure `e1` together with the closures of all the free variables referenced there (its whole environment) are *copied*, in the current evaluation state (possibly unevaluated), to a new processor, and the child process is created there to evaluate the expression `(\ x -> e) e2`, where the value of `e2` must be remotely received. (2) Expression `e2` is eagerly evaluated in the parent process. The resulting full normal form data is communicated to the child process as its input argument. (3) The normal form of the value of `(\ x -> e) e2` is sent back to the parent. For input or output tuples, independent concurrent threads are created to evaluate each component.

Processes communicate via *unidirectional channels* which connect one writer to exactly one reader. Once a process is running, only fully evaluated data objects are communicated. The only exceptions are lists, which are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to full normal

form and then transmitted. Concurrent threads trying to access input which is not available yet, are temporarily suspended. This is the only way in which Eden processes synchronize.

Lazy evaluation is changed to eager evaluation in two cases: processes are eagerly instantiated, and instantiated processes produce their output even if it is not demanded. These modifications aim at increasing the parallelism degree and at speeding up the distribution of the computation. In general, a process is implemented by several threads concurrently running in the same processor, so that different values can be produced independently. The concept of a virtually shared global graph does not exist. Each process evaluates its outputs autonomously.

The following example defines a simple reactive system where a set of user processes interact with a binary semaphore which provides mutual exclusion in the access to a critical region. A user process is an endless cycle of the sequence of states "`Think, Wait, Eat ...`", where `Eat` means that the process is inside the critical region and `Think` that it is outside. When a user needs to enter the critical region, sends a request to the semaphore and waits for an acknowledge. When it leaves the critical region, it sends a *release* message to the semaphore:

```
user :: Int -> Process [AcK] [Req]
user i = process acks -> cycle Think acks
         where cycle Think acks       = Req i : cycle Wait acks
               cycle Wait  (Ack:acks) = cycle Eat acks
               cycle Eat   acks       = Rel i : cycle Think acks
```

The semaphore life is also a cycle of the sequence of states "`Free, Busy ...`". It receives requests from the users and provides them with acknowledges, one user at a time, using a FIFO policy:

```
sem :: Process [Req] [[Ack]]
sem = process reqs -> cycle Free [] reqs
      where cycle Free (Req i:q) reqs         = reply i (cycle Busy q reqs)
            cycle Busy q         (Rel i:reqs) = cycle Free q reqs
            cycle st   q         (Req i:reqs) = cycle st (q ++ [Req i]) reqs
            reply 0 ~(rs:rss) = (Ack:rs) : rss
            reply i ~(rs:rss) = rs : reply (i-1) rss
```

The whole system is instantiated by a set of mutually recursive equations connecting n users to a `merge` process, this one to the semaphore and the latter to the users. The instance of `merge` is crucial to propagate users requests to the semaphore as soon as they are produced:

```
reqss = [user i # acks | (i,acks) <- zip [0..n-1] ackss]
reqs  = merge # reqss
ackss = sem # reqs
```

2.2 Core language

As Eden is implemented by modifying the Glasgow Haskell Compiler, the core language of Eden is an extension of Core-Haskell (Peyton Jones *et al.*, 1993). This

prog	→	$bind_1; \ldots; bind_m$		
bind	→	$v = expr$		{non-recursive binding}
		| **rec** $v_1 = expr_1; \ldots; v_m = expr_m$		{recursive binding}
expr	→	$expr\ x$		{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: variable v or literal k}
		| $\Lambda\beta.expr$		{type abstraction}
		| $expr\ type$		{type application}
		| $expr\ \#\ x$		{process instantiation}
		| **process** $v \rightarrow expr$		{process abstraction}
		| **merge**		{non-deterministic process}
alts	→	$Calt_1; \ldots; Calt_m; [Deft]$	$m \geq 0$	
		| $Lalt_1; \ldots; Lalt_m; [Deft]$	$m \geq 0$	
Calt	→	$C\ v_1 \ldots v_m \rightarrow expr$	$m \geq 0$	{algebraic alternative}
Lalt	→	$k \rightarrow expr$		{primitive alternative}
Deft	→	$v \rightarrow expr$		{default alternative}
type	→	K		{basic types: integers, characters}
		| β		{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}
		| $\forall\beta.type$		{polymorphic type}

Fig. 1. Language definition and type expressions

is a simple functional language with second-order polymorphism, so it includes type abstraction and type application.

In Figure 1 the syntax of the language and of the type expressions is shown. There, v denotes a variable, k denotes a literal, x denotes an atom (a variable or a literal), and T denotes a type constructor. A program is a list of possibly recursive bindings from variables to expressions. Such expressions include variables, lambda abstractions, applications of a functional expression to an atom, constructor applications, primitive operators 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. When necessary we will write $e :: t$ to make explicit the type of an expression. A type may be a basic type K, a tuple typle $(t_1, \ldots, t_m)$, an algebraic type $T\ t_1 \ldots t_m$, a functional type $t_1 \rightarrow t_2$ or a polymorphic type $\forall\beta.t$. The second-order polymorphism is only used as a mechanism to preserve the Hindley-Milner polymorphic types along the transformations done at Core-Haskell level (Peyton Jones & Santos, 1998). Consequently we can assume the polymorphic types are Hindley-Milner despite the abstract syntax.

The new Eden expressions are a process abstraction **process** $v \rightarrow e$, and a process instantiation $e\ \#\ x$. There is also a new type $Process\ t_1\ t_2$ 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 and each tuple element represents an input/output channel of the process. Additionally, there is a predefined polymorphic constant **merge** of type $\forall\beta.Process\ [[\beta]]\ [\beta]$.

3 Non-determinism

3.1 Non-determinism in functional languages

The introduction of non-determinism in functional languages has a long tradition and has been a source of strong controversy. McCarthy (1963) introduced the operator `amb :: a -> a -> a` which non-deterministically chooses between two values. Henderson (1982) 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,

$$\texttt{let x = amb 0 1 in x + x} \neq \texttt{amb 0 1 + amb 0 1}$$

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 (1990) proposed a functional language in which non-determinism is compatible with referential transparency. However, Sondergaard and Sestoft (1990; 1992) 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 $e = op\ e_1 \cdots e_n$, 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 all of its operators are.

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 to 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*).

3.2 Denotational semantics for non-determinism in Eden

In this section we define a denotational semantics that approximates the actual semantics of Eden. Our aim is to prove the correctness of the non-determinism analyses we define in Section 4. Very recently it has been published in our group a complete denotational semantics (Hidalgo & Ortega, 2003) for Eden based on continuations. There, non-determinism is expressed by the fact that, after evaluating an expression, a process may arrive to *a set* of different states, so that several continuations are possible. Unfortunately this semantics is not appropriate for our purposes. On the one hand it provides lots of details that would obscure the proof of correctness. On the other, the set of states a process may arrive to do not constitute a mathematical domain and this is essential when abstract interpretation is used. The semantics we define here is enough to prove the correctness of the analyses. Moreover, as concurrency and parallelism aspects are abstracted away, the non-determinism analyses would also be correct for any non-deterministic functional language whose semantics is (upper) approximated by this one.

3.2.1 Intuitions for a simplified semantics

Under the definitions given in the previous section, we can characterize Eden as 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 be replaced by its denotation without affecting the resulting set of values (see Section 3.2.3 to confirm this issue).

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, within the same process, a variable is evaluated at most once and its value is shared thereafter. Consequently, variables are definite within the same process and are not definite within different processes. In general, in Eden the unfoldability property does not hold, except in the case that the unfolded expression is deterministic. This is a consequence of having definite variables within a process. So, there are some occurrences that surely have the same value but others may have different values. The following example illustrates this situation. Assume ne is a non-deterministic expression in

$$\begin{array}{l}\textbf{let } v = ne\\ \textbf{in } (p_1\ v)\#v + (p_2\ v)\#v\end{array}$$

The second and fourth occurrences of v necessarily have the same value as they are evaluated in the parent process. However the first and third occurrences may have different values as v is copied twice and evaluated in two children processes.

So, an upper approximation to the semantics can be obtained by considering that

- All the occurrences of each variable may have a different value, i.e. all the variables are non-definite.
- All functions behave as processes, and all function applications behave as process instantiations.

The denotational semantics defined below will make these assumptions. Such semantics is defined for a *simplified* version of the core language defined in Section 2.2. The simplifications are the following:

- We have removed polymorphism from the language, so that there are neither type abstractions nor type applications. Consequently we do not consider polymorphic types and we assume that algebraic types are defined as **data** $T = C_1\ t_{11} \dots t_{1n_1} \mid \dots \mid C_m\ t_{m1} \dots t_{mn_m}$.
- As polymorphism is omitted, the **merge** operator is monomorphic, so we consider the existence of an instance $merge_t$ for every type t. Additionally we simplify this operator so that it merges just two lists of values: $merge_t : [t] \to [t] \to [t]$. Eden's **merge** is more convenient since it may receive as arguments any finite number of lists, but it can be simulated by the simplified one, $merge_t$.
- Process abstractions **process** $v \to e$, process instantiations $e \ \# \ x$ and the type *Process* do not appear in the language either. Consequently, we will only have syntactical lambda abstractions and function applications (with the semantics of process abstractions and process instantiations).
- We omit here primitive operators, primitive cases and the default alternative as they do not add anything significantly new.

3.2.2 The domain of values

To capture the idea of a non-deterministic value, the traditional approach is to make an expression to denote a *set* of values. This is obvious for basic types such as integers, but things get more complex when we move to structured types such as functions or tuples. Should a functional expression denote a set of functions or a function from sets to sets? Should a tuple expression denote a set of tuples or a tuple of sets? Additionally, the denoted values should constitute a domain. In the literature, three powerdomains with different properties have been proposed: Hoare, Smyth and Plotkin powerdomains (Søndergaard & Sestoft, 1992). The first one models *angelic* or bottom-avoiding nondeterminism (in which bottom is never chosen unless it is the only option), the second one models *demonic* non-determinism (it chooses bottom whenever it is a possible option) and the third one models *erratic* non-determinism (in which bottom is an option similar to the other ones).

Regarding structured domains we have chosen a functional expression to denote a single function from sets to sets. In this sense, the following two bindings

$$\begin{aligned} f_1 = &\quad head(merge_{Int \to Int}[\lambda x.0][\lambda x.1]) \\ f_2 = &\quad \lambda x.head(merge_{Int}[0][1]) \end{aligned}$$

will both denote the function $\lambda x.\{0, 1, \bot\}$. That is, the information whether the non-deterministic decision is taken at binding evaluation time or at function application time is lost. Non-deterministic decisions are deferred as much as possible; in this example to function application time. This is consistent with the plural semantics we have adopted for our language in this section: several occurrences of the same variable (let us say f_1) may represent different values.

$$
\begin{array}{l}
A_K = \mathcal{P}([\![K]\!]) \quad \text{where } [\![Int]\!] = \mathbb{Z}_\perp \\
A_{(t_1,\ldots,t_m)} = A_{t_1} \times \ldots \times A_{t_m} \\
A_T = \mathcal{P}(\oplus_{i=1}^{m}(C_i \times \times_{j=1}^{n_i} A_{t_{ij}})_\perp) \\
A_{t_1 \to t_2} = [A_{t_1} \to A_{t_2}]
\end{array}
$$

Fig. 2. Domain of values

Regarding the selection of powerdomain, we have decided to use Hoare's one. This is consistent with the implementation of `merge` in Eden: if one of the input lists is blocked (i.e., it denotes $\perp$), `merge` will still produce an output list by copying values from the non-blocked list. Only if both lists are blocked will the output list be blocked. Nevertheless, `merge` will terminate only when both input lists terminate. This behaviour is very near to angelic non-determinism. If D is a domain, $\mathcal{P}(D)$ will denote the Hoare powerdomain of D. First, a preorder relation is defined in $P(D)$ (all subsets of D) as follows:

$$A \sqsubseteq_{P(D)} B \quad \textit{iff} \quad \forall a \in A . \exists b \in B . a \sqsubseteq_D b$$

This preorder relation induces an equivalence relation $\equiv \stackrel{\text{def}}{=} \sqsubseteq \cap \sqsupseteq$ identifying sets such as $\{0, 1, \perp\}$ and $\{0, 1\}$. Hoare powerdomain is the quotient $\mathcal{P}(D) \stackrel{\text{def}}{=} (P(D) - \emptyset)/\equiv$. A property enjoyed by all elements of a Hoare powerdomain is that they are downwards closed, i.e. $\forall x \in A . y \sqsubseteq_D x \Rightarrow y \in A$.

In Figure 2, the domains of semantic values for every type are defined. Notice that, for basic and algebraic types, the domains consist of sets of values while for tuples and functions, the domains consist of single values. Notice also that we identify the tuple of bottoms with the bottom of tuples and the function returning bottom with the bottom of functions. We could have distinguished them and still the propositions shown in this paper would remain true. In the definition for constructed types, $\oplus$ denotes the coalesced sum of (lifted) domains. Sets of values are needed for the constructed types because non-deterministic values of such types may contain several different constructors. However, those with only one constructor could be treated as tuples.

If the constructed type is recursive, notice that the recursive occurrences denote sets of values. For instance, a non-deterministic list would consist of a set of lists. A non-empty list of this set would consist of a head value and a tail value formed by a set of lists.

3.2.3 A maximal semantics: non-definite variables

In Figure 3 the approximated denotational semantics for Eden is given. There $\{v\}^*$ denotes the downwards closure of a value, i.e. a set of values containing all values below v. The environment ρ maps variables of type t to values of their corresponding non-deterministic domains A_t. The semantic function $[\![.]\!]$ maps an expression of type t and an environment ρ to a value in A_t. The only expression introducing sets of values is $merge_t$. Its behaviour is that of a lambda abstraction returning all the possible interleavings of all pairs of input lists. The detail of the auxiliary function *mergeS* is given in Figure 4.

$$
\begin{array}{l}
[\![v]\!]\ \rho = \rho(v) \\
[\![k]\!]\ \rho = \{k\}^* \\
[\![(x_1,\ldots,x_m)]\!]\ \rho = ([\![x_1]\!]\ \rho,\ldots,[\![x_m]\!]\ \rho) \\
[\![C\ x_1 \ldots x_m]\!]\ \rho = \{C\ [\![x_1]\!]\ \rho \ldots [\![x_m]\!]\ \rho\}^* \\
[\![\lambda v.e]\!]_2\ \rho = \lambda s \in A_{t_v}.[\![e]\!]\ \rho\ [v \mapsto s] \text{ where } v :: t_v \\
[\![e\ x]\!]\ \rho = ([\![e]\!]\ \rho)\ ([\![x]\!]\ \rho) \\
[\![merge_t]\!]\ \rho = \lambda s_1 \in A_{[t]}.\lambda s_2 \in A_{[t]}.\bigcup\{mergeS\ l_1\ l_2 \mid l_1 \in s_1, l_2 \in s_2\} \\
[\![\textbf{let } v = e \textbf{ in } e']\!]\ \rho = [\![e']\!]\ \rho\ [v \mapsto [\![e]\!]\ \rho] \\
[\![\textbf{let rec } \overline{\{v_i = e_i\}} \textbf{ in } e']\!]\ \rho = [\![e']\!]\ (\mathit{fix}\ (\lambda\rho'.\rho\ \overline{[v_i \mapsto [\![e_i]\!]\ \rho']})) \\
[\![\textbf{case } e \textbf{ of } (v_1,\ldots,v_m) \to e']\!]\ \rho = [\![e']\!]\ \rho\ \overline{[v_i \mapsto \pi_i([\![e]\!]\ \rho)]} \\
[\![\textbf{case } e \textbf{ of } \overline{C_i\ \overline{v_{ij}} \to e_i}]\!]\ \rho = \left\{ \begin{array}{l} \bot_{A_t} \text{ if } [\![e]\!]\ \rho = \bot_{A_T} \\ \bigsqcup_{A_t} \{[\![e_k]\!]\ \rho\overline{[v_{kj} \mapsto s_{kj}]}^{m_k} \mid C_k\ \overline{s_{kj}}^{m_k} \in [\![e]\!]\ \rho\} \text{ otherwise} \end{array} \right. \\
\qquad \text{where } e_i :: t
\end{array}
$$

Fig. 3. A denotational semantics for Eden

These decisions configure a plural semantics for Eden as every occurrence of the same variable within an expression is mapped to *all* possible values for that variable (see definitions for **let** and lambda in Figure 3). This is not the actual semantics of Eden, but just a safe upper approximation to it in the sense that, if an Eden expression e may evaluate to value v, then v is included in the set s denoted by e in the semantics, but s may include values that the implementation will never arrive to.

As an example, the expression

$$\textbf{let } f = head(merge_{Int \to Int}[\lambda x.0]\ [\lambda x.1])\ \textbf{in}\ (f\ 3) + (f\ 4)$$

in fact may only produce the values 0 or 2 while the approximated semantics will say that it may also produce the value 1. It is *maximal* in the sense that all variables are considered non-definite, while in the actual semantics only those variables duplicated in different processes may be non-definite if they are non-deterministic. Notice that with this approximated semantics unfoldability holds although in the actual semantics this is not true.

The reason for this maximal semantics is that, if we are able to show the correctness of the analysis with respect to it, then the analysis will be correct with respect to the actual semantics. As we will see, the sure value of the analysis is the deterministic one: if the analysis detects an expression as deterministic then it should be semantically deterministic.

An exception is the algebraic **case** expression where the variables in the right hand side of the alternatives are definite. The discriminant's value is a set that may contain different constructors, so we have to take the least upper bound of all the alternatives' values that match them. As the discriminant is immediately evaluated, the non-deterministic decision is immediately taken so that all the occurrences of the same variable in the right hand side have the chosen value.

For example, let a type **data** $Fool = C\ Int \mid C'\ Int$ and the values $s_1 = \{\bot, C\{0,\bot\}, C'\{0,\bot\}\ \}, s_2 = \{\bot, C\{1,\bot\},\ C\{0,\bot\}\ \}$ and $s_2' = \{\bot, C\{1,\bot\}, C\{0,\bot\},$

$$
\begin{array}{lll}
\mathit{mergeS}\ \bot\ \bot = \{\bot\} & \mathit{mergeS}\ \bot\ l_2 = \{l_2 +\!\!+ \bot\}^* & \mathit{mergeS}\ l_1\ \bot = \{l_1 +\!\!+ \bot\}^* \\
\mathit{mergeS}\ [\,]\ [\,] = \{[\,]\}^* & \mathit{mergeS}\ [\,]\ l_2 = \{l_2\}^* & \mathit{mergeS}\ l_1\ [\,] = \{l_1\}^* \\
\multicolumn{3}{l}{\mathit{mergeS}\ (s_1 : ls_1)\ (s_2 : ls_2) =} \\
\multicolumn{3}{l}{\quad \{s_1 : (\bigcup_{l' \in ls_1} \mathit{mergeS}\ l'\ (s_2 : ls_2)), s_2 : (\bigcup_{l' \in ls_2} \mathit{mergeS}\ (s_1 : ls_1)\ l')\}^*} \\
\multicolumn{3}{l}{\quad \text{where}\ \ \bot +\!\!+ \bot = \bot} \\
\multicolumn{3}{l}{\qquad\quad [\,] +\!\!+ \bot = \bot} \\
\multicolumn{3}{l}{\qquad\quad (xs : xss) +\!\!+ \bot = xs : \{xss' +\!\!+ \bot \mid xss' \in xss\}}
\end{array}
$$

Fig. 4. Non-determinism semantics

$C\{0, 1, \bot\}\ \}$. Let the expression $e' =$ **case** e **of** $\ C\ v \to v + v;\quad C'\ v' \to v' + 4$. If $[\![e]\!]\ \rho = s_1$, then $[\![e']\!]\ \rho = \{0, 4, \bot\}$. Notice that s_2 and s_2' are different: if $[\![e]\!]\ \rho = s_2$ then $[\![e']\!]\ \rho = \{0, 2, \bot\}$, but if $[\![e]\!]\ \rho = s_2'$, then $[\![e']\!]\ \rho = \{0, 1, 2, \bot\}$. This is because the variables in the right hand side of a **case** alternative are definite. We could have chosen another option when building the environments for the right hand sides: if there were several values with the same constructor then we could take the least upper bound of the components so that the variables would be non-definite:

$$
[\![\textbf{case}\ e\ \textbf{of}\ \overline{C_i\ \overline{v_{ij}} \to e_i}]\!]\ \rho = \begin{cases} \bot_{A_t} \text{ if } [\![e]\!]\ \rho = \bot_{A_T} \\ \bigsqcup_{A_t} \{[\![e_k]\!]\ \rho[\overline{v_{kj} \mapsto \bigsqcup s_{kj}}]^{m_k} \mid C_k\ \overline{s_{kj}}^{m_k} \in [\![e]\!]\ \rho\} \text{ otherwise} \end{cases}
$$

With this alternative version then, both when $[\![e]\!]\ \rho = s_2$ and when $[\![e]\!]\ \rho = s_2'$, we would have $[\![e']\!]\ \rho = \{0, 1, 2, \bot\}$ because v is bound to $\{0, \bot\} \sqcup \{1, \bot\} = \{0, 1, \bot\}$. We have chosen the first option because it is nearer to the actual semantics. The rest of the rules are self-explanatory.

4 Analyses for Non-determinism

4.1 Motivation for the analyses

This section introduces several abstract interpretation-based non-determinism analyses. They annotate the expressions with a mark which, in the simplest case is just d or n. The first one means that the expression is *sure* to be deterministic, while the second one means that it *may be* non-deterministic. So, a possible better name for these analyses would be *determinism* analyses because the sure value is the deterministic one.

We found at least three motivations for developing these analyses:

- On the one hand, to annotate the places in the text where equational reasoning may be lost due to the presence of non-determinism. This is important in an optimizing compiler such as that of Eden built on top of GHC. A lot of internal transformations such as *inlining* or *full laziness* are done on the assumption that it is always possible to replace equals by equals. This is not true when the expressions involved are non-deterministic. For instance, the full laziness transformation moves a binding out of a lambda when it does not depend on

```
e = let rec
        f = λp.λx.case p of
                    (p1, p2) → case p2 of
                                 0 → (p1, x)
                                 z → f (p1 * p1, p2 − 1) (x * p2)
    in let
        q = head(merge_Int [0] [1])
        f1 = f (q, 3) 4
        f2 = f (1, 2) q
        x1 = case f1 of (f11, f12) → f12
        x2 = case f2 of (f21, f22) → f21
       in (x1, x2)
```

Fig. 5. An example expression e

the lambda argument. So, the expression

$$\begin{array}{lll} \textbf{let} & f = \lambda x. & \textbf{let } y = e_1 \\ & & \textbf{in } e_2 \\ \textbf{in} & e_3 & \end{array}$$

is transformed to

$$\begin{array}{ll} \textbf{let} & y = e_1 \\ \textbf{in} & \textbf{let } f = \lambda x.e_2 \\ & \textbf{in } e_3 \end{array}$$

if e_1 does not depend on x. If e_1 is non-deterministic, this transformation restricts the set of values the whole expression may evaluate to, as now expression e_1 is evaluated only once instead of many times. There are other transformations that have the same effect. We have not found any that increases non-determinism.

- A second motivation is to be able to implement in the future a semantics for Eden, different from the currently implemented one, in which all variables will be guaranteed to be definite, i.e. they will denote the same value in all the processes. To this aim, when a non-deterministic binding is to be copied to a newly instantiated process, the runtime system will take care of previously evaluating the binding to normal form. Doing this evaluation for all bindings would make Eden more eager than needed and would decrease the amount of parallelism as more work would be done in parent processes. So, it is important to do this evaluation only when it is known that the binding is possibly non-deterministic.
- A third motivation could be to be able to inform the programmer of the deterministic expressions of the program. In this way, the part of the program where equational reasoning is still possible would be clearly determined. To this aim, a first step is doing the analysis at the core language level. A translation of the annotations to source level would also be required in order to provide the programmer with meaningful information. For the moment we have not implemented this translation.

In order to show what we expect from the analysis we show in Figure 5 an example

$$
\begin{array}{l}
Basic = \{d, n\} \text{ where } d \sqsubseteq n \\
D_{2K} = D_{2T} = Basic \\
D_{2(t_1,\ldots,t_m)} = D_{2t_1} \times \ldots \times D_{2t_m} \\
D_{2t_1 \to t_2} = [D_{2t_1} \to D_{2t_2}]
\end{array}
$$

Fig. 6. Abstract domains for the analysis $[\![\cdot]\!]_2$

expression $e :: (Int, Int)$. Given a pair of integers (p_1, p_2) and another integer x, the function $f :: (Int, Int) \to Int \to (Int, Int)$, calculates the pair $({p_1}^{2*p_2}, x * p_2!)$. Clearly, the result we expect from the analysis in this example is a tuple (d, d) telling us that both components of the resulting tuple are deterministic, even though q is non-deterministic. Less accurate analyses could produce an (safe) n in one of the components of the tuple or even in both of them.

4.2 An abstract interpretation-based analysis

Now we define an abstract interpretation-based analysis in the style of Burn, Hankin and Abramsky (1986), where the abstract domains corresponding to functional types are domains of continuous functions.

4.2.1 Abstract interpretation

In Figure 6 the abstract domains for $[\![\cdot]\!]_2$ are shown. There is a domain *Basic* with two values: d represents *determinism* and n *possible non-determinism*, with the ordering $d \sqsubseteq n$. This is the abstract domain corresponding to basic types and algebraic types. The abstract domains corresponding to a tuple type and a function type are respectively the cartesian product of the components' domains and the domain of continuous functions between the domains of the argument and the result. In (Peña & Segura, 2001a) polymorphism was also included, but in this paper we do not treat it.

Tuples are treated differently from other algebraic types because Eden processes use them to distinguish between different input and output channels. We want to detect the determinism of each one separately.

In Figure 7 the abstract interpretation for this analysis is shown. The interpretation of a tuple is the tuple of the abstract values of the components. Functions are interpreted as abstract functions. So, an application is interpreted as an abstract function application. In a recursive **let** expression the least fixpoint can be calculated by iterating over the elements of an ascending Kleene chain.

4.2.2 Flattening and unflattening functions

The interpretation of a constructor belongs to *Basic*, obtained as the least upper bound (lub) of the component's abstract values. But each component $x_i :: t_i$ has an abstract value belonging to D_{2t_i}, that must be first *flattened* to a basic abstract value. This is done by a function called *flattening function* $\phi_t : D_{2t} \to Basic$, defined

$$
\begin{array}{l}
[\![v]\!]_2\ \rho_2 = \rho_2(v) \\
[\![k]\!]_2\ \rho_2 = d \\
[\![(x_1,\ldots,x_m)]\!]_2\ \rho_2 = ([\![x_1]\!]_2\ \rho_2,\ldots,[\![x_m]\!]_2\ \rho_2) \\
[\![C\ x_1 \ldots x_m]\!]_2\ \rho_2 = \bigsqcup_i \phi_{t_i}([\![x_i]\!]_2\ \rho_2) \text{ where } x_i :: t_i \\
[\![e\ x]\!]_2\ \rho_2 = ([\![e]\!]_2\ \rho_2)\ ([\![x]\!]_2\ \rho_2) \\
[\![\lambda v.e]\!]_2\ \rho_2 = \lambda z \in D_{2t_v}.[\![e]\!]_2\ \rho_2\ [v \mapsto z] \text{ where } v :: t_v \\
[\![merge_t]\!]_2\ \rho_2 = \lambda z_1 \in Basic.\lambda z_2 \in Basic.n \\
[\![\textbf{let } v = e \textbf{ in } e']\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ [v \mapsto [\![e]\!]_2\ \rho_2] \\
[\![\textbf{let rec } \overline{\{v_i = e_i\}} \textbf{ in } e']\!]_2\ \rho_2 = [\![e']\!]_2\ (\mathit{fix}\ (\lambda\rho_2'.\rho_2\ \overline{[v_i \mapsto [\![e_i]\!]_2\ \rho_2']})) \\
[\![\textbf{case } e \textbf{ of } (v_1,\ldots,v_m) \to e']\!]_2\ \rho_2 = [\![e']\!]_2\ \rho_2\ \overline{[v_i \mapsto \pi_i([\![e]\!]_2\ \rho_2)]} \\
[\![\textbf{case } e \textbf{ of } \overline{C_i\ \overline{v_{ij}} \to e_i}]\!]_2\ \rho_2 = \begin{cases} \mu_t(n) \text{ if } [\![e]\!]_2\ \rho_2 = n \\ \bigsqcup_i [\![e_i]\!]_2\ \rho_{2i} \text{ otherwise} \end{cases} \\
\qquad\qquad \text{where } \rho_{2i} = \rho_2\ \overline{[v_{ij} \mapsto \mu_{t_{ij}}(d)]}, v_{ij} :: t_{ij}, e_i :: t
\end{array}
$$

Fig. 7. Abstract interpretation $[\![\cdot]\!]_2$

$$
\begin{array}{ll}
\phi_t : D_{2t} \to Basic & \mu_t : Basic \to D_{2t} \\
\phi_K = \phi_T = id_{Basic} & \mu_K = \mu_T = id_{Basic} \\
\phi_{(t_1,\ldots,t_m)}(e_1,\ldots,e_m) = \bigsqcup_i \phi_{t_i}(e_i) & \mu_{(t_1,\ldots,t_m)}(b) = (\mu_{t_1}(b),\ldots,\mu_{t_m}(b)) \\
\phi_{t_1\to t_2}(f) = \phi_{t_2}(f(\mu_{t_1}(d))) & \mu_{t_1\to t_2}(b) = \begin{cases} \lambda z \in D_{2t_1}.\mu_{t_2}(n) \text{ if } b = n \\ \lambda z \in D_{2t_1}.\mu_{t_2}(\phi_{t_1}(z)) \text{ if } b = d \end{cases}
\end{array}
$$

Fig. 8. Functions ϕ_t and μ_t

in Figure 8. The idea is to flatten the tuples (by applying the lub operator) and to apply the functions to deterministic arguments. As an example, if $t = Int \to Int$, $\phi_t(\lambda z.z) = \phi_t(\lambda z.d) = d$. In Figure 9 we show the flattening function for the type $(Int \to Int) \to Int \to Int$.

We have two different kinds of *case* expressions (for tuple and algebraic types). The more complex one is the algebraic *case*. Its abstract value is non-deterministic if either the discriminant or any of the expressions in the alternatives is non-deterministic. Note that the abstract value of the discriminant e, let us call it b, belongs to *Basic*. That is, when it was interpreted, the information about the components was lost. We want now to interpret each alternative's right hand side in an extended environment with abstract values for the variables $v_{ij} :: t_{ij}$ in the left hand side of the alternative. We do not have such information, but we can safely approximate it by using the *unflattening function* $\mu_t : Basic \to D_{2t}$ defined in Figure 8. Given a type t, it *unflattens* a basic abstract value and produces an abstract value in D_{2t}. The idea is to obtain the best safe approximation both to d and n in a given domain.

In particular n is mapped to the top of the domain D_{2t}, and d to the biggest value in D_{2t} that reflects our idea of determinism, considering that a function is deterministic if it produces deterministic results from deterministic arguments. 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 applies the unflattening function corresponding to the type of the result. The unflattening of n for a function

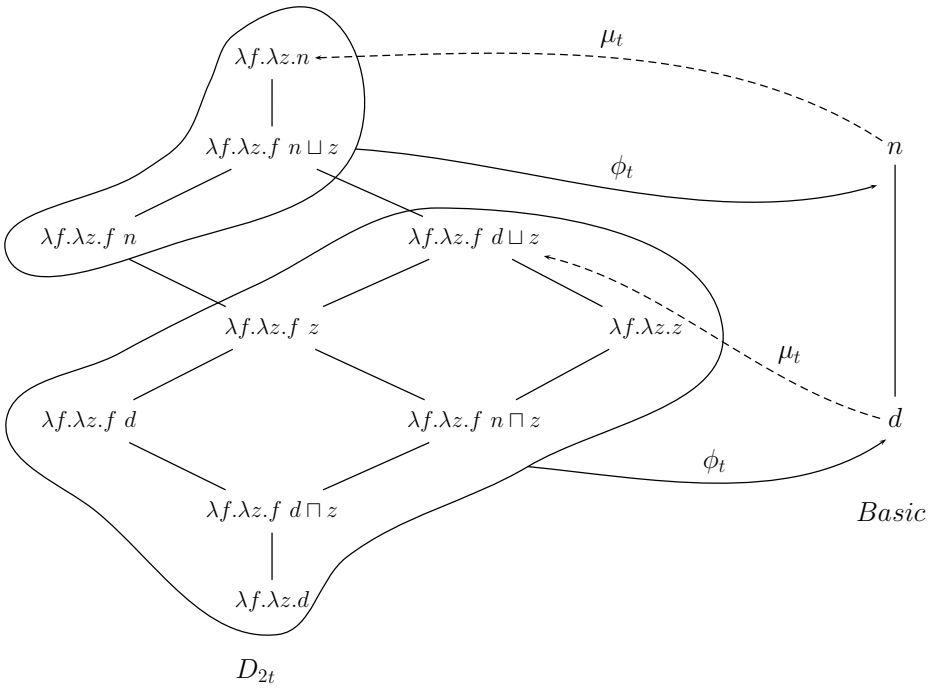

Fig. 9. Flattening and unflattening functions for $t = (Int \to Int) \to Int \to Int$

type is the function that returns a non-deterministic result independently of the argument. In Figure 9 we show the unflattening function for the type $(Int \to Int) \to Int \to Int$.

The flattening and unflattening functions are mutually recursive. They have some interesting properties studied in (Peña & Segura, 2001b). In particular the fact that they are a Galois insertion pair (Cousot & Cousot, 1979) is essential in the correctness proof of the analysis.

4.3 An efficient approximation

4.3.1 Introduction

The exponential cost of $[\![\cdot]\!]_2$ is due to the fixpoint calculation (Peña & Segura, 2001b). At each iteration a comparison between abstract values is done. Such comparison is exponential in case functional domains are involved. So, a good way of speeding up the calculation of the fixpoint is finding a quickly comparable representation of functions. Some different techniques have been developed in this direction, such as frontiers algorithms (Peyton Jones & Clack, 1987) and widening/narrowing operators (Cousot & Cousot, 1977; Hankin & Hunt, 1992). Here, we will represent functions by *signatures*. A signature for a function is obtained by *probing* the function with some explicitly chosen combinations of arguments. For example, in the strictness analysis of Peyton Jones and Partain (1993), a function f with m arguments was probed with m combinations of arguments, those where $\bot$ occupies each argument position and the rest of arguments are given a $\top$

$$
\begin{array}{ll}
S_K = S_T = \{D, N\} \text{where } D \preceq N & \mathcal{H}_K = \mathcal{H}_T = 1 \\
S_{(t_1,\ldots,t_m)} = S_{t_1} \times \ldots \times S_{t_m} & \mathcal{H}_{(t_1,\ldots,t_m)} = \sum_{i=1}^{m} \mathcal{H}_{t_i} \\
S_t = \{s_1\ s_2\ \ldots\ s_m\ s_{m+1} \mid & \mathcal{H}_t = (m+1)\ \mathcal{H}_{t_r} \\
\quad \forall\ i \in \{1..(m+1)\}.s_i \in S_{t_r} \wedge s_{m+1} \preceq s_i\} & \quad \text{where} \\
\quad \text{where} \quad t = t_1 \to t_2 & \qquad t = t_1 \to t_2 \\
\qquad\quad m = nArgs(t), t_r = rType(t) & \qquad m = nArgs(t), t_r = rType(t)
\end{array}
$$

Fig. 10. The domain of signatures and its height $\mathcal{H}_t$

value: $\bot, \top, \ldots, \top;\ \top, \bot, \top, \ldots, \top;\ \ldots;\ \top, \top, \ldots, \bot$. So, for example, the function $f = \lambda x :: Int.\lambda y :: Int.y$ has a signature $\top\ \bot$.

If we probe only with some arguments, different functions may have the same signature and consequently some information is lost. Then the fixpoint calculation is not exact, but just approximate. A compromise must be found between the amount of information the signature keeps and the cost of signatures comparison. Several probings can be proposed. Here we concentrate on the one we have implemented, and mention other possibilities in Section 5.1. We probe a function of m arguments with $m+1$ combinations of arguments. In the first m combinations, a non-deterministic abstract value (of the corresponding type) $\mu_{t_i}(n)$ occupies each argument position while a deterministic abstract value $\mu_{t_i}(d)$ is given to the rest of the arguments: $\mu_{t_1}(n), \mu_{t_2}(d), \ldots, \mu_{t_m}(d);\ \mu_{t_1}(d), \mu_{t_2}(n), \ldots, \mu_{t_m}(d);\ \ldots;\ \mu_{t_1}(d), \mu_{t_2}(d), \ldots, \mu_{t_m}(n)$. In the $(m+1)$-th combination, all the arguments are given a deterministic value: $\mu_{t_1}(d), \mu_{t_2}(d), \ldots, \mu_{t_m}(d)$. This is the most important combination as it tells us whether the function is deterministic or it may be non-deterministic.

4.3.2 The domain of signatures

In Figure 10 the domains S_t of signatures are formally defined. The domain corresponding to a basic or an algebraic type is a two-point domain, very similar to the *Basic* domain. However we will use uppercase letters D and N when talking about signatures. The domain corresponding to a tuple type is a tuple of signatures of the corresponding types, for example we could have (D, N) for the type (Int, Int). The ordering between tuples is the usual componentwise one. With respect to the functions some intuition must be given. If a function has m arguments then its signature is composed by $m+1$ signatures, each one corresponding to the (non-functional) type of the result. By m arguments, we mean that the type is $t_1 \to \ldots \to t_m \to t_r$, where t_r is not functional. We will call this type the *unrolled* version of the functional type. As an example, the unrolled version of $Int \to (Int \to\ (Int, Int))$ is $Int \to Int \to (Int, Int)$.

Three useful functions, *nArgs*, *rType* and *aTypes*, can be easily defined. Given a type t, the first one returns the number of arguments of t; the second one returns the (non-functional) type of its result (it is the identity in the rest of cases); and the third one returns the list (of length $nArgs(t)$) of the types of the arguments. Then the unrolled version of a type t has $nArgs(t)$ arguments of types $aTypes(t)$,

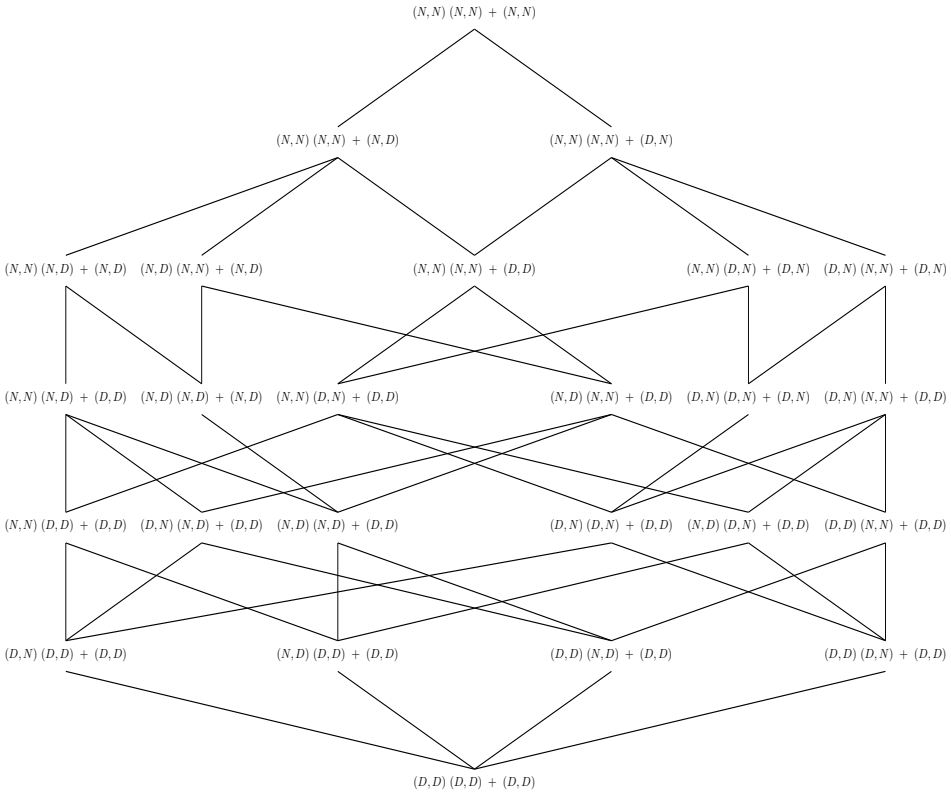

Fig. 11. Signatures for $Int \to Int \to (Int, Int)$

and $rType(t)$ as result type. In order to make the signatures for a function type readable, in the examples the last component is separated with a + symbol. So, an example of signature for the type $Int \to (Int, Int)$ could be $(N, D) + (D, D)$. But not every sequence of signatures is a valid signature. As we have previously said, the last component is obtained by probing the function with all the arguments set to a deterministic value, while the rest of them are obtained by probing the function with one non-deterministic value. As the functions are monotone, this means that the last component must always be less than or equal to all the other components. The ordering between the signatures ($\preceq$) is componentwise, so least upper bound and greatest lower bound can also be obtained in the same way. It is easy to see that with this ordering, the domain of signatures S_t for a given type t is a complete lattice of height $\mathcal{H}_t$, see Figure 10. In Figure 11 the domain S_t, where $t = Int \to Int \to (Int, Int)$ is shown.

4.3.3 The probing

Now we define the probing function $\wp_t :: D_{2t} \to S_t$, that given an abstract value in D_{2t}, obtains the corresponding signature in S_t. In Figure 12 the formal definition is shown.

$$
\begin{array}{l}
\wp_t :: D_{2t} \to S_t \\
\wp_K(b) = \wp_T(b) = B \\
\wp_{(t_1,\ldots,t_m)}(e_1,\ldots,e_m) = (\wp_{t_1}(e_1),\ldots,\wp_{t_m}(e_m)) \\
\wp_t(f) = \wp_{t_r}(f\ \mu_{t_1}(n)\ \mu_{t_2}(d)\ldots\mu_{t_m}(d))\ \wp_{t_r}(f\ \mu_{t_1}(d)\ \mu_{t_2}(n)\ldots\mu_{t_m}(d))\ldots \\
\qquad \wp_{t_r}(f\ \mu_{t_1}(d)\ \mu_{t_2}(d)\ldots\mu_{t_m}(n))\ \wp_{t_r}(f\ \mu_{t_1}(d)\ \mu_{t_2}(d)\ldots\mu_{t_m}(d)) \\
\quad \text{where}\ \ t = t'_1 \to t'_2,\ \ t_r = rType(t),\ \ [t_1,\ldots,t_m] = aTypes(t)
\end{array}
$$

Fig. 12. The probing function

$$
\begin{array}{l}
\Re_t :: S_t \to D_{2t} \\
\Re_K(B) = \Re_T(B) = b \\
\Re_{(t_1,\ldots,t_m)}(s1,\ldots,s_m) = (\Re_{t_1}(s_1),\ldots,\Re_{t_m}(s_m)) \\
\Re_t(\overline{s_j}) = \lambda\overline{z_j \in D_{2t_j}}.
\begin{cases}
\Re_{t_r}(s_{m+1}) \text{ if } \bigwedge_{j=1}^{m} z_j \sqsubseteq \mu_{t_j}(d) \\
\Re_{t_r}(s_i) \text{ if } \bigwedge_{j=1,j\neq i}^{m} z_j \sqsubseteq \mu_{t_j}(d) \wedge z_i \not\sqsubseteq \mu_{t_i}(d) \quad \forall i \in \{1..m\} \\
\mu_{t_r}(n) \text{ otherwise } (m > 1)
\end{cases} \\
\quad \text{where}\ \ \overline{s_j} = s_1\ldots s_m\ s_{m+1},\ \ t = t'_1 \to t'_2 \\
\qquad m = nArgs(t),\ \ t_r = rType(t),\ \ [t_1,\ldots,t_m] = aTypes(t)
\end{array}
$$

Fig. 13. The unflattening function corresponding to the probing

The signature of a basic value b is the corresponding basic signature B, that is, if $b = d$ then $B = D$ and if $b = n$ then $B = N$. The signature of a tuple is the tuple of signatures of the components. And finally, the signature for a function $f :: t$ is a sequence of $m+1$ signatures, where $m = nArgs(t)$, that are obtained by probing f with the combinations of arguments we have previously mentioned.

We have already said that in the probing process some information is lost. This means that a signature represents several abstract values. When we want to recover the original value, we can only return an approximation. This is what the *signatures unflattening function* $\Re_t :: S_t \to D_{2t}$ does. This function is defined in Figure 13. All the cases but the functional one are simple. Given a signature $s = s_1\ldots s_m\ s_{m+1}$, where $s \in S_t$, $\Re_t(s)$ is a function of m arguments $z_i \in D_{2t_i}$. We know that the last element s_{m+1} was obtained by probing the original function with $\mu_{t_i}(d)$, $i \in \{1..m\}$. So, if all the arguments are less than or equal to the corresponding $\mu_{t_i}(d)$, then the unflattening of s_{m+1} can be safely returned. The original function might have more precise information for some of the arguments combinations below $\mu_{t_i}(d)$, but now it is lost. We already know that s_i was obtained by probing the original function with $\mu_{t_i}(n)$ value for the ith argument and $\mu_{t_j}(d)$ for the rest of them ($j \in \{1..m\}$, $j \neq i$). So, if all the arguments but the ith one are less than or equal to the corresponding $\mu_{t_j}(d)$, then we can safely return the unflattening of s_i. Again we are losing information. If there is more than one value that is not less than or equal to the corresponding $\mu_{t_j}(d)$, we can only return the pessimistic value $\mu_{t_r}(n)$, as we do not have information for these combinations of arguments in the signature.

We have said that we will use a widening operator to speed up the fixpoint calculation. This is defined as $\mathcal{W}_t = \Re_t \cdot \wp_t$. In fact we will prove that $\mathcal{W}_t$ is an upper closure operator ($\mathcal{W}_t \sqsupseteq id_{D_{2t}}$). The definition of a widening operator is more general (Cousot & Cousot, 1977), but given an upper closure operator $\mathcal{W}_t$, we can define a corresponding widening operator $\nabla_t = \lambda(x, y).x \sqcup \mathcal{W}_t(y)$, as done by Hankin and Hunt (1992). So we will use the term 'widening operator' instead, as done by Peyton Jones and Partain (1993).

4.3.4 The analysis

The analysis is very similar to $[\![\cdot]\!]_2$, presented in Section 4.2. We will use the underscript 3 to identify it. The only expression where there are differences is the recursive **let** expression where a fixpoint must be calculated:

$$[\![\textbf{let rec } \overline{\{v_i = e_i\}} \textbf{ in } e']\!]_3\ \rho_3 = [\![e']\!]_3\ (\mathit{fix}\ (\lambda\rho'_3.\rho_3\ \overline{[v_i \mapsto \mathcal{W}_{t_i}([\![e_i]\!]_3\ \rho'_3)]}))$$

where $e_i :: t_i$. Notice that by modifying the widening operator we can have several different variants of the analysis. We can express them parameterised by the (collection of) widening operator wop_t, $[\![\cdot]\!]_3^{wop}$.

4.3.5 Some theoretical results

We prove now some properties that will help in the implementation of the analysis. Proposition 1 tells us that $\wp_t$ and $\Re_t$ are a Galois insertion pair, which means that $\Re_t$ recovers as much information as possible, considering how the signature was built. As a consequence, $\mathcal{W}_t$ is a widening operator. Proposition 2 tells us that $\mu_t(d)$ and $\mu_t(n)$ can be represented by their corresponding signatures without losing any information, which will be very useful in the implementation of the analysis. Finally, Proposition 3 tells us that the comparison between an abstract value and $\mu_t(d)$ can be done by comparing their corresponding signatures, which is much less expensive. This will be very useful in the implementation, as such comparison is done very often. In the worst case it is made in $\mathcal{H}_t$ steps.

Proposition 1
For each type t,

(a) The functions $\wp_t$, $\Re_t$, and $\mathcal{W}_t$ are monotone and continuous.
(b) $\mathcal{W}_t \sqsupseteq id_{D_{2t}}$.
(c) $\wp_t \cdot \Re_t = id_{S_t}$.

Proposition 2
For each type t, $\mathcal{W}_t \cdot \mu_t = \mu_t$.

Proposition 3
For each type t, $\forall z \in D_{2t}.z \sqsubseteq \mu_t(d) \Leftrightarrow \wp_t(z) \preceq \wp_t(\mu_t(d))$.

Propositions 1 and 2 can be proved by structural induction on t and both are used to prove Proposition 3. The proofs can be found in (Peña & Segura, 2001b).

$$
\begin{array}{lcl@{\qquad}lcl}
av & \to & b & b & \to & \mid d \\
 & & \mid (av_1, \ldots, av_m) & & & \mid n \\
 & & \mid \lambda v.(e, \rho) & aw & \to & \mid b \\
 & & \mid \lfloor \mathrm{F} \rfloor [av_1, \ldots, av_m] & & & \mid (aw_1, \ldots, aw_m) \\
 & & \mid aw & & & \mid <t,\ aw_1 \ldots aw_m + aw> \\
 & & & & & \mid <t,\ + aw>
\end{array}
$$

Fig. 14. Abstract values definition

4.4 Analysis implementation

4.4.1 Introduction

In this section we describe the main aspects of the analysis implementation. The algorithm we describe here not only obtains the abstract values of the expressions, but it also annotates each expression (and its subexpressions) with its corresponding signature. A full version of this algorithm has been implemented in Haskell. The implementation of the analysis includes also a little parser and a pretty printer (Hughes, 1995). It is important to annotate the subexpressions, even inside the body of a lambda-abstraction (see full laziness transformation in Section 4.1).

In the algorithm we make use of the fact that it is implemented in a lazy functional language. The interpretation of a lambda $\lambda v.e$ in an environment ρ is an abstract function. We will use a suspension $\lambda v.(e, \rho)$ to represent the abstract value of $\lambda v.e$, see Figure 14. Only when the function is applied to an argument, the body e of the function will be interpreted in the proper environment, emulating in this way the behaviour of the abstract function. So, we use the lazy evaluation of Haskell as our interpretation machinery. Otherwise, we should build a whole interpreter which would be less efficient. But this decision introduces some problems. Sometimes we need to build an abstract function that does not come from the interpretation of a lambda in the program. There are several situations where this happens. One of these is the application of $\mu_t(b)$ when t is a function type. By Proposition 2 we can use the corresponding signature to represent $\mu_t(b)$ without losing information. So, in this case we do not need to build a function. Given a basic value b, function $\mu'_t = \wp_t \cdot \mu_t$, defined in Figure 15, returns the signature of $\mu_t(b)$.

We also need to build a function when computing a lub of functions. In this case, we use a new suspension $\lfloor \mathrm{F} \rfloor [av_1, \ldots, av_m]$, see Figure 14. When the function is applied, the lub will be computed, see Figure 17.

4.4.2 Abstract values definition

In the implementation of the analysis, signatures are considered also as abstract values, where a signature $s \in S_t$ is just a representation of the abstract value $\Re_t(s)$. In Figure 14 the abstract values are defined. They can be basic abstract values d or n, that represent both a true basic abstract value or a basic signature. Tuples of abstract values are also abstract values. A functional abstract value may have several different representations: it may be represented by a signature or as

$$\mu'_t :: Basic \to S_t$$
$$\mu'_t(b) = \begin{cases} B \text{ if } t = K, t = T \\ (\mu'_{t_1}(b), \ldots, \mu'_{t_m}(b)) \text{ if } t = (t_1, \ldots, t_m) \\ <t,\ \mu'_{t_r}(n)\ \overset{(m)}{\cdots}\ \mu'_{t_r}(n)\ +\ \mu'_{t_r}(b)> \quad \text{if } t = t_1 \to t_2 \\ \quad \text{where } \ m = nArgs(t),\ \ t_r = rType(t) \end{cases}$$

Fig. 15. The signatures corresponding to $\mu_t(n)$ and $\mu_t(d)$.

$$n \sqcup b = n$$
$$d \sqcup b = b$$
$$(av_1, \ldots, av_m) \sqcup (av'_1, \ldots, av'_m) = (av_1 \sqcup av'_1, \ldots, av_m \sqcup av'_m)$$
$$<t,\ aw_1 \ldots aw_m + aw> \sqcup <t,\ aw'_1, \ldots, aw'_m + aw'> =$$
$$<t,\ (aw_1 \sqcup aw'_1) \ldots (aw_m \sqcup aw'_m) + (aw \sqcup aw')>$$
$$<t,\ +aw> \sqcup <t,\ +aw'> = <t,\ +(aw \sqcup aw')>$$
$$<t,\ aw_1 \ldots aw_m + aw> \sqcup <t,\ +aw'> = <t,\ +aw'> \sqcup <t,\ aw_1 \ldots aw_m + aw>$$
$$= <t,\ +(aw \sqcup aw')>$$
$$(\lfloor_{\mathrm{F}}\rfloor\, avs) \sqcup av = av \sqcup (\lfloor_{\mathrm{F}}\rfloor\, avs) = \lfloor_{\mathrm{F}}\rfloor\, av : avs$$
$$av \sqcup \lambda v.(e, \rho) = \lambda v.(e, \rho) \sqcup av = \lfloor_{\mathrm{F}}\rfloor\, [av, \lambda v.(e, \rho)]$$
$$\sqcup\, [av_1, \ldots, av_m] = av_1 \sqcup \ldots \sqcup av_m$$

Fig. 16. Lub operator definition

a *suspension*. In Figure 14 a functional signature is either $<t, aw_1 \ldots aw_m + aw>$ or $<t, +aw>$. The first one is a normal signature. The signature $<t, +aw>$ represents a function returning aw when all the arguments are deterministic (that is, less than or equal to $\mu_{t_i}(d)$) and $\mu_{t_r}(n)$ otherwise. So, it is just a particular case of $<t, aw_1 \ldots aw_m + aw>$ where $aw_i = \mu_{t_r}(n)$ $(i \in \{1..m\})$. A function may also be represented as a suspension. As we have previously said, it can be a suspended lambda abstraction $\lambda v.(e, \rho)$ or a suspended lub $\lfloor_{\mathrm{F}}\rfloor\, [av_1, \ldots, av_m]$.

In Figure 16 the lub operator between abstract values is defined. For basic values/signatures and tuples it is simple. In the functional case, if both functions are represented by a signature then we just apply the lub operator componentwise. If one of the functions is a suspension, then the result is a lub suspension.

4.4.3 Abstract application of a function

In Figure 17 the definition of the application of an abstract function to an abstract argument is shown. In case the abstract function is a signature of the form $<t_1 \to \ldots \to t_m \to t_r,\ aw_1 \ldots aw_m + aw>$ we check if the argument av' is less than or equal to $\mu_{t_1}(d)$. This is done by comparing their signatures, $\wp_{t_1}(av')$ and $\mu'_{t_1}(d)$ (this can be done by Proposition 3). If this happens, we will discard the first element aw_1 of the signature and return $<t_2 \to \ldots \to t_m \to t_r, aw_2 \ldots aw_m\ +\ aw>$, as these elements have been obtained by giving the first argument a value $\mu_{t_1}(d)$. Otherwise, we can return aw_1 as result of the function, only if the rest of the arguments are deterministic, so a signature $<t_2 \to \ldots \to t_m \to t_r, +aw_1>$ is returned. If the abstract function is a signature $<t_1 \to \ldots \to t_m \to t_r,\ +aw>$, only if all the

$(\lambda v.(e,\rho))\ av = [\![e]\!]'\ \rho[v \mapsto (av, aw, b)]$ where $v :: t_v,\quad aw = \wp_{t_v}(av),\quad b = \phi_{t_v}(av)$
$(\lfloor_F\rfloor\,[av_1,\ldots,av_m])\ av' = \bigsqcup\ [av_1\ av',\ldots,av_m\ av']$
$<t_1 \to \ldots \to t_m \to t_r,\ aw_1 \ldots aw_m + aw > av'\ (m > 1)$
  $\mid \wp_{t_1}(av') \preceq \mu'_{t_1}(d) = <t_2 \to \ldots \to t_m \to t_r,\ aw_2 \ldots aw_m + aw >$
  $\mid$ otherwise $= <t_2 \to \ldots \to t_m \to t_r,\ + aw_1 >$
$<t_1 \to t_r,\ aw_1 + aw > av'$
  $\mid \wp_{t_1}(av') \preceq \mu'_{t_1}(d) = aw$
  $\mid$ otherwise $= aw_1$
$<t_1 \to \ldots \to t_m \to t_r,\ + aw > av'\ (m > 1)$
  $\mid \wp_{t_1}(av') \preceq \mu'_{t_1}(d) = <t_2 \to \ldots \to t_m \to t_r,\ + aw >$
  $\mid$ otherwise $= \mu'_{t_2 \to \ldots \to t_m \to t_r}(n)$
$<t_1 \to t_r,\ + aw > av'$
  $\mid \wp_{t_1}(av') \preceq \mu'_{t_1}(d) = aw$
  $\mid$ otherwise $= \mu'_{t_r}(n)$

Fig. 17. Application of abstract functions

arguments are deterministic (that is, less than or equal to $\mu_{t_i}(d)$) the value aw is returned. If any of them is not, a non-deterministic result is returned.

The suspensions are just a way of delaying the evaluation until the arguments are known. The application of a suspended function to an argument evaluates the function as far as it is possible, until the result of the function or a new suspension is obtained. If it is a suspension $\lambda v.(e,\rho)$, we continue by evaluating the body e with the interpretation function $[\![\cdot]\!]'$, studied in the following section. The environment ρ keeps the abstract values of all the free variables but v in e. So we just have to add a mapping from v to the abstract value of the argument.

If it is a suspended lub, we apply each function to the argument and then try to calculate the lub of the results. The algorithm proceeds by suspending and evaluating once and again.

4.4.4 The algorithm

In the algorithm there are two different interpretation functions $[\![\cdot]\!]'$ and $[\![\cdot]\!]$. Given a non-annotated expression e and an environment ρ, $[\![e]\!]\ \rho$ returns a pair $(av, e'@aw)$ where av is the abstract value of e, e' is e where all its subexpressions have been annotated, and aw is the external annotation of e. While annotations in the expressions are always signatures, the first component of the pair is intended to keep as much information as possible, except in the fixpoint calculation where it will be replaced by its corresponding signature. In Figure 18 the algorithm for $[\![\cdot]\!]$ is shown in pseudo-code. The one for $[\![\cdot]\!]'$ is very similar: $[\![e]\!]'\ \rho$ returns just the abstract value of the expression. The rest of computations of $[\![e]\!]\ \rho$ are not done. In an environment ρ, there is a triple (av, aw, b) of abstract values associated to each program variable v. The first component av is the abstract value of the expression, aw is the corresponding signature $\wp_t(av)$, and b is the basic abstract value corresponding to $\phi_t(av)$. As these three values may be used several times along the interpretation, they are calculated just once, when the variable is bound, and used wherever needed.

$[\![\cdot]\!]' :: Expr \to Env \to AbsVal$
$[\![e]\!]'\ \rho = \pi_1([\![e]\!]\ \rho)$
$[\![\cdot]\!] :: Expr(\ ) \to Env \to (AbsVal, Expr\ AbsVal)$
$[\![v]\!]\ \rho = (av, v@aw)$
where $(av, aw, \_) = \rho(v)$;
$[\![k]\!]\ \rho = (d, k@d)$
$[\![(x_1, \ldots, x_m)]\!]\ \rho = ((av_1, \ldots, av_m), (x'_1, \ldots, x'_m)@(aw_1, \ldots, aw_m))$
where $(av_i, x'_i) = [\![x_i]\!]\ \rho$; $x_i@aw_i = x'_i$
$[\![C\ x_1 \ldots x_m]\!]\ \rho = (aw, C\ x'_1 \ldots x'_m@aw) \quad \{x_i :: t_i\}$
where $(av_i, x'_i) = [\![x_i]\!]\ \rho$; $aw = \sqcup_{i=1}^{m}\ b_i$
$b_i =$ if $isvar(x_i)$ then $(\pi_3(\rho(x_i)))$ else d
$[\![\lambda v.e]\!]\ \rho = (a, (\lambda v@awv.e')@aw) \quad \{v :: t_v, (\lambda v.e) :: t\}$
where $a = \lambda v.(e, \rho)$; $aw = \wp_t(a)$
$awv = \mu'_{t_v}(n)$; $(\_\ , e') = [\![e]\!]\ \rho[v \mapsto (awv, awv, n)]$
$[\![e\ x]\!]\ \rho = (a, (e'\ x')@aw) \quad \{(e\ x) :: t\}$
where $(ae, e') = [\![e]\!]\ \rho$; $(ax, x') = [\![x]\!]\ \rho$
$a = ae\ ax$; $aw = \wp_t(a)$
$[\![merge_t]\!]\ \rho = (aw, merge_t@aw) \quad \{merge_t :: t_{merge}\}$
where $aw = \mu'_{t_{merge}}(n)$
$[\![\textbf{let}\ bind\ \textbf{in}\ e]\!]\ \rho = (a, (\textbf{let}\ bind'\ \textbf{in}\ e')@aw) \quad \{e :: t\}$
where $(\rho', bind') = [\![bind]\!]_B\ \rho$; $(a, e') = [\![e]\!]\ \rho'$ $e''@aw = e'$
$[\![\textbf{case}\ e\ \textbf{of}\ (v_1, \ldots, v_m) \to e']\!]\ \rho =$
$(a, (\textbf{case}\ (e_1@awe)\ \textbf{of}\ (v'_1, \ldots, v'_m) \to (e'_1@aw))@aw) \quad \{v_i :: t_i\}$
where $(ae, e1@awe) = [\![e]\!]\ \rho$; $aw_i = \pi_i(awe)$
$v'_i = v_i@aw_i$; $av_i = \pi_i(ae)$
$b_i = \phi_{t_i}(av_i)$; $(a, e'_1@aw) = [\![e']\!]\ \rho[\overline{v_i \mapsto (av_i, aw_i, b_i)}]$
$[\![\textbf{case}\ e\ \textbf{of}\ \overline{alt_i}]\!]\ \rho = (av, (\textbf{case}\ e'\ \textbf{of}\ \overline{alt'_i})@aw) \quad \{\textbf{case}\ e\ \textbf{of}\ \overline{alt_i} :: t\}$
where $(ae, e') = [\![e]\!]\ \rho$; $(av_i, alt'_i) = [\![alt_i]\!]_A\ ae\ \rho$
$C_i\ v_{i1} \ldots v_{im_i} \to (e_i@wa_i) = alt'_i$
$aw =$ if $ae = n$ then $\mu'_t(n)$ else $\sqcup_{i=1}^{m}\ aw_i$
$av =$ if $ae = n$ then $\mu'_t(n)$ else $\sqcup_{i=1}^{m} av_i$
$[\![v = e]\!]_B\ \rho = (\rho[v \mapsto (av, aw, b)], v@aw = e'@aw)$
where $(av, e'@aw) = [\![e]\!]\ \rho$; $b = \phi_{t_v}(av)$
$[\![\textbf{rec}\ \overline{v_i = e_i}]\!]_B\ \rho = (\rho_{fix}, \textbf{rec}\ \overline{v'_i = e''_i}) \quad \{v_i :: t_i\}$
where $\rho_{fix} = fix\ f\ init$; $init = \rho[\overline{v_i \mapsto (aw_i, aw_i, d)}]$; $aw_i = \mu'_{t_i}(d)$
$f\rho' = \rho'[\overline{v_i \mapsto (aw'_i, aw'_i, b_i)}]$
where $av'_i = [\![e_i]\!]'\ \rho'$; $aw'_i = \wp_{t_i}(av'_i)$; $b_i = \phi_{t_i}(av'_i)$
$(\_\ , e''_i) = [\![e_i]\!]\ \rho_{fix}$; $(\_, aw_i, \_) = \rho_{fix}(v_i)$; $v'_i = v_i@aw_i$
$[\![C\ v_1 \ldots v_m \to e]\!]_A\ avd\ \rho = (av, C\ v'_1 \ldots v'_m \to e') \quad \{v_i :: t_i\}$
where $aw_i = \mu'_{t_i}(avd)$; $(av, e') = [\![e]\!]\ \rho[\overline{v_i \mapsto (aw_i, aw_i, avd)}]$
$v'_i = v_i@aw_i$

Fig. 18. The expressions annotation algorithm

The computation of the first component of the result av follows the definition of $[\![\cdot]\!]_3^{\mathcal{W}}$, so we just explain the annotation part. In general, to annotate the expression we first recursively annotate its subexpressions and then calculate the annotation for the whole expression by probing the resulting abstract value (the first component) of the expression. But, in many cases the annotations of the subexpressions are used to build the annotation of the whole expression, which is more efficient.

4.4.5 Complexity of the analysis

Analysing the cost of the interpretation algorithm has proved to be a hard task. This is due to the fact that many of the functions involved —in particular $[\![\cdot]\!]$, $[\![\cdot]\!]'$, abstract application, $\wp_t$, and ϕ_t— are heavily mutually recursive. Fortunately, there are small functions whose cost can be directly computed. For instance, a comparison between two signatures in S_t, or computing their lub, can be done in $O(\mathcal{H}_t)$. So, the lub of m abstract values of type t is in $O((m-1)\ \mathcal{H}_t)$. The cost of $\mu'_t(b)$ is in $O(m + \mathcal{H}_{t_r})$, being $m = nArgs(t)$ and $t_r = rType(t)$. To analyse the cost of the main interpretation functions we define in (Peña & Segura, 2001b) two functions $s, s' : Expr \to Int$ respectively giving the 'size' of an expression e when interpreted by $[\![\cdot]\!]$ and by $[\![\cdot]\!]'$. Then $[\![e]\!]'\ \rho \in O(s'(e))$ and $[\![e]\!]\ \rho \in O(s(e))$. Most of the time, $s(e)$ and $s'(e)$ are linear with e using any intuitive notion of size of an expression and including in this notion the size of the types involved. There are three exceptions to this linearity:

Applications Interpreting a lambda binding with $[\![\cdot]\!]'$ costs $O(1)$ because a suspension is immediately created. But the body of this lambda is interpreted as many times as the lambda appears applied in the text, each time with possibly different arguments. Being e_λ the body of a lambda, the algorithm costs $O(s'(e_\lambda))$ each time the lambda is applied.

Probing a function It is heavily used by $[\![\cdot]\!]$ to annotate expressions with signatures and also by both $[\![\cdot]\!]$ and $[\![\cdot]\!]'$ in fixpoints. The cost of $\wp_t(e)$ involves $m+1$ abstract applications, each one to m parameters, being $m = nArgs(t)$. Calling e_λ to e's body, the cost will be in $O((m+1)\ s'(e_\lambda))$.

Fixpoints Assuming a recursive binding $v = e$ of functional type t, being $m = nArgs(t)$, $t_r = rType(t)$, and e_λ the body of e, algorithm $[\![\cdot]\!]'$ will compute a fixpoint in a maximum of $\mathcal{H}_t = (m+1)\ \mathcal{H}_{t_r}$ iterations. At each iteration, the signature of e is obtained, so the cost of fixpoints is in $O(m^2\ \mathcal{H}_{t_r}\ s'(e_\lambda))$. The annotation algorithm $[\![\cdot]\!]$ will add to this cost that of completely annotating e, which involves m probings more, each one with one parameter less, i.e. in total $O(m^2\ s'(e_\lambda))$.

Summarizing, the complete interpretation/annotation algorithm is linear with e except in applications —where the interpretation of the body must be multiplied by the number of applications—, in the annotation of functions —where it is quadratic because of probing—, and in fixpoints where it can reach a cubic cost. We have tried the algorithm with actual definitions of typical Eden skeletons (Peña & Segura, 2001b). For files of 3.000 net lines and 80 seconds of compilation time in a SUN 4 250 MHz Ultra Sparc-II, the analysis adds an overhead in the range of 0.5 to 1 second, i.e. less than 1 % overhead.

5 A Proof of Correctness

In this section we show the relation between the analyses and prove that they are correct with respect to the approximated semantics defined in Section 3.2. The

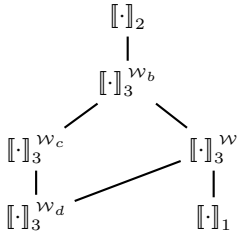

Fig. 19. A hierarchy of analyses

proofs of the propositions shown in this section can be found as supplementary material in the web page of this journal.

5.1 A hierarchy of analyses

Two non-determinism analyses have been presented to determine when an Eden expression is sure to be deterministic and when it may be non-deterministic. We have also developed another analysis $[\![\cdot]\!]_1$, presented in (Peña & Segura, 2001a). It was efficient (linear) but not very powerful. We first developed $[\![\cdot]\!]_1$ and then $[\![\cdot]\!]_2$. Analysis $[\![\cdot]\!]_3$ was intended to be an intermediate analysis, a compromise between power and efficiency. In (Peña & Segura, 2001b) we mentioned other possible widening operators ($\mathcal{W}_b$, $\mathcal{W}_c$ and $\mathcal{W}_d$) and their relation with $\mathcal{W}$. The main difference between them lies in their treatment of the tuples, in the arguments and/or in the result of the functions, either as indivisible entities or componentwise.

In (Peña & Segura, 2001b), the three analyses were formally related so that they become totally ordered by increasing cost and precision. In Figure 19 we illustrate the relation between $[\![\cdot]\!]_1$, $[\![\cdot]\!]_2$ and some variants of $[\![\cdot]\!]_3$.

The example shown in Figure 5 can be used to clarify the difference in power between $[\![\cdot]\!]_1$, $[\![\cdot]\!]_3^{\mathcal{W}}$ and $[\![\cdot]\!]_2$. By applying the analyses definitions we obtain that $[\![e]\!]_1\ \rho = (n, n) \sqsupset [\![e]\!]_3^{\mathcal{W}} \rho = (n, d) \sqsupset [\![e]\!]_2\ \rho = (d, d)$, where ρ is the empty environment.

Here we show the relation between $[\![\cdot]\!]_2$ and $[\![\cdot]\!]_3$, and also between the variants of $[\![\cdot]\!]_3$. On the one hand, Proposition 4 tells us that the third analysis is less precise than the second one. This is true for any variant of the third analysis, and in particular for the one we have described. On the other hand, Proposition 5 tells us that given two comparable widening operators, the corresponding variants of the third analysis are also comparable. In (Peña & Segura, 2001b) it was also shown that the first analysis is only a safe approximation to those variants of the third analysis satisfying a property (in particular $[\![\cdot]\!]_3^{\mathcal{W}}$ satisfies it).

Proposition 4
Let $\mathcal{W}'_t : D_{2t} \to D_{2t}$ be a widening operator for each type t. Given ρ_2 and ρ_3 such that for each variable $v :: t_v$ $\rho_2(v) \sqsubseteq \rho_3(v)$, then for each expression $e :: t_e$, $[\![e]\!]_2\ \rho_2 \sqsubseteq [\![e]\!]_3^{\mathcal{W}'} \rho_3$.

$$
\begin{array}{l}
det_K(s) = unit(s) \\
\quad \text{where} \\
\quad unit(\{\bot\}) = true \\
\quad unit(\{z, \bot\}) = true \\
\quad unit\_ = false \\
det_{(t_1,\ldots,t_m)}((s_1,\ldots,s_m)) = \bigwedge_{i=1}^{m} det_{t_i}(s_i) \\
det_T(s) = \left\{ \begin{array}{ll} \bigwedge_{i=1}^{m} det_{t_i}(\sqcup\{s_i \mid C\ s_1 \ldots s_m \in s, s_i :: t_i\}) & \text{if } one(s) \\ false & \text{otherwise} \end{array} \right. \\
\quad \text{where} \quad one(s) = (s = \{\bot\}) \vee (\exists C. \forall s' \in s. s' \neq \bot \Rightarrow s' = C\ s_1 \ldots s_m) \\
det_{t_1 \to t_2}(f) = \forall s \in A_{t_1}. det_{t_1}(s) \Rightarrow det_{t_2}(f(s))
\end{array}
$$

Fig. 20. Semantic definition of determinism

Proposition 5
Let $\mathcal{W}'_t$, $\mathcal{W}''_t$ be two widening operators for each type t. Let ρ_3, ρ'_3 such that for each variable $v :: t_v$, $\rho_3(v) \sqsubseteq \rho'_3(v)$. If for each type t, $\mathcal{W}'_t \sqsubseteq \mathcal{W}''_t$, then for each expression $e :: t_e$, $[\![e]\!]_3^{\mathcal{W}'} \rho_3 \sqsubseteq [\![e]\!]_3^{\mathcal{W}''} \rho'_3$.

Both propositions can be proved by structural induction on e.

Now we prove the correctness of $[\![\cdot]\!]_2$ with respect to Eden denotational semantics. The previous results lead us to the correctness of the whole hierarchy of analyses with respect to Eden semantics.

5.2 Capturing the determinism meaning

5.2.1 Deterministic values

In order to establish the correctness predicate we need first to define the semantic property we want to capture, that is the determinism of an expression. In Figure 20 the boolean functions det_t are defined. Given $s \in A_t$, $det_t(s)$ tells us whether s is a deterministic value or not. A value of type K is deterministic if it is a set with at most one element different from $\bot$ (as $\bot$ belongs to each $s \in A_K$), which is established by the function *unit*. A tuple is deterministic if each component is deterministic. A constructed value $s \in A_T$ is deterministic if its elements different from $\bot$ (again $\bot$ belongs to each $s \in A_T$) have the same constructor, which is established by the function *one*, and additionally the least upper bound of the values in each component is deterministic. For example, values s_1, s_2 and s'_2 defined in Section 3.2.3 are non-deterministic: the first one because it has two different constructors, and the other two because the least upper bound of the first component, $\{0, 1, \bot\}$, is non-deterministic. The definition of det_t in Figure 20 and the propositions below assume that there are not algebraic infinite values. This is not a severe restriction as processes communicating infinite values will not terminate and Hoare powerdomains ignores non-termination ($\bot$ is included in all values).

Finally, a function is deterministic if given a deterministic argument it produces a deterministic result.

Let us note that this semantical definition of determinism characterizes a possibly non-terminating single value expression as being deterministic. This is in accordance

$$\alpha_t : A_t \to D_{2t}$$
$$\alpha_K(s) = \begin{cases} d & \text{if } det_K(s) \\ n & \text{otherwise} \end{cases}$$
$$\alpha_{(t_1,\ldots,t_m)}((s_1,\ldots,s_m)) = (\alpha_{t_1}(s_1),\ldots,\alpha_{t_m}(s_m))$$
$$\alpha_T(s) = \begin{cases} d & \text{if } det_T(s) \\ n & \text{otherwise} \end{cases}$$
$$\alpha_{t_1\to t_2}(f) = \lambda z \in D_{2t_1}.\bigsqcup_{s_1\in\Gamma_{t_1}(z)} \alpha_{t_2}(f(s_1))$$

$$\Lambda_t : \mathcal{P}(A_t) \to D_{2t}$$
$$\Lambda_t(S) = \bigsqcup_{s\in S} \alpha_t(s)$$

Fig. 21. Abstraction function

$$\Gamma_t : D_{2t} \to \mathcal{P}(A_t)$$
$$\Gamma_K(b) = \begin{cases} \{s \in A_K \mid unit(s)\} & \text{if } b = d \\ \mathcal{P}(A_K) & \text{if } b = n \end{cases}$$
$$\Gamma_{(t_1,\ldots,t_m)}((z_1,\ldots,z_m)) = \{(s_1,\ldots,s_m) \mid \alpha_{t_i}(s_i) \sqsubseteq z_i \forall i \in \{1..m\}\}$$
$$\Gamma_T(b) = \begin{cases} \{s \in A_T \mid det_T(s)\} & \text{if } b = d \\ \mathcal{P}(A_T) & \text{if } b = n \end{cases}$$
$$\Gamma_{t_1\to t_2}(f^\#) = \{f \in A_{t_1\to t_2} \mid \forall s \in A_{t_1}.\alpha_{t_2}(f(s)) \sqsubseteq f^\#(\alpha_{t_1}(s))\}$$

Fig. 22. Concretisation function

with the Hoare powerdomain semantics we have adopted producing Scott-closed sets: where the actual semantics produces a single value, our approximate semantics produces a non-singleton set because it always includes $\perp$. That is, predicate det_t characterizes determinism up to non-termination. Notice also that, if we eliminate $\perp$ in the definitions of *unit* and *one*, then predicate det_t characterizes real singleton sets in the basic type, tuples and algebraic type cases; and functions mapping single values into single values in the functional type case. Predicates det_t have some properties (Segura & Peña, 2003b) we do not show here.

5.2.2 Abstraction and concretisation functions

Now we define the abstraction Λ_t and concretisation Γ_t functions that relate the abstract and concrete domains, following the ideas in (Burn *et al.*, 1986).

The function Λ_t is just an extension of a function α_t to Hoare sets by applying it to each element of the set and taking the least upper bound. So α_t will also be called abstraction function. With this function, defined in Figure 21, we want to abstract the determinism behaviour of the concrete values. It loses information, i.e. several concrete values may have the same abstract value. In Figure 22 the concretisation function is defined. For each abstract value, it returns all the concrete values that can be approximated by that abstract value. They are mutually recursive. We will prove that Λ_t and Γ_t are a Galois connection, which implies that for each concrete value there may be several abstract approximations but there exists only one best (least) approximation. This is a crucial property in the correctness proof.

A value of type K or T is abstracted to d only if it is deterministic. The abstraction of a tuple is the tuple of the abstractions. The abstraction of a function f of type $t_1 \to t_2$ is a little more involved. It is an abstract function taking an argument $z \in D_{2t_1}$. Such z represents several concrete values $s_1 \in \Gamma_t(z)$ whose abstract images are $\alpha_{t_2}(f(s_1))$. So the abstraction of the result is the least upper bound of these abstract images.

The concretisation function is defined in such a way that it builds a Galois connection with Λ_t. The concretisation Γ_t of the basic abstract value d is the set of deterministic concrete values (when $t = K$ or $t = T$), and the concretisation of n is the whole corresponding Hoare powerdomain ($\mathcal{P}(A_K)$ or $\mathcal{P}(A_T)$). The concretisation of a tuple is the set of tuples whose components are abstracted to the abstract components. The concretisation of an abstract function $f^\#$ is again more involved. It is a set of concrete functions such that the abstraction of its behaviour on a concrete argument s is safely approximated (it is less or equal than) by the behaviour of the abstract function on the abstraction of the argument.

It can easily be proved that Γ_t is well defined, i.e. it produces downwards closed sets of concrete values. It can also be proved that for each type t, functions α_t, Λ_t and γ_t are continuous. Both things are shown in (Segura & Peña, 2003b).

The most important result in this section is that Λ_t and Γ_t are a Galois connection (i.e. $\Lambda_t \cdot \Gamma_t \sqsubseteq id_{D_{2_t}}$ and $\Gamma_t \cdot \Lambda_t \sqsupseteq id_{\mathcal{P}_{(A_t)}}$), which is equivalent to the following proposition, that will be intensively used in the correctness proof.

Proposition 6
For each type t, $z \in D_{2t}$, and $s \in A_t$: $s \in \Gamma_t(z) \Leftrightarrow \alpha_t(s) \sqsubseteq z$.

This proposition can be proved by structural induction on t.

Finally we present an interesting property that only holds when the concrete domains of basic and algebraic types have at least two elements different from $\perp$. In the following proposition we show that α_t is surjective, i.e. each abstract value is the abstraction of a concrete value, which in particular belongs to the concretisation of that abstract value. This means that Λ_t and Γ_t are a Galois insertion ($\Lambda_t \cdot \Gamma_t = id_{D_{2_t}}$).

Proposition 7
If all $[\![K]\!]$ and $[\![T]\!]$ have at least two elements different from $\perp$, then for each type t and $z \in D_{2t}$, there exists $s \in \Gamma_t(z)$ such that $\alpha_t(s) = z$.

This can also be proved by structural induction on t. If the theorem hypothesis about $[\![K]\!]$ and $[\![T]\!]$ does not hold then it is easy to see that all the concrete values are abstracted to d and none to n. In fact we are avoiding the *Unit* type. However this property is not necessary in the correctness proof.

5.2.3 A proof of partial correctness

Now we prove that $[\![\cdot]\!]_2$ is correct with respect to the denotational semantics: when the analysis tells that an expression is deterministic, then the concrete value produced by the denotational semantics is semantically deterministic. Otherwise we do

not know anything about it. We have to formally describe this intuition. On the one hand, we said in Section 4 that $\mu_t(d)$ is the best safe approximation to d in a given domain, so the analysis tells us that an expression is deterministic when its abstract value is less or equal than $\mu_t(d)$. On the other hand the semantical determinism of a concrete value has been established by predicate det_t. So, the main correctness result is expressed as follows.

Theorem 8
Let ρ and ρ_2 be two environments, such that for each variable $x :: t_x$, $\alpha_{t_x}(\rho(x)) \sqsubseteq \rho_2(x)$. Then for each expression $e :: t$:

$$[\![e]\!]_2\ \rho_2 \sqsubseteq \mu_t(d) \Rightarrow det_t([\![e]\!]\ \rho)$$

Notice that this only proves the partial correctness of the analysis with respect to the actual semantics of Eden. This (not formally defined) semantics only produces non-singleton sets when expression e contains at least one occurrence of `merge`. If expression e completely terminates, then we can ignore the undefined values in $[\![e]\!]\ \rho$ and then $det_t([\![e]\!]\ \rho)$ amounts to saying that $[\![e]\!]\ \rho$ consists of a single value, i.e. e is deterministic in the actual semantics sense.

The theorem is proved in two parts written as Propositions 9 and 10, shown below. The first one tells us that all the values whose abstraction is below $\mu_t(d)$ are semantically deterministic. The second one asserts that the analysis is an upper approximation to the abstraction of the concrete semantics. The theorem is then immediately obtained.

Proposition 9
For each type t, and $s \in A_t$: $\alpha_t(s) \sqsubseteq \mu_t(d) \Leftrightarrow det_t(s)$.

This proposition can be proved by structural induction on t. We need Proposition 6 and also some properties satisfied by ϕ_t and μ_t, proved in (Peña & Segura, 2001b). In particular we need the fact that they are a Galois insertion.

Proposition 10
Let ρ and ρ_2 be two environments, such that for each variable $x :: t_x$, $\alpha_{t_x}(\rho(x)) \sqsubseteq \rho_2(x)$. Then for each expression $e :: t$: $\alpha_t([\![e]\!])\ \rho \sqsubseteq [\![e]\!]_2\ \rho_2$.

This proposition can be proved by structural induction on e. We need Propositions 6 and 9, and some properties satisfied by ϕ_t and μ_t, proved in (Peña & Segura, 2001b). Additionally we need to prove that α_t reflects the bottom element, and that the denotational semantics we have defined is monotone with respect to the environments. Both things are proved by structural induction in (Segura & Peña, 2003b).

6 Conclusions and Related Work

We have not found any previous analyses for the non-determinism problem in the literature. Our analysis $[\![\cdot]\!]_2$ is based on abstract interpretation in the style of Burn, Hankin and Abramsky (1986), where functions are interpreted as abstract functions.

There, a strictness analysis was presented. Both analyses can be seen as particular cases of what is commonly known as *dependency analysis* (Jones & Nielson, 1995).

In this broad group many different analyses fall. The general idea is to study different forms of dependencies between the program variables. In the logic programming field, *groundness analysis*, *finiteness analysis* and *suspension analysis* (Armstrong *et al.*, 1998) are some examples. For instance, the groundness analysis tries to capture the groundness dependencies between the logic variables. In the functional languages field, the *binding time analysis* (BTA) also falls into this category. Here, the analysis distinguishes those variables being *static* (S), or known at compile time, from those being *dynamic* (D).

Several techniques have been used in these analyses: abstract interpretation (Armstrong *et al.*, 1998), projections based analysis (Launchbury, 1991; Mogensen, 1989), and type based analysis (Mossin, 1994).

In our non-determinism analysis we study how the result of an expression depends on the non-determinism information collected about its free variables. However, the three analyses (strictness, non-determinism and binding time) are different. The basic abstract domains are in the three cases two-point domains: $\bot \sqsubseteq \top$ in the strictness analysis, $S \sqsubseteq D$ in the binding time analysis and $d \sqsubseteq n$ in the non-determinism analysis. BTA is essentially a dual problem to strictness analysis: where strictness analysis finds how much of the parameters of a function is needed to produce the result, BTA finds how much of the result will be known at compile time, given which parts of the parameters are known. Comparing the three analyses the interpretation of the constants are different: the abstract values of $head(merge\#[[0],[1]]))$ and 1 are respectively $\top$, S, n and $\top$, S, d. Also the interpretation of the analysis results may be different even when the abstract value is the same. As an example, let $f :: (Int \to Int) \to Int$ be $f = \lambda g.g\ (head(merge\#[[0],[1]]))$. In the strictness analysis, the abstract function for f is $\lambda g \in [\mathbf{2} \to \mathbf{2}].g\ \top$ while in the non-determinism analysis it is $\lambda g \in [Basic \to Basic].g\ n$. Although the abstract functions are basically the same, the strictness analysis tells us that f is strict in its argument, i.e. $f^{\#}\ (\lambda z.\bot) = \bot$, while the non-determinism analysis tells us that it may be non-deterministic, i.e. $f^{\#}\ (\lambda z.z) = n$.

Following some ideas in (Peyton Jones & Partain, 1993) about how to define a widening operator by using a signature to represent functions, an intermediate analysis has been developed that is a little less powerful but much less expensive than the second one. It needs polynomial time, and compared to the second analysis, it only loses information in the fixpoints. It has been implemented in Haskell and tested with many examples.

Regarding algebraic types, our analyses only distinguish between two posible values: deterministic and non-deterministic. It is interesting to wonder whether it would have given more precise results to use richer abstract domains in the style of the 4-points domains for lists of Wadler (1987). For instance, we could distinguish four cases for lists: non-deterministically generated lists with non-deterministic or deterministic elements inside, and deterministally generated lists with non-deterministic or deterministic elements inside. Let us respectively call them $N\ n, N\ d, D\ n$ and $D\ d$. Our current analysis collapses the first three values into

one. For many functions, such as the sum or the head of a list, this is adequate as they will not distinguish between the first three possibilities. The result will be non-deterministic in the three cases. Only functions such as `length`, taking into account only the list structure, will distinguish between the first two cases and the third one. We have considered that the gain in precision in some cases would not compensate the extra complexity.

We have proved the correctness of a whole hierarchy of non-determinism analyses. In order to do this, we have defined first a denotational semantics for Eden where non-determinism is represented. We have chosen to use a plural semantics in which non-deterministic choices for variables are deferred as much as possible. A semantics nearer to the actual one (within a single process) would have been a singular one in which environments map variables to single values. This would reflect the fact that non-deterministic choices are done at binding evaluation time instead of at each variable occurrence. For instance, a let-bound variable will get its value the first time it is evaluated and this value will be shared thereafter by all its occurrences. In order to consider all the possible values the variable can have, we build one environment for each of them:

$$[\![\textbf{let } v = e \textbf{ in } e']\!]\ \rho = \bigsqcup_{z \in [\![e]\!]\ \rho} [\![e']\!]\ \rho[v \mapsto z]$$

We would use the same approach for case-bound and lambda-bound variables. We have tried to define this singular semantics and things go wrong when trying to give semantics to mutually recursive definitions. The traditional fixpoint computation by using Kleene's ascending chain gives a semantics more plural than expected. For instance, in the definition

$$\begin{array}{ll} \textbf{letrec} & f = head(merge_{Int \to Int}\ [g]\ [\lambda x.0]) \\ & g = head(merge_{Int \to Int}\ [f]\ [\lambda x.1]) \\ \textbf{in } (f, g) & \end{array}$$

Kleene's ascending chain will compute the following set of possible environments:

$$\begin{array}{ll} \overline{\rho} = \{ & \{f \mapsto \lambda x.\{\bot\},\ g \mapsto \lambda x.\{\bot\}\}, \\ & \{f \mapsto \lambda x.\{0\}^*,\ g \mapsto \lambda x.\{1\}^*\}, \\ & \{f \mapsto \lambda x.\{0\}^*,\ g \mapsto \lambda x.\{0\}^*\}, \\ & \{f \mapsto \lambda x.\{1\}^*,\ g \mapsto \lambda x.\{1\}^*\}, \\ & \{f \mapsto \lambda x.\{1\}^*,\ g \mapsto \lambda x.\{0\}^*\}\ \} \end{array}$$

However, the lazy evaluation of the expression will never produce the fifth possibility. In (Søndergaard & Sestoft, 1992) a singular semantics for a small non-deterministic recursive functional language was defined. The problem with fixpoints did not arise there because the language was extremely simple: only one recursive binding was allowed in the program and this had to bind a lambda abstraction. Additionally, the language was only first-order. The problem arises when there are at least two mutually recursive bindings to non normal-form expressions. In order to define a real singular semantics, we think that an operational approach should be taken, similar to that of Hughes and Moran (1995). In this way, the actual

lazy evaluation with its updating of closures and sharing of expressions could be appropriately modeled.

For our purposes this development closes our original problem. Although the main motivation for developing the analyses has been the correct compilation of our Eden programs, we think that the analyses could also be useful for other higher-order functional languages with non-deterministic constructs. A possible utility could be the annotation of the parts of the text where equational reasoning is still possible.

References

Armstrong, T., Marriott, K., Schachte, P., & Søndergaard, H. (1998). Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming*, **31**(1), 3–45.

Breitinger, S., Loogen, R., Ortega-Mallén, Y., & Peña, R. (1997). The Eden Coordination Model for Distributed Memory Systems. *Pages 120–124 of: Workshop on High-level Parallel Programming Models, HIPS'97*. IEEE Computer Science Press.

Breitinger, S., Klusik, U., Loogen, R., Ortega-Mallén, Y., & Peña, R. (1998a). DREAM: the Distributed Eden Abstract Machine. *Pages 250–269 of: Implementation of Functional Languages, IFL'97*. LNCS 1467. Springer-Verlag.

Breitinger, S., Loogen, R., Ortega-Mallén, Y., & Peña, R. (1998b). *Eden: Language Definition and Operational Semantics*. Technical Report, Bericht 96-10. Revised version 1.998, Philipps-Universität Marburg, Germany.

Burn, G. L., Hankin, C. L., & Abramsky, S. (1986). The Theory of Strictness Analysis for Higher Order Functions. *Pages 42–62 of: Programs as Data Objects*. LNCS, vol. 217. Springer-Verlag.

Cousot, P., & Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixed Points. *Pages 238–252 of: Proceedings of the 4th ACM Symposium on Principles of Programming Languages*. ACM.

Cousot, P., & Cousot, R. (1979). Systematic Design of Program Analysis Frameworks. *Pages 269–282 of: Conference Record of the 6th Annual ACM Symposium on Principles on Programming Languages*. ACM.

Hankin, C., & Hunt, S. (1992). Approximate Fixed Points in Abstract Interpretation. *Pages 219–232 of:* Krieg-Brückner, B. (ed), *4th European Symposium on Programming, ESOP '92*. LNCS, vol. 582. Springer, Berlin.

Henderson, P. (1982). Purely Functional Operating Systems. *Pages 177–191 of: Functional Programming and its Applications: An Advanced Course*. Cambridge University Press.

Hidalgo, M., & Ortega, Y. (2003). Continuation Semantics for Parallel Haskell Dialects. *Pages 303–321 of: First Asian Symposium on Programming Languages and Systems, APLAS'03*. LNCS, vol. 2895. Springer-Verlag.

Hughes, J. (1995). The Design of a Pretty-printing Library. *Pages 53–96 of:* Jeuring, J., & Meijer, E. (eds), *Advanced Functional Programming*. LNCS, vol. 925. Springer Verlag.

Hughes, J., & Moran, A. (1995). Making Choices Lazily. *Pages 108–119 of: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA'95*. ACM Press.

Hughes, R. J. M., & O'Donnell, J. (1990). Expressing and Reasoning About Non-Deterministic Functional Programs. *Pages 308–328 of: Functional Programming: Proceedings of the 1989 Glasgow Workshop*. Springer-Verlag.

Jones, N. D., & Nielson, F. (1995). Abstract Interpretation: a Semantics-Based Tool

for Program Analysis. *Handbook of Logic in Computer Science, Volume 4.* Oxford University Press. 527–629.

Launchbury, J. (1991). Strictness and Binding-Time Analyses: Two for the Price of One. *Pages 80–91 of: Proceedings of the Conference on Programming Language Design and Implementation, PLDI'91.*

McCarthy, J. (1963). Towards a Mathematical Theory of Computation. *Pages 21–28 of: Proceedings of IFIP Congress 62.* Amsterdam: North-Holland.

Mogensen, T. (1989). Binding Time Analysis for Polymorphically Typed Higher Order Languages. *Pages 298–312 of: Proceedings of the International Joint Conference on Theory and Practice of Software Development, TAPSOFT'89.* LNCS, vol. 352. Springer-Verlag.

Mossin, C. (1994). *Polymorphic Binding-Time Analysis.* M.Phil. thesis, DIKU, University of Copenhagen, Denmark.

Peña, R., & Segura, C. (2001a). Non-Determinism Analysis in a Parallel-Functional Language. *Pages 1–18 of: Selected Papers of the 12th International Workshop on Implementation of Functional Languages, IFL'00.* LNCS, vol. 2011. Springer-Verlag.

Peña, R., & Segura, C. (2001b). *Three Non-determinism Analyses in a Parallel-Functional Language.* Technical Report 117-01, Dep. Sistemas Informticos y Programacin, Universidad Complutense de Madrid, Spain. (http://dalila.sip.ucm.es/miembros/clara/publications.html).

Peña, R., & Segura, C. (2002). A Polynomial Cost Non-Determinism Analysis. *Pages 121–137 of: Selected papers of the 13th International Workshop on Implementation of Functional Languages, IFL'01.* LNCS, vol. 2312. Springer-Verlag.

Peyton Jones, S. L., & Clack, C. (1987). Finding fixpoints in abstract interpretation. *Chap. 11, pages 246–265 of:* Abramsky, S., & Hankin, C. (eds), *Abstract Interpretation of Declarative Languages.* Ellis-Horwood.

Peyton Jones, S. L., & Partain, W. (1993). Measuring the effectiveness of a simple strictness analyser. *Pages 201–220 of: Glasgow Workshop on Functional Programming 1993.* Workshops in Computing. Springer-Verlag.

Peyton Jones, S. L., & Santos, A. L. M. (1998). A Transformation-based Optimiser for Haskell. *Science of Computer Programming*, **32**(1-3), 3–47.

Peyton Jones, S. L., Hall, C. V., Hammond, K., Partain, W. D., & Wadler, P. L. (1993). The Glasgow Haskell Compiler: A Technical Overview. *Pages 249–257 of: Joint Framework for Inf. Technology, Keele, DTI/SERC.*

Segura, C., & Peña, R. (2003a). Correctness of Non-determinism Analyses in a Parallel-Functional Language. *Selected papers of the 15th International Workshop on Implementation of Functional Languages, IFL'03.* LNCS, vol. To appear. Springer-Verlag.

Segura, C., & Peña, R. (2003b). *Correctness of Non-determinism Analyses in a Parallel-Functional Language.* Technical Report 131-03, Dep. Sistemas Informticos y Programacin, Universidad Complutense de Madrid, Spain. (http://dalila.sip.ucm.es/miembros/clara/publications.html).

Søndergaard, H., & Sestoft, P. (1990). Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, **27**(6), 505–517.

Søndergaard, H., & Sestoft, P. (1992). Non-Determinism in Functional Languages. *Computer Journal*, **35**(5), 514–523.

Wadler, P. L. (1987). Strictness analysis on non-flat domains (by abstract interpretation). *Chap. 12, pages 266–275 of:* Abramsky, S, & Hankin, C (eds), *Abstract Interpretation of Declarative Languages.* Ellis-Horwood.