Chapter 7
Strictness analysis

Programs in lazy functional languages are evaluated lazily. The order of evaluation corresponds to normal order reduction of an equivalent λ-term in the λ-calculus. It also corresponds to evaluation according to the functional strategy for an equivalent FTRS or FGRS. Each of these evaluation orders is normalizing for its corresponding system.

However, in practice, lazy evaluation has as a disadvantage that this order of evaluation is not very efficient. In many cases, a lot of efficiency (both in space and time) can be gained when eager evaluation is chosen instead of lazy evaluation, both for sequential (see Part 4) as well as for parallel implementations (see Part 5). Eager evaluation corresponds in the λ-calculus to applicative order reduction, while in FTRSs and FGRSs it corresponds to innermost strategies.

As shown in the previous part of this book one cannot simply change the overall evaluation order from lazy to eager. The normal order reduction strategy as well as the functional strategy are both hypernormalizing (see Part 2). So if the corresponding evaluation orders are diluted with an arbitrary *finite* number of reduction steps, the result will always be the same. The problem is that by switching to eager evaluation the default strategy might be diluted with an *infinite* number of reduction steps. This can lead to a change in the termination behaviour of the program. So the evaluation order can only be changed at those places where it is certain that the termination behaviour of the program is not affected. The arguments in which a function is strict have this property.

A function is *strict* in a certain argument if the (eager) evaluation of that argument does not change the termination behaviour of the program. *Strictness analysis* is a compile-time analysis of a program that is used to tell whether or not a function is strict in its arguments. This *strictness information* is used to change the default reduction order at certain points in the program such that the performance of that program is improved. Whether or not a function is strict in a certain argument is, of course, in general undecidable. So strictness analysers use some kind of *approximation* method. Such an approximation method must be safe, i.e. the termination behaviour of a program has to remain unchanged.

There are various kinds and levels of approximations and therefore also various kinds of strictness analysers. Good strictness analysers are complex programs: for a good approximation it is not sufficient just to look at the syntactic structure of a program, as is the case with type assignment systems. For an accurate analysis some kind of approximation of the run-time behaviour of a program has to be performed at compile-time. This technique is known as *abstract interpretation*.

Strictness is explained in Section 7.1. A general introduction in abstract interpretation is given in Section 7.2. Some necessary basic formal knowledge about domain theory is presented in Section 7.3. Then, strictness analysis for lazy functional languages based on abstract interpretation is explained. This technique, introduced by Mycroft (1981), can only deal with first-order functions and flat domains (Section 7.4). In Section 7.5 we explain how one can deal with recursive functions and higher order functions (Burn *et al.*, 1985). The analysis for lists (Wadler, 1987) is treated in Section 7.6.

Sections 7.7 and 7.8 explore a related approximation method that can deal with arbitrary data structures. This method is known as *abstract reduction* (Nöcker, 1988; 1993). An analyser based on abstract reduction is incorporated in the Clean System (see Chapter 8).

7.1 Strictness

A function is **strict** in a certain argument if the (eager) evaluation of that argument does not change the termination behaviour of the program. Formally, this can be defined as follows: a function f of arity n is *strict* in its *i*th argument iff

$$f\ x_1 \dots x_{i-1} \perp x_{i+1} \dots x_n \quad = \quad \perp$$

for all possible values x_j ($j \neq i$). In this definition $\perp$ (bottom) stands for any non-terminating expression.

A **strict argument** can arise in two situations. In the first place it can be the case that an argument is **needed** (see Chapter 4), i.e. it certainly has to be reduced in each possible application of the function. So when an argument is needed it can be evaluated in advance without changing the termination behaviour of the function. This property is very important for changing lazy evaluation into eager evaluation. In the second place an argument is strict when the function represents an infinite computation, even disregarding the argument.

Example of strictness due to neededness:

```
g x   =  x + 1
```

The function g is strict in x because the value of x is needed for the addition. Hence, x can be evaluated safely before applying g. If the evaluation of x does not terminate, the lazy strategy would also end up in non-termination because x would be evaluated anyway. So indeed: g $\perp = \perp$.

Example of strictness due to non-termination:

```
g x   =  g x
```

In spite of the fact that x is not needed in g, it is still safe to evaluate it before applying g. The termination properties remain the same: g $\perp = \perp$.

Strictness analysis is a compile-time analysis of the strictness properties of a program. Strictness is in general undecidable. So inherently strictness analysers use some kind of *approximation* method which is safe, i.e. the termination behaviour of a program has to remain unchanged. Two analysis methods are described in this chapter: *abstract interpretation* and *abstract reduction*.

7.2 Abstract interpretation

The technique of abstract interpretation is used to perform a compile-time analysis of certain properties of a complex computation. It is often used to perform strictness analysis, but the technique can also be used to analyse many other properties, such as the complexity of a program, or to predict whether a computation is shared or not. The basic idea of the method is that the *standard interpretation* at run-time (the ordinary, possibly non-terminating, evaluation of the program) is approximated by a terminating *abstract interpretation* at compile-time. To make this compile-time evaluation possible, only those evaluation aspects are regarded that are needed for the derivation of the desired property. So the kind of analysis is highly dependent upon the property of interest.

7.2.1 Rule of signs

As an example the well-known ‘rule of signs’ is presented. Assume that one is interested in the sign of the result of a multiplication of integer values. The sign of the result can be calculated in two ways: either by first calculating the result and then taking the sign or by first taking the sign of the arguments and then deducing the sign of the result.

> Take the following computation:
>
> –33 * 728 * –301 → +7231224
>
> The obvious way to find out the sign of the result is by simply calculating the result, +7231224, which obviously is positive. The other method only looks at the signs of the arguments:
>
> (–) * (+) * (–) → (+)
>
> The sign of the result follows by applying a special 'multiplication' on signs.

Formally, the first method starts with calculating the result in the domain Z of numbers as usual (with the standard interpretation). Then, an abstraction of the result from that domain to an abstract domain $Z^{\#}$ of representations of signs is made.

> The first method calculates the sign after performing a standard multiplication.
>
> abs +7231224 → plus
>
> where the abstraction function abs and the domains Z and $Z^{\#}$ are defined as:

```
Z     = { …, –2, –1, 0, 1, 2, … }
Z#    = {minus, zero, plus}

abs:: Z → Z#
abs x   = plus,    if x > 0
        = minus,   if x < 0
        = zero,    otherwise
```

The second method first abstracts numbers to signs with the abstraction function from Z to $Z^{\#}$. Then, instead of the standard multiplication in Z an abstract multiplication is performed in $Z^{\#}$.

Abstract multiplication is an abstract function from $Z^{\#}$ to $Z^{\#}$. Do not confuse such an abstract function with the abstraction function that is defined from Z to $Z^{\#}$.

> The second method uses an abstract multiplication *#: an abstract version (in prefix form) of the standard multiplication, working on abstracted values.

```
*# (abs –33) (*# (abs 728) (abs –301))
→   *# minus (*# plus minus)  →   *# minus minus  →   plus
```

where the abstract multiplication $*^{\#}$ is defined as:

$*^{\#}$:: $Z^{\#} \times Z^{\#} \rightarrow Z^{\#}$

$*^{\#}$	plus	plus	=	plus
$*^{\#}$	plus	zero	=	zero
$*^{\#}$	zero	plus	=	zero
$*^{\#}$	plus	minus	=	minus
$*^{\#}$	minus	plus	=	minus
$*^{\#}$	minus	zero	=	zero
$*^{\#}$	zero	minus	=	zero
$*^{\#}$	zero	zero	=	zero
$*^{\#}$	minus	minus	=	plus

$$\begin{tikzcd} Z \times Z \arrow[r, "*"] \arrow[d, "\text{abs}"] & Z \arrow[d, "\text{abs}"] \\ Z^{\#} \times Z^{\#} \arrow[r, "*^{\#}"] & Z^{\#} \end{tikzcd}$$

Figure 7.1 Two ways to calculate the sign of a multiplication.

So the sign of the result of a multiplication can be obtained in two ways (see Figure 7.1). The first way corresponds to the arrows going from $Z \times Z$ to Z (using the *standard interpretation*) and then from Z to $Z^{\#}$ (an abstraction from Z). The second way corresponds to the arrows going from $Z \times Z$ to $Z^{\#} \times Z^{\#}$ (an abstraction from $Z \times Z$) and then from $Z^{\#} \times Z^{\#}$ to $Z^{\#}$ (using an *abstract interpretation*). Both ways reach the same property, represented by an element of $Z^{\#}$. But, with abstract interpretation the result can be obtained more easily and faster.

In the running example the result of both calculation methods is the same:

abs (x * y) = $*^{\#}$ (abs x) (abs y)

So the abstract interpretation can replace the standard interpretation.

We have shown that abstract interpretation can be used to derive signs of multiplicative expressions. However, not all properties can be derived via abstract interpretation.

Assume that we want to calculate the sign of the result of integer addition via abstract interpretation. Then, an abstract function $+^{\#}$ has to be defined:

$+^{\#}$:: $Z^{\#} \times Z^{\#} \rightarrow Z^{\#}$

$+^{\#}$ plus plus = plus

```
+# plus zero  =  plus
+# plus minus =  ??
…
```

The third alternative is problematic. The sign of the addition of a positive and a negative value can only be computed by actually doing the computation. In the abstract domain the result is unknown. This can be expressed by adding a new element to $Z^{\#}$: unknown. The definition of $+^{\#}$ can now be completed. One also has to take care that the new value unknown can appear as an argument of $+^{\#}$ as well.

```
+# plus minus        =  unknown
+# plus unknown      =  unknown
+# zero zero         =  zero
+# zero unknown      =  unknown
…
```

When the outcome of the abstract addition yields the value unknown nothing can be said of the sign of the actual result in the standard interpretation.

In the abstract domain various approximations are possible. If a value y in the abstract domain represents many elements in the concrete domain, it is a coarser approximation than an abstract value x representing fewer of these concrete elements. The operator ≤ (less than or equal) expresses this ordering. If $x \leq y$ then x represents *fewer* concrete *elements* than y. So x contains *more information* than y. The inverse abstraction function abs^{-1} yields, given a value $v^{\#}$ in the abstract domain, the set of all corresponding values v in the concrete domain for which $abs^{-1}\ v^{\#} = v$. Formally, the **abstract ordering** is defined as $x \leq y$ iff $abs^{-1}\ x \leq abs^{-1}\ y$.

All integers in the concrete domain Z correspond to the value unknown, whereas, for instance, only the positive integers correspond to the abstract value plus. Hence the value unknown holds less information than plus.

```
plus     ≤  unknown
```

Not all elements in $Z^{\#}$ contain the same amount of information. So in some cases the abstract interpretation is less informative. Therefore only an approximation of the desired property can be found. Such an approximation should not yield wrong results, i.e. be in contradiction with the results obtained by using the standard interpretation. However, being an approximation it may contain less information. The main requirement for an abstract interpretation is that it is *safe* (correct), i.e.

```
abs . f   ≤  f# . abs
```

for all elements in the concrete domain ('.' stands for function composition). This important criterion is called the **safety criterion**.

For deriving the sign for multiplication there is no difference between the outcome with abstract interpretation or without abstract interpretation, since

```
abs . *   =  *# . abs
```

For deriving the sign for addition only an approximation can be achieved:

```
abs . +  ≤  +# . abs
```

So due to the loss of information, abstract interpretation will generally only give an approximation. This does not have to be a serious problem. A decidable analysis at compile-time that can predict an undecidable run-time property of a program can only be achieved by using an approximation method.

For a further understanding of strictness analysis via abstract interpretation, a short informal introduction to domain theory is needed.

7.3 Basic domain theory

The fundamental concept in domain theory is a **semantic domain**, a set of elements grouped together because they share some common property or use. In the literature many approaches, more or less equivalent, can be found (see, for example, Markovsky (1976), Stoy (1977)). All of them are based on the concept of a **c**omplete **p**artial **o**rdering (**cpo**). A set D with an ordering relation over that set $\leq_d$ (or $\leq$ for short) is a **complete partial ordering** (D, $\leq$) if it has the following properties:

(1) It is a partially ordered set (a **poset**), i.e. $\leq$ is
- **reflexive**: $d \leq d$ for all $d \in D$;
- **transitive**: if $d \leq e$ and $e \leq f$ for $d,e,f \in D$ then $d \leq f$;
- and **anti-symmetric**: if $d \leq e$ for $d,e \in D$ and $d \neq e$, then not $e \leq d$.

(2) It is **complete**, i.e.
- D has a **least** element $\perp_d$ (or $\perp$) such that $\perp \leq d$ for all $d \in D$;
- and, for each **increasing sequence** $S = d_1 \leq d_2 \leq \ldots \leq d_n \leq \ldots$ in D, there exists a *least upper bound* $\cup S \in D$.

A **least upper bound** is defined as:
- it is an **upper bound**: $d_j \leq \cup S$ for all $d_j \in S$;
- and it is **least**: if $d_j \leq u$ for all $d_j \in S$ and some $u \in D$, then $\cup S \leq u$.

The **l**east **u**pper **b**ound (lub) is also used as an infix operator.

From now on all domains are assumed to be cpos.

A pictorial representation of a finite semantic domain is shown in Figure 7.2. All points represent elements of the domain. The arcs define the ordering. If x and y are connected and x is 'below' y in the picture, then $x \leq y$.

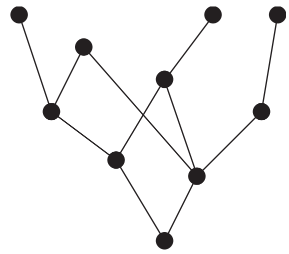

Figure 7.2 A finite cpo.

A domain $<D, \leq>$ is **flat** (Figure 7.3) iff:

- for all $a,b \in D$: if $a \leq b$ then $a = b$ or $a = \perp$.

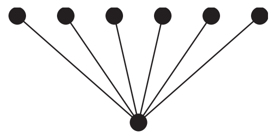

Figure 7.3 Pictorial representation of a flat domain.

Some examples of complete partial orderings:

$<\{0,1\}, \leq>$ with $0 \leq 1$. The least element is 0; for the sequence $0 \leq 1$ the least upper bound is 1. This is a flat domain.

$<N_\perp, \leq>$ with $\perp \leq n$, $n \in N_\perp$ where $N_\perp$ is the set of natural numbers extended with a bottom element: $N \cup \{\perp\}$ (also called the **lifted domain** of N). Note that the $\leq$ has nothing to do with the ordinary less than or equal: elements of N are incomparable in this domain. Further note that this is a flat domain.

$<P(\{0, 1, 2\}), \leq>$ where $P(\{0, 1, 2\})$ is the powerset of the set $\{0, 1, 2\}$ containing all subsets of this set: $\{\{\}, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \ldots , \{0, 1, 2\}\}$ where $\leq$ is the standard subset relation defined on sets.

The functions over domains are in general *monotonic* and *continuous*:

- A function f:: $D \to E$ is **monotonic** iff for all $d,e \in D$, if $d \leq e$, then $f\ d \leq f\ e$.
- A monotonic function f:: $D \to E$ is **continuous** iff for every increasing sequence S in D, $f\,(\cup\, S) = \cup\, \{f\ d_i \mid d_i \in S\}$.

If a function is continuous, it is also monotonic. For finite domains the two are equivalent. The intuitive meaning of monotonicity is that all 'nice' properties and the structure of a domain are maintained. For example, an increasing sequence stays an increasing sequence after function application. Continuity ensures that there is no surprise at least upper bounds: the least upper bound of an increasing sequence stays a least upper bound after function application.

Starting with domains new domains can be constructed. Suppose D and E are domains. Then the **product domain** $D \times E$ is defined as: $D \times E = <\{(x,y) \mid x \in D, y \in E\}, \leq_x>$. The **ordering relation** $\leq_x$ **for the product domain** is defined as: $(x,y) \leq_x (x',y')$ iff $x \leq_d x'$ and $y \leq_e y'$.

The **function domain** $D \to E$ contains all continuous functions from D to E. It is defined as: $D \to E = <\{f \mid f$ is a continuous function from D to $E\}, \leq_\to>$. The **ordering relation for the function domain** is defined as: $f \leq_\to g$ iff $f\ d \leq_e g\ d$ for all $d \in D$.

Take for example the following domain with its product domain:

```
D       =    <{0,1}, ≤> with 0 ≤ 1
D x D   =    <{(0,0), (0,1), (1,0), (1,1)}, ≤x>
```

The domain D contains only two values, 0 and 1. Therefore only four functions from D to D can be constructed:

$f_\perp\ x = 0$
$f_I\ x = x$
$f_{\sim I}\ x = \sim x$ (~ is negation)
$f_T\ x = 1$

Only the function $f_{\sim I}$ is not continuous, so the function domain $D \to D$ is (Figure 7.4):

$D \to D = <\{f_\perp, f_I, f_T\}, \leq_\to>$

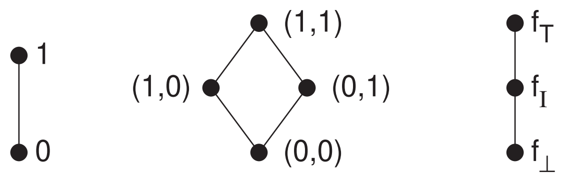

Figure 7.4 Pictorial representation of the domains D, D x D and D → D.

For continuous functions the **fixed point theorem** holds: if f:: $D \to D$ is continuous, then

- f has a **least fixed point** $d \in D$ with the following properties:
  - d is a fixed point: $f\ d = d$;
  - d is least: if $f\ e = e$, $e \in D$, then $d \leq e$.

This least fixed point d can be constructed as follows: $d = \cup \{f^n \perp \mid n \geq 0\}$. This set equals $\{\perp, f\perp, f(f\perp), \ldots\}$. So it follows that *if* the domain D is *finite*, the least fixed point can be found in finite time: just start computing the sequence $\perp, f\perp, f(f\perp), \ldots$ (which is increasing since f is continuous) until two consecutive values are the same.

7.4 Strictness analysis using abstract interpretation

In this section strictness analysis is discussed for a very restricted situation: only simple non-recursive functions are considered working on a flat domain D of integers, booleans and the like. Function bodies can only contain function applications (non-recursive, no higher order), constants (numbers, booleans etc.) and variables. So it is assumed that generally a function f is defined as:

$$\text{f} :: \text{D}^n \rightarrow \text{D}$$
$$\text{f } x_1 \ldots x_i \ldots x_n \quad = \quad \text{E}$$

To make an abstract interpretation to determine strictness two things have to be done (Figure 7.5).

$$\begin{tikzcd} \text{D}^n \arrow[r] \arrow[d, "\text{abs}"] & \text{D} \arrow[d, "\text{abs}"] \\ (\text{D}^\#)^n \arrow[r, "\text{f}^\#"] & \text{D}^\# \end{tikzcd}$$

Figure 7.5 Abstract interpretation diagram for strictness analysis.

First an abstraction function abs and a domain $\text{D}^\#$ have to be found in which the strictness property can be expressed. Then, the interpretation function $\text{f}^\#$ has to be defined working on the abstract domain. For each possible function f such an abstract interpretation function $\text{f}^\#$ has to be found.

The strictness question contains entities like 'all possible values' and 'any non-terminating expression'. So it is natural to represent these notions as elements of $\text{D}^\#$.

Define:

1 the value representing all possible (terminating as well as non-terminating) expressions (generally also indicated as 'top');

0 the value representing all non-terminating ($\perp$) expressions (generally also indicated as 'bottom');

and define the abstraction function abs accordingly:

$$
\begin{aligned}
&\text{abs} :: D \to D^{\#} \\
&\text{abs } x = 0, \text{ if } x \text{ does not terminate} \\
&\qquad\quad\; = 1, \text{ otherwise}
\end{aligned}
$$

where

$$
\begin{aligned}
D^{\#} &= \{0, 1\} \\
D &= \{\text{expressions}\}
\end{aligned}
$$

$D^{\#}$ is a domain with $0 \leq 1$ (since the set of all non-terminating expressions is a subset of the set of all possible expressions); 0 is the least element; 1 is the least upper bound for the increasing sequence $0 \leq 1$.

The abstraction function abs is clearly undecidable. Fortunately, the abstraction does not actually have to be made. Suppose that for an abstract function $f^{\#}$ the equation

$$f^{\#}\ 1 \ldots 1\ 0\ 1 \ldots 1 \quad = \; 0$$

can be solved. Then, the remaining problem is to find the proper abstract version $f^{\#}$ for each function f, and to prove that this abstraction is safe ($\text{abs} \,.\, f \leq f^{\#} \,.\, \text{abs}$). It can be proven that it is possible to construct abstract functions in such a way that the abstraction is safe. Proofs like this go beyond the scope of this book. For a more formal treatment of strictness analysis and abstract interpretation the reader is referred to Mycroft (1981), Burn *et al.* (1985) or Renardel (1988).

Owing to the safety criterion the actual determination of the undecidable abs functions can be avoided, since, when it is known that

$$f^{\#}\ 1 \ldots 1 \quad 0 \quad 1 \ldots 1 \qquad = \; 0$$

it follows that

$$f\ x_1 \ldots x_{i-1}\ \perp\ x_{i+1} \ldots x_n \quad = \; \perp$$

which can be shown as follows (using the safety criterion):

$$
\begin{aligned}
&(\text{abs} \,.\, f)\ x_1 \ldots x_{i-1} \perp x_{i+1} \ldots x_n \;\leq\; f^{\#}\ (\text{abs}\ x_1 \ldots x_{i-1} \perp x_{i+1} \ldots x_n) \\
\Rightarrow\quad &(\text{abs} \,.\, f)\ x_1 \ldots x_{i-1} \perp x_{i+1} \ldots x_n \;\leq\; f^{\#}\ 1 \ldots 1\ 0\ 1 \ldots 1 \\
\Rightarrow\quad &(\text{abs} \,.\, f)\ x_1 \ldots x_{i-1} \perp x_{i+1} \ldots x_n \;\leq\; 0 \\
\Rightarrow\quad &(\text{abs} \,.\, f)\ x_1 \ldots x_{i-1} \perp x_{i+1} \ldots x_n \;=\; 0 \\
\Rightarrow\quad &\text{abs}\ (f\ x_1 \ldots x_{i-1} \perp x_{i+1} \ldots x_n) \;=\; 0 \\
\Rightarrow\quad &f\ x_1 \ldots x_{i-1} \perp x_{i+1} \ldots x_n \;=\; \perp
\end{aligned}
$$

For the construction of an abstract function $f^{\#}$ a transformation function # can be defined that translates an arbitrary function f over D^n to its abstract counterpart $f^{\#}$. If f is defined by

$$f :: D^n \to D$$
$$f\ x_1 \ldots x_i \ldots x_n \quad = \quad E$$

then $f^{\#}$ is defined using a transformation function #:

$$f^{\#} :: (D^{\#})^n \to D^{\#}$$
$$f^{\#}\ x_1 \ldots x_i \ldots x_n \quad = \quad \# [\![E]\!]$$

Next we show how this transformation function # has to be defined. The resulting functions will satisfy the safety criterion.

7.4.1 Constants, variables and function applications

First, assume that a right-hand side of a function body only consists of constants, variables and applications of other functions. Constant expressions always terminate, so $\# [\![\text{constant}]\!] = 1$. The translation function # then becomes:

$\# [\![c]\!]$	$= 1$	where c is a constant
$\# [\![x]\!]$	$= x$	where x is a variable
$\# [\![f\ E_1 \ldots E_n]\!]$	$= f^{\#}\ \# [\![E_1]\!] \ldots \# [\![E_n]\!]$	where f is a function

7.4.2 δ-rules

Now consider abstract versions of simple functions like δ-rules. Take, for example, the addition. To perform an addition both arguments are needed. So clearly, $\perp + E = \perp$ for all expressions E. In the abstract world this means that always $+^{\#}\ 0\ x = 0$. In this way $+^{\#}$ can be defined for all possible arguments. It follows that $+^{\#}$ equals the logical AND operator & (where 0 and 1 represent Boolean values):

```
& 0 x    =  0
& x 0    =  0
& 1 1    =  1
```

The same holds for multiplication and many other δ-rules. So the abstraction function # can now be refined:

$$\# [\![+\ E1\ E2]\!] \ = \ \&\ \# [\![E1]\!]\ \# [\![E2]\!]$$
$$\# [\![*\ E1\ E2]\!] \ = \ \&\ \# [\![E1]\!]\ \# [\![E2]\!]$$
$$\ldots$$

A more difficult standard function is the conditional. The condition in a conditional expression is needed, so: $\text{if} \perp t\ e = \perp$. Hence, in the abstract world: $\text{if}^{\#}\ 0\ t\ e = 0$ for all t and e in $D^{\#}$. Furthermore, either the 'then' or the 'else' part is needed. Non-termination is possible if only one of these branches is non-terminating. Non-termination is certain when both 'then' and 'else' parts are non-terminating. The abstract conditional is defined as

$$\text{if}^{\#}\ \text{c t e} \ = \ \&\ \text{c}\ (|\ \text{t e})$$

where | is the logical OR operator. So:

$$\#[\![\ \text{if } E_c\ E_t\ E_e\ ]\!] \ = \ \&\ \#[\![\ E_c\ ]\!]\ (|\ \#[\![\ E_t\ ]\!]\ \#[\![\ E_e\ ]\!]\ )$$

7.5 Analysing function definitions

The definitions above can be used for deriving abstract versions of simple functions. Clearly, it can be the case that in the resulting abstract functions information is lost.

Consider the following example:

```
f x y z    =  if (x = 0) (y + z) (x – y)
```

then

$$
\begin{aligned}
& f^{\#}\ x\ y\ z \\
= \ & \#[\![\ \text{if } (x = 0)\ (y + z)\ (x - y)\ ]\!] \\
= \ & \&\ \#[\![\ (x = 0)\ ]\!]\ (|\ \#[\![\ (y + z)\ ]\!]\ \#[\![\ (x - y)\ ]\!]\ ) \\
= \ & \&\ (\&\ \#[\![\ x\ ]\!]\ \#[\![\ 0\ ]\!]\ )\ (|\ (\&\ \#[\![\ y\ ]\!]\ \#[\![\ z\ ]\!]\ )\ (\&\ \#[\![\ x\ ]\!]\ \#[\![\ y\ ]\!]\ )) \\
= \ & \&\ (\&\ x\ 1)\ (|\ (\&\ y\ z)\ (\&\ x\ y))
\end{aligned}
$$

Now this abstract function can be used for deriving strictness information by evaluating $f^{\#}$ 0 1 1, $f^{\#}$ 1 0 1 etc.:

$$
\begin{array}{llllll}
f^{\#}\ 0\ 1\ 1 & \rightarrow & \&\ (\&\ 0\ 1)\ (|\ (\&\ 1\ 1)\ (\&\ 0\ 1)) & \rightarrow & \&\ 0\ (|\ 1\ 0) & \rightarrow \quad 0 \\
f^{\#}\ 1\ 0\ 1 & \rightarrow & \&\ (\&\ 1\ 1)\ (|\ (\&\ 0\ 1)\ (\&\ 1\ 0)) & \rightarrow & 0 & \\
f^{\#}\ 1\ 1\ 0 & \rightarrow & \&\ (\&\ 1\ 1)\ (|\ (\&\ 1\ 0)\ (\&\ 1\ 1)) & \rightarrow & 1 &
\end{array}
$$

From the first two derivations it can be concluded that f is strict in x and y. This is what would be expected: x will always be evaluated in the condition, y will always be evaluated because it appears in both the 'then' and 'else' branches of the conditional. Of course, it cannot be concluded from the derivations above that f is *not* strict in z. Consider a slightly different definition of f:

```
f x y z    =  if (x = x) (y + z) (x – y)
```

Abstract interpretation will give that $f^{\#}$ 1 1 0 = 1. However, f is strict in z, because the 'then' part will always be chosen (so always: f x y $\perp = \perp$).

7.5.1 Recursive functions

When recursive functions are considered the analysis is more complex.

Consider the following recursive definition:

$$
\begin{array}{llll}
 & f\ x\ y & = & \text{if } (x = 0)\ y\ (f\ (x - 1)\ y) \\
\text{then,} & f^{\#}\ x\ y & = & \&\ (\&\ x\ 1)\ (|\ y\ (f^{\#}\ (\&\ x\ 1)\ y)) \\
\text{hence,} & f^{\#}\ 0\ 1 & \rightarrow & \&\ (\&\ 0\ 1)\ (|\ y\ (f^{\#}\ (\&\ 0\ 1)\ y)) \quad \rightarrow \quad 0
\end{array}
$$

So the method described above finds f to be strict in its first argument.

In general, however, (mutually) recursive functions can lead to a non-terminating reduction in the abstract world. It is in principle undecidable whether or not this situation occurs.

For instance, when the strictness of the second argument of the example above is calculated, the abstract reduction sequence is non-terminating:

$$f^{\#}\ 1\ 0 \rightarrow \&\ (\&\ 1\ 1)\ (|\ 0\ (f^{\#}\ (\&\ 1\ 1)\ 0)) \rightarrow \&\ 1\ (f^{\#}\ (\&\ 1\ 1)\ 0)) \rightarrow f^{\#}\ 1\ 0 \rightarrow \ldots$$

Since 1 is the upper bound of $D^{\#}$, it is always true that $f^{\#}\ x_1 \ldots x_n \leq 1$. This can be used to stop a (possibly infinite) computation at any moment to approximate the result with 1. In this way the interpretation stays safe, but too much information is lost.

With some help from domain theory the problem can be solved. Because $D^{\#}$ is a domain, the continuous functions over it also form a domain. Since $D^{\#}$ is finite, the domain $D^{\#} \times D^{\#} \rightarrow D^{\#}$ is also finite. Abstract functions that are constructed with continuous functions like &, | and if are continuous as well.

With abstract functions that are defined by a recursive equation, initially the only thing that is known of the function is the recursive equation and the fact that it is continuous. Since for finite domains this means that the function is also monotonic, the function can be approximated from below by a fixed point computation as follows: if $f^{\#}$ is a function of arity n, defined by the equation $f^{\#}\ x_1 \ldots x_n = E$, then functions $f^{\#m}$, $m \geq 0$, are defined as follows:

$$
\begin{array}{lll}
f^{\#0}\ x_1 \ldots x_n & = & 0 \\
f^{\#m}\ x_1 \ldots x_n & = & E\ [\![\ f^{\#} := f^{\#m-1}\ ]\!]
\end{array}
$$

where $[\![\ x := y\ ]\!]$ means that all occurrences of x have to be replaced by y. This sequence of functions is increasing. Therefore it has a least upper bound $f^{\#u}$, for which $f^{\#u} = f^{\#u+1}$ and therefore $f^{\#u} = f^{\#}$.

In the function domain the function defined by

$$F\ f\ x_1 \ldots x_n \quad = \quad E\ [\![\ f^{\#} := f\ ]\!]$$

takes a function and returns the next approximation which is a continuous function as well. Hence, this function has a fixed point for which $F\ d = d$, i.e. E ⟦ $f^{\#}$:= f ⟧ = f. This corresponds directly to the solution of the equation being found. Therefore, the computation via an approximating increasing sequence is a proper fixed point computation. In the case of mutually recursive functions a set of equations has to be solved simultaneously with a fixed point calculation. As a first approximation all functions are returning the bottom element for all their arguments. Each iteration calculates the next approximation for each function.

The fixed point calculation applied to the example above:

$$f^{\#}\ x\ y \quad = \quad \&\ (\&\ x\ 1)\ (|\ y\ (f^{\#}\ (\&\ x\ 1)\ y))$$

then (by definition):

$$f^{\#0}\ x\ y \quad = \quad 0$$

and (by repeated substitution):

$$f^{\#1}\ x\ y \quad = \quad \&\ (\&\ x\ 1)\ (|\ y\ (f^{\#0}\ (\&\ x\ 1)\ y)) \quad = \quad \&\ (\&\ x\ 1)\ (|\ y\ 0) \quad = \quad \&\ x\ y$$

$$\begin{aligned} f^{\#2}\ x\ y \quad &= \quad \&\ (\&\ x\ 1)\ (|\ y\ (f^{\#1}\ (\&\ x\ 1)\ y)) \quad && = \quad \&\ (\&\ x\ 1)\ (|\ y\ (f^{\#1}\ x\ y)) \\ &= \quad \&\ x\ (|\ y\ (\&\ x\ y)) && = \quad \&\ x\ y \end{aligned}$$

The sequence for this function converges quickly. The first and second element are already the same: $f^{\#1} = f^{\#2}$. So the solution is $f^{\#} = f^{\#1}$ and therefore:

$$f^{\#}\ x\ y \quad = \quad \&\ x\ y$$

such that $f^{\#}\ 1\ 0 \rightarrow 0$. Hence, f is also strict in its second argument.

A disadvantage of the method is that it requires that functions are tested for equality. Two functions are **equal** if they yield the same value for each possible combination of arguments. Because the complexity of comparing functions is exponential in the number of arguments, the derivation of the fixed point can be computationally expensive. In practice, the fixed point can often be found quickly. In the example above, one can see directly without computing any result that $f^{\#1}$ and $f^{\#2}$ are equal. However, a brute force comparison of $f^{\#1}$ and $f^{\#2}$ requires 2^2 comparisons.

7.5.2 Higher order functions

Until now only functions working from D^n to D were considered. Strictness analysis is more complex when higher order functions are allowed.

Consider the following higher order function:

```
hof g a b   =  (g a b) + (g b b)
```

The first element of hof is a function: an element of D x D → D assumed that hof delivers an element of D. For finding the abstract function $hof^{\#}$ the same technique as before can be used:

$hof^{\#}$ g a b = & (g a b) (g b b)

The test for strictness becomes difficult because $hof^{\#}$ 0 1 1 cannot be computed. The first argument of $hof^{\#}$ is of the wrong type. It should be a member of the domain $D^{\#} \times D^{\#} \rightarrow D^{\#}$, so the appropriate top and bottom elements from this domain should be chosen.

In order to deal with higher order functions new functions for the top element and bottom element have to be defined in the appropriate domain. So the bottom element is represented by a function that always yields 0, and the top element by a function that always yields 1.

For example, for the domain $D^{\#} \times D^{\#} \rightarrow D^{\#}$ define $f_{\perp}$ and f_{T}

$f_{\perp}$:: $D^{\#} \times D^{\#} \rightarrow D^{\#}$
$f_{\perp}$ x y = 0

f_{T}:: $D^{\#} \times D^{\#} \rightarrow D^{\#}$
f_{T} x y = 1

With these elements the strictness properties of hof can be determined: hof is only strict in its first argument.

$hof^{\#}$ $f_{\perp}$ 1 1 = 0
$hof^{\#}$ f_{T} 0 1 = 1
$hof^{\#}$ f_{T} 1 0 = 1

Functions over higher order functions can be recursive as well. The fixed point technique for recursive functions must now be applied in more complex domains. Finding such a fixed point can be expensive.

```
rechof g x y    =  if  (x = 0) (g x y) (rechof g (x – 1) (y + 1))
```

The function $rechof^{\#u}$ has to be found in the domain $(D^{\#} \times D^{\#} \rightarrow D^{\#}) \times D^{\#} \times D^{\#}$. For testing whether rechof is strict in argument x no fixed point computation has to be done. But for determining the strictness in y a least upper bound has to be determined.

Fortunately, many programs do not contain very complex higher order functions. So in practice, fixed points can often be found fairly easily.

7.6 Analysing non-flat domains

So far, only functions over flat domains such as integers and booleans are considered. No lists or other data types (constructors) were involved. Again, the analysis method described above can easily be extended by mapping all constructors to top:

$$\# [\![\text{Con } E_1 \ldots E_n]\!] = 1$$

where Con is a constructor of arity n. The disadvantage of such a rough approximation is that too much information is lost.

> Consider for example the function that sums all elements of a list:
>
> ```
> sum [ ]       =  0
> sum (a : x)   =  a + sum x
> ```
>
> It can easily be seen that sum is strict in its argument: the list given to sum can be evaluated to head normal form before applying sum to it. Furthermore, all the elements of the list can be evaluated in advance also.

The sum example above shows that for data structures like lists another kind of strictness analysis is required. The kind of strictness cannot be found by the abstract interpretation used thus far. But what abstract interpretation is strong enough for finding this kind of information? The most natural way seems to maintain the constructors in the abstract domain:

$$\# [\![\text{Con } E_1 \ldots E_n]\!] = \text{Con}^{\#} \, \# [\![E_1]\!] \ldots \# [\![E_n]\!]$$

So the abstract domain (for lists) contains elements like $\text{Nil}^{\#}$, $\text{Cons}^{\#}\ 0\ \text{Nil}^{\#}$, $\text{Cons}^{\#}\ 0\ (\text{Cons}^{\#}\ 0\ \text{Nil}^{\#})$ etc. This (non-flat) domain has a disadvantage: the domain is in general *infinite*. As a consequence, fixed point computations are no longer possible. Also, the kind of evaluation needed for the sum example is very difficult to express in this domain. What is needed is a finite domain in which several kinds of termination property can be expressed.

7.6.1 The four-point domain on lists

The method of Wadler (1987) is based on the observation that only some special kinds of reduction on lists are useful:

- reduction to head normal form:

```
is_empty (h : t)   =  False
is_empty [ ]       =  True
```

is_empty does not terminate when its argument has no head normal form.

- **spine reduction:** reduction of the 'spine' $[e_1, e_2, \ldots, e_m]$ of a list, but no reduction of the list elements $e_1, e_2, \ldots, e_m$ themselves:

```
length [ ]       =  0
length (h : t)   =  1 + length t
```

length only terminates for finite lists. The elements of the list are not needed.

- **element reduction:** reduction of the spine of the list and reduction to head normal form of all the list elements:

```
sum [ ]       =  0
sum (h : t)   =  h + sum t
```

sum only terminates for finite lists with no non-terminating elements.

All these kinds of reduction have to be represented in the abstract domain. Therefore abstract values av_i have to be introduced such that:

$$\text{is\_empty}^{\#}\ av_1 = 0$$
$$\text{length}^{\#}\ av_2 = 0$$
$$\text{sum}^{\#}\ av_3 = 0$$

Let LD be the domain of lists over the simple values of D. The following abstract values in the abstract domain $LD^{\#}$ are defined:

Bot	the bottom element, representing *all certainly non-terminating expressions*;
Inf	represents the same as Bot extended with *all infinite lists*;
BotMem	represents the same as Inf extended with *all lists with at least one non-terminating element* (member);
TopMem	represents *all possible lists*.

These values are sometimes also represented as the numbers 0, 1, 2 and 3. We shall use the above notation to prevent confusion with the elements of the domain $D^{\#}$ (0 and 1). In this analysis an infinite list can be a real infinite list as well as a list of which the tail is a non-terminating reduction. As a consequence, the domain $LD^{\#}$ has the following ordering:

Bot $\leq$ Inf $\leq$ BotMem $\leq$ TopMem. The meaning of these values is explained by considering the function f::LD $\rightarrow$ D. Suppose that in the abstract domain the result of applying $f^{\#}$ to the four abstract values can be derived:

- *reduction to head normal form*: if $f^{\#}$ Bot = 0, it is known that calling f with a non-terminating reduction will result in non-termination (hence f $\perp$ = $\perp$). The argument of f can be reduced safely to head normal form before applying f. If $f^{\#}$ Bot = 1, this is not possible.

```
is_empty#  Bot   =  0
length#    Bot   =  0
sum#       Bot   =  0
```

- *spine reduction*: if $f^{\#}$ Inf = 0, it is known that calling f with an infinite list will result in non-termination (hence f inflist = $\perp$, where inflist is some infinite list). Note that then also $f^{\#}$ Bot = 0 since $f^{\#}$ is continuous. In this case the spine of the argument of f can safely be evaluated before applying the function. If $f^{\#}$ Inf = 1, this is not possible.

```
is_empty#  Inf   =  1
length#    Inf   =  0
sum#       Inf   =  0
```

- *element reduction*: if $f^{\#}$ BotMem = 0, it is known that calling f with a list containing a non-terminating element will result in non-termination (hence f botelemlist = $\perp$, where botelemlist is such a list). Note that then also $f^{\#}$ Inf = 0 and $f^{\#}$ Bot = 0. In this case both the spine and the elements of the argument of f can safely be evaluated before applying the function. If $f^{\#}$ BotMem = 1, this is not possible.

```
is_empty#  BotMem   =  1
length#    BotMem   =  1
sum#       BotMem   =  0
```

- if $f^{\#}$ TopMem = 0, an application of f will never terminate (then also $f^{\#}$ BotMem = 0, $f^{\#}$ Inf = 0 and $f^{\#}$ Bot = 0).

```
is_empty#  TopMem   =  1
length#    TopMem   =  1
sum#       TopMem   =  1
```

7.6.2 Constructing abstract functions in the four-point domain

For the construction of the abstract functions so far only a translation of the right-hand side was necessary. Now also pattern matching on the left-hand side is taken into account.

The translation is explained by considering a more or less general function definition (in which E and N are functions with respectively zero and two arguments representing the right-hand sides of the alternatives):

```
f [ ]      =  E
f (h : t)  =  N h t
```

The abstract function $f^{\#}$ is now examined for the four possible arguments in the domain $LD^{\#}$.

The result of $f^{\#}$ Bot corresponds to the result of a call to f with a non-terminating element: f ⊥. Because the pattern matching never will terminate, the result of this will be ⊥, corresponding to bottom in the result domain. So f ⊥ = ⊥. Hence, we deduce $f^{\#}$ Bot = bottom.

The result of $f^{\#}$ Inf corresponds to the result of a call to f with an infinite list. Hence, the second member of f will always match. What can be said of the head and tail of an infinite list? The head (variable h) can have any value (Top), and in the abstract domain all occurrences of variable h can be replaced by 1. The tail of an infinite list can either be a non-terminating reduction (⊥), or it can also be an infinite list. Therefore, in the abstract domain variable t can be replaced by Bot or Inf. The result in the abstract domain will be the least upper bound of both possibilities (the smallest safe value in the abstract domain). Since all abstract functions are monotonic it is sufficient to replace the variable t by Inf.

A similar construction can be made for determining $f^{\#}$ BotMem. This corresponds to calling f with a list containing at least one element that does not terminate. This implies that in this case the second member of f will always match. Again two possibilities arise: either the head of the list does not terminate, in which case nothing more can be said about the tail, or the head of the list might terminate, in which case the tail of the list has to contain the non-terminating element. In the abstract world the first case corresponds to replacing variable h by 0 and variable t by TopMem. The second case corresponds to replacing variable h by 1 and variable t by BotMem.

Lastly, the same procedure has to be followed for TopMem. Function f is called with some list. One of the two members will match. If the second one does, nothing special can be said of the tail of the list. So the least upper bound has to be taken of the two right-hand sides. Hence

$f^{\#}$ Bot = bottom
$f^{\#}$ Inf = # ⟦ N ⟧ 1 Inf

$$f^{\#}\ \text{BotMem} = (\#[\![\ \text{N}\ ]\!]\ 0\ \text{TopMem}) \cup (\#[\![\ \text{N}\ ]\!]\ 1\ \text{BotMem})$$
$$f^{\#}\ \text{TopMem} = \#[\![\ \text{E}\ ]\!] \cup (\#[\![\ \text{N}\ ]\!]\ 1\ \text{TopMem})$$

Extending the method to functions with more arguments and more complex types is straightforward.

Another part of the translation scheme is formed by the right-hand sides. With the list constructors special things have to be done. Nil (or [ ]) is clearly a terminating list, and its abstract representation is TopMem. Cons (or :) becomes a predefined function: $\text{Cons}^{\#}$.

$$\#[\![\ [\,]\ ]\!] = \text{TopMem}$$
$$\#[\![\ E_1 : E_2\ ]\!] = \text{Cons}^{\#}\ \#[\![\ E_1\ ]\!]\ \#[\![\ E_2\ ]\!]$$

The Cons of a non-terminating expression and some list is a list with a non-terminating element, or abstractly: $\text{Cons}^{\#}$ 0 TopMem = BotMem. Similarly, $\text{Cons}^{\#}$ x y is defined for other arguments (see the table below).

$\text{Cons}^{\#}$ x y	x = 1	x = 0
y = TopMem	TopMem	BotMem
y = BotMem	BotMem	BotMem
y = Inf	Inf	Inf
y = Bot	Inf	Inf

Consider for example the function reverse: it reverses a list by using the standard function append:

```
reverse [ ]      =  [ ]
reverse (h : t)  =  append (reverse t) (h : [ ])
```

Convert the right-hand sides:

$$\#[\![\ [\,]\ ]\!] = \text{TopMem}$$
$$\#[\![\text{append (reverse t) (h : [ ])}]\!] = \ldots = \text{append}^{\#}\ (\text{reverse}^{\#}\ \text{t})\ (\text{Cons}^{\#}\ \text{h TopMem})$$

Convert pattern matching (with E = [ ] and N h t = append (reverse t) (h:[ ])) and apply the $\text{Cons}^{\#}$ table defined above:

$\text{reverse}^{\#}$ Bot = Bot || bottom
$\text{reverse}^{\#}$ Inf || $\#[\![\ \text{N}\ ]\!]$ 1 Inf
  = $\text{append}^{\#}$ ($\text{reverse}^{\#}$ Inf) ($\text{Cons}^{\#}$ 1 TopMem)
  = $\text{append}^{\#}$ ($\text{reverse}^{\#}$ Inf) TopMem
$\text{reverse}^{\#}$ BotMem || ($\#[\![\ \text{N}\ ]\!]$ 0 TopMem)$\cup$($\#[\![\ \text{N}\ ]\!]$ 1 BotMem)
  = ($\text{append}^{\#}$ ($\text{reverse}^{\#}$ BotMem) ($\text{Cons}^{\#}$ 1 TopMem))
  $\cup$ ($\text{append}^{\#}$ ($\text{reverse}^{\#}$ TopMem) ($\text{Cons}^{\#}$ 0 TopMem))

```
    =  (append# (reverse# BotMem) TopMem)
       ∪ (append# (reverse# TopMem) BotMem)
reverse# TopMem                    ||  # ⟦ E ⟧ ∪(# ⟦ N ⟧ 1 TopMem)
    =  TopMem ∪ (append# (reverse# TopMem) (Cons# 1 TopMem))
    =  TopMem
```

The resulting equations will still have to be solved to achieve proper abstract function definitions. Also the following defining equations of abstract functions can be derived in the same way:

```
is_empty#  Bot      =  0
is_empty#  Inf      =  1
is_empty#  BotMem   =  1
is_empty#  TopMem   =  1

length#    Bot      =  0
length#    Inf      =  length# Inf
length#    BotMem   =  length# TopMem = 1
length#    TopMem   =  1

sum#       Bot      =  0
sum#       Inf      =  sum# Inf
sum#       BotMem   =  sum# BotMem
sum#       TopMem   =  1
```

Some of the resulting equations are solved directly, e.g. for the function is_empty. For a function like sum, however, the abstract equation is recursive. As for functions over the domain D, a fixed point computation has to be performed to solve this recursive equation. For the functions sum and length this is easy since sum$^{\#}$ Inf, sum$^{\#}$ BotMem and length$^{\#}$ Inf are all 0.

The fixed point computation for the function sum:

```
sum#    Bot      =  0
sum#    Inf      =  sum# Inf
sum#    BotMem   =  sum# BotMem
sum#    TopMem   =  1
```

then (by definition):

```
sum#0   x        =  0
```

and (by substitution):

```
sum#1   Bot      =  0
```

$\text{sum}^{\#1}$	Inf	=	$\text{sum}^{\#0}$ Inf	=	0
$\text{sum}^{\#1}$	BotMem	=	$\text{sum}^{\#0}$ BotMem	=	0
$\text{sum}^{\#1}$	TopMem	=	1		
$\text{sum}^{\#2}$	Bot	=	0		
$\text{sum}^{\#2}$	Inf	=	$\text{sum}^{\#1}$ Inf	=	0
$\text{sum}^{\#2}$	BotMem	=	$\text{sum}^{\#1}$ BotMem	=	0
$\text{sum}^{\#2}$	TopMem	=	1		

The first and second approximations are equal, so the fixed point is found.

For reverse the computation is more complex: first the values for the function append should be known, then a fixed point of reverse can be determined in the domain $\text{LD}^{\#} \rightarrow \text{LD}^{\#}$ (see Exercise 7.3).

For mutually recursive function definitions fixed points cannot be calculated separately. In such a case the fixed points have to be calculated together.

7.6.3 Lists with more complex elements

To derive more information for lists with more complex elements, another abstract domain has to be constructed. Suppose the list consists of elements of a domain S for which the abstract domain $\text{S}^{\#}$ contains n elements s_i ($i \le n$). The abstract domain $\text{LS}^{\#}$ is constructed as follows. $\text{LS}^{\#}$ contains the elements Inf and Bot. Furthermore, for each element s_i of $\text{S}^{\#}$ an equivalent element s_iMem is added to $\text{LS}^{\#}$.

Consider for example the construction of $\text{LLD}^{\#}$:

$\text{D}^{\#}$	$\text{LD}^{\#}$	$\text{LLD}^{\#}$	…
1 (Top)	TopMem	TopMemMem	…
0 (Bot)	BotMem	BotMemMem	
	Inf	InfMem	
	Bot	BotMem	
		Inf	
		Bot	

Then, for example, BotMemMem corresponds to lists that have at least one element that is a list containing at least one element that is bottom. Of course the same can be done for lists of functions etc.

It will be clear that this kind of strictness analysis is difficult when more complicated data structures (e.g. trees) are taken into account. In that case the abstract domains that have to be constructed are complicated as well. As a result, the complexity of the analysis can explode.

7.7 Abstract reduction

Another strictness information deduction method is based on **abstract reduction**. With both abstract interpretation and abstract reduction a compile-time abstract computation is performed in an abstract domain. An important difference is that abstract reduction takes arbitrary data structures (data constructors) into account as well as pattern matching on these data constructors. Abstract reduction can be regarded as a special kind of symbolic graph rewriting and is therefore based on GRSs, whereas abstract interpretation is based on the λ-calculus.

A disadvantage of the possibility of handling arbitrary data structures is that the abstract domain in which the abstract reduction takes place can become *infinite*. Hence the computation of a fixed point can take an infinite amount of time. So, in principle, strictness analysis based on abstract reduction may not terminate. In practice, an approximating analyser can be defined that will terminate and that will find strictness in many cases in a relatively short time.

In this section proofs of properties are omitted. For further details the reader is referred to (Nöcker, 1988; 1993).

7.7.1 The abstract domain

Strictness analysis via abstract reduction proceeds as follows: via an undecidable abstraction function abs the concrete domain S is abstracted to the abstract *power domain* PS#. PS# is defined by first taking the **powerset** of S, i.e. the set PS of all subsets of S. Then, PS# is constructed by joining every element of PS with the set Bot of all non-terminating and undefined expressions of S. In this way Bot is the least element of the domain PS#. The greatest element of PS# is S, from now on referred to as Top. PS# is in general an infinite domain since S can be infinite (S contains all possible terms).

No distinction is made between a partial function that is undefined for a particular value of its arguments and a function that is non-terminating for a particular value of its arguments. Both kinds of function are united in the least element Bot.

An important advantage of the abstract domain PS# is that it contains all subsets of all kinds of expressions. This enables reasoning about all kinds of structures without the necessity to define special abstract domains for it. Another consequence is that a least upper bound *always* exists for any two elements in the abstract domain and that this lub can be found by taking the set union of these elements.

Rewriting in PS# implies that rewriting is now defined on *sets*. The constructors that are used in the rewrite rules denote elements of PS#.

Top	≡	S
Bot	≡	the set of all non-terminating expressions
Constructor# Top	≡	the set of all terms starting with Constructor ∪ Bot

$\mathsf{Nil}^\#$	$\equiv$	$\{\mathsf{Nil}\} \cup \mathsf{Bot}$
$\mathsf{Cons}^\#\ \mathsf{Bot}\ \mathsf{Nil}^\#$	$\equiv$	the set of all lists of length 1 with a non-terminating element $\cup$ Bot

Below, the method of abstract reduction will be explained. Compared with abstract interpretation the construction of abstract functions will turn out to be a trivial task. The important aspect of the analysis now lies in the definition of abstract rewriting.

7.7.2 Transforming functions to abstract functions

Functions defined on S are transformed into abstract functions defined on $PS^\#$ in the following way. Assume that a function f of arity n is defined with k rule alternatives:

$$
\begin{array}{lll}
\mathsf{f} :: \mathsf{S}^n \to \mathsf{S} & & \\
\mathsf{f}\ p_{11} \ldots p_{1n} & = & \mathsf{rhs}_1 \\
\ldots & & \\
\mathsf{f}\ p_{k1} \ldots p_{kn} & = & \mathsf{rhs}_k
\end{array}
$$

For the construction of an abstract function $\mathsf{f}^\#$ the following transformation function # can be defined that translates an arbitrary function f over D^n to its abstract counterpart $\mathsf{f}^\#::(PS^\#)^n \to PS^\#$.

$$
\begin{array}{lllll}
\#[\![\ \mathsf{f}\ p_{11} \ldots p_{1n} = \mathsf{rhs}_1 & & & & \\
\quad \ldots & & & & \\
\quad \mathsf{f}\ p_{k1} \ldots p_{kn} = \mathsf{rhs}_k\ ]\!] & = & \#[\![\ \mathsf{f}\ p_{11} \ldots p_{1n}\ ]\!] & = & \#[\![\ \mathsf{rhs}_1\ ]\!] \\
 & & \ldots & & \\
 & & \#[\![\ \mathsf{f}\ p_{k1} \ldots p_{kn}\ ]\!] & = & \#[\![\ \mathsf{rhs}_k\ ]\!] \\
\#[\![\ s\ x_1 \ldots x_n\ ]\!] & = & s^\#\ \#[\![\ x_1\ ]\!] \ldots \#[\![\ x_n\ ]\!] & & \\
\#[\![\ x\ ]\!] & = & x & &
\end{array}
$$

Here s stands for a function or constructor symbol; x stands for a variable.

So the transformation is trivial: every symbol s is converted to $s^\#$.

For the following function definitions the corresponding abstracted functions are constructed by applying the transformation #.

```
f x                  =  g (Constructor (h x))
g (Constructor x)    =  x
h 0                  =  0
h x                  =  x
```

$\mathsf{f}^\#$ x	=	$\mathsf{g}^\#$ ($\mathsf{Constructor}^\#$ ($\mathsf{h}^\#$ x))
$\mathsf{g}^\#$ ($\mathsf{Constructor}^\#$ x)	=	x

```
h# 0#                    =  0#
h# x                     =  x
```

7.7.3 Rewriting

In principle, the functional strategy is used for reduction as usual. However, rewriting in the abstract domain is defined on *sets*. In order to define rewriting on sets some small changes are necessary. A variable on a left-hand side now stands for the set of all expressions (Top). Furthermore, four situations have to be distinguished during the pattern matching phase (for each situation an example is given):

(1) There is a **total match**: the *formal* argument is a superset of the *actual* argument, i.e. the actual argument is contained in the formal argument. When there is a total match, the matching can continue.

The term h# 0# totally matches h# 0#

(2) There is a **partial match**: the *actual* argument is a superset of the *formal* argument. When there is a partial match another rule alternative may be applicable as well. So although a match has been found and matching can continue with this rule alternative, the next rule alternatives also have to be considered with as actual argument the remaining non-matching part of the original argument. The remaining part is defined as follows: if A is the actual argument, B the formal argument and $A \supset B$, then the remaining part C is all elements of A that are not contained in B, united with Bot. This is denoted as $C = A \setminus B$.

The term g# Top partially matches g# (Constructor# x).
The remaining part is Top \ (Constructor# Top)

(3) There is an **infinite match**: the formal argument is a non-variable; the actual argument is Bot. This case corresponds to forcing evaluation of a non-terminating expression to match it against a non-variable pattern. This will lead to a non-terminating reduction in the concrete world. In the abstract world the rule alternative is applicable: the term reduces to Bot, independent of its right-hand side.

The term g# Bot infinitely matches g# (Constructor# x)

(4) There is **no match**: none of the previous cases arises: the rule alternative is not applicable. Matching proceeds with the next rule alternative.

The term $g^{\#}$ $Nil^{\#}$ does not match $g^{\#}$ ($Constructor^{\#}$ x)

After the matching phase rewriting is done accordingly with the special extension that when more than one rule alternative is applicable (due to a partial match) the *union* of all possible results is taken.

If none of the rule alternatives matches, the result of the rewriting is Bot, indicating that the partial function is undefined.

One-step rewriting of the terms in the examples of Section 7.7.2:

$$
\begin{array}{lll}
h^{\#}\ 0^{\#} & \rightarrow & 0^{\#} \\
g^{\#}\ \mathrm{Top} & = & g^{\#}\ ((\mathrm{Constructor}^{\#}\ \mathrm{Top}) \cup (\mathrm{Top} \setminus (\mathrm{Constructor}^{\#}\ \mathrm{Top}))) \rightarrow \mathrm{Top} \cup \mathrm{Bot} \\
 & = & \mathrm{Top} \\
g^{\#}\ \mathrm{Bot} & \rightarrow & \mathrm{Bot} \\
g^{\#}\ \mathrm{Nil}^{\#} & \rightarrow & \mathrm{Bot}
\end{array}
$$

In an implementation of abstract reduction, for reasons of efficiency and termination, further approximations can be made (see also Section 7.8.1). For example, one can approximate all numbers by Top. In the case of a partial match one can continue with the complete argument instead of with the remaining non-matching part. Another optimization concerns the use of graph structures in the analysis. Combined with sensible criteria for termination (see Section 7.8.1), an efficient implementation of abstract reduction can be obtained.

7.8 Strictness analysis using abstract reduction

The concrete reduction in the standard domain is approximated by reduction on sets in the abstract domain. Approximation takes place in two ways. Firstly, the objects of reduction are sets of which the elements cannot be distinguished any more. Secondly, the reduct can be a set that is larger than necessary, e.g. the reduct always contains Bot.

Just as with abstract interpretation the safety criterion has to hold. For abstract reduction the *safety criterion* ($\mathrm{abs} \,.\, f \leq f^{\#} \,.\, \mathrm{abs}$) means that for every function, when it is applied with concrete arguments, the abstraction of all its results has to be a subset of the set of the results of the corresponding abstract function applied on all corresponding abstract arguments.

In other words, the **safety criterion for abstract reduction** requires that for all $x_1 \ldots x_n \in S$ it holds that when $t, t' \in PS^{\#}$ are defined as $t = \cup\, x^{\#} \mid f\, x_1 \ldots x_n \rightarrow_* x$ and $t' = \cup\, x^{\#} \mid f^{\#}\, x_1{}^{\#} \ldots x_n{}^{\#} \rightarrow_* x^{\#}$ then, $t \leq t'$.

Matching and rewriting on sets are defined in such a way that this safety criterion holds. As a consequence, the strictness question for the ith argument of a function f of arity n:

$$f\ x_1 \ldots x_{i-1}\ \perp\ x_{i+1} \ldots x_n \quad = \quad \perp$$

can be approximated via abstract reduction as follows:

$f^{\#}$ Top … Top Bot Top … Top $\rightarrow_*$ Bot

This question can be answered via abstract reduction to normal form.

Consider again an example of Section 7.5: f x y z = if (x = 0) (y + z) (x – y). With pattern matching this function can now be defined as:

```
f 0 y z   =  y + z
f x y z   =  x – y
```

The abstract reduction for the various strictness questions yields the same answers as in Section 7.5:

$f^{\#}$ Bot Top Top $\rightarrow$ Bot
$f^{\#}$ Top Bot Top $\rightarrow$ (Bot $+^{\#}$ Top)$\cup$(Top $-^{\#}$ Bot) $\rightarrow \dots \rightarrow$ Bot$\cup$Bot = Bot
$f^{\#}$ Top Top Bot $\rightarrow$ (Top $+^{\#}$ Bot)$\cup$(Top $-^{\#}$ Top) $\rightarrow \dots \rightarrow$ Bot$\cup$Top = Top

However, with respect to the strictness of $f^{\#}$ of the example in Section 7.7.2 abstract reduction yields better results than abstract interpretation:

$f^{\#}$ Bot $\rightarrow$ $g^{\#}$ (Constructor$^{\#}$ ($h^{\#}$ Bot)) $\rightarrow$ $h^{\#}$ Bot $\rightarrow$ Bot

So $f^{\#}$ is strict in its first argument. This fact would not have been found using abstract interpretation as explained in this chapter.

Strictness analysis using abstract reduction on sets can be performed relatively easily. The introduction of new constructors does not increase the complexity of the analysis. Owing to the possibility of performing *pattern matching on abstract constructors*, results can be obtained that are much harder to obtain with strictness analysis based on abstract interpretation. However, abstract reduction has the disadvantage that it is not guaranteed to terminate.

Non-terminating abstract reduction:

$f^{\#}$ x = $f^{\#}$ x

$f^{\#}$ Bot $\rightarrow$ $f^{\#}$ Bot $\rightarrow$ …

Since the domain $PS^{\#}$ is infinite a fixed point analysis cannot be performed. So termination of the analysis must be realized in another way.

7.8.1 Termination of abstract reduction

Although in principle the analysis is non-terminating, it can always be stopped. When it is stopped the result is approximated. Fortunately, it is always safe to approximate a reduction with Top since this set contains all possible terms. Some limit must be placed upon the reduction sequence. For instance, we can use the depth of the recursion in the reduction, or the number of reductions that have been performed in the analysis, or the length of the term that is analysed. An implementation criterion can also be the depth of the recursion stack, or the time needed, or the memory required, or a combination of these.

In certain special cases an analysis of the reduction path (**path analysis**) can introduce an extra stop criterion when the analysis leads to the conclusion that a non-terminating reduction is going on. An example of such an path analysis is the following:

Suppose an analysis of the reduction path yields the following about a term t: t $\rightarrow_*$ E(t) in which E(t) is an expression, not in head normal form, in which t occurs. When the reduction of t is needed to reduce E(t), the term t in E(t) can be replaced by Bot. When it is not known whether the reduction of t is needed to reduce E(t), the term t in E(t) can be replaced by Top.

Path analysis avoids non-termination in the previous example:

$f^{\#}$ Bot → $f^{\#}$ Bot → Bot

Clearly, reduction of the term $f^{\#}$ Bot is needed in the expression $f^{\#}$ Bot.

Consider one of the examples of Section 7.5.1: f x y = if (x = 0) y (f (x – 1) y). With pattern matching this function can now be defined as:

```
f 0 y  =  y
f x y  =  f (x – 1) y
```

For this case path analysis yields the same answer as the fixed point analysis in Section 7.5.1:

$f^{\#}$ Top Bot		=	($f^{\#}$ $0^{\#}$ Bot)∪($f^{\#}$ (Top\\$0^{\#}$) Bot)
→	Bot ∪ $f^{\#}$ ((Top\\$0^{\#}$)–$^{\#}$ $1^{\#}$) Bot	=	$f^{\#}$ ((Top\\$0^{\#}$)–$^{\#}$ $1^{\#}$) Bot
→	$f^{\#}$ Top Bot		
→	Bot		

7.8.2 Abstract reduction on non-flat domains

Other questions than the standard strictness question fit nicely in the general framework of abstract reduction. Consider, for example, strictness analysis on non-flat domains, e.g. list structures. With abstract interpretation the most simple analysis is done via the introduction of the four-point domain for which appropriate abstract functions have to be constructed. With abstract reduction a similar analysis can be performed directly. The elements of the four-point domain correspond directly with sets that are elements of the domain PS#:

Bot	≡ the set of all non-terminating expressions: least element of PS#
Inf	≡ the set of all infinite lists ∪ Bot
BotMem	≡ the set of all lists with at least 1 non-terminating element ∪ Inf
TopMem	≡ the set of all possible lists

These sets are elements of the domain PS# and as sets they obey the following relation: Bot ⊆ Inf ⊆ BotMem ⊆ TopMem which corresponds to the ordering in the four-point domain.

In order to perform abstract reduction, first abstract functions have to be constructed, and then strictness questions can be asked. The results of the abstract functions for each of the indicated sets can be computed in the standard way using abstract reduction. Note that the abstract functions are not just defined on the four special sets, but are defined on the complete domain PS#.

Take again an example of Section 7.6: sum. The abstract function is now:

```
sum# Nil#            =  0#
sum# (Cons# a x)     =  a +# sum# x
```

In order to be able to perform the pattern matching on the indicated sets, more about the structure of these sets has to be known. The following recursive set equations are sufficient to perform pattern matching.

```
Inf       =  Cons# Top Inf
BotMem    =  (Cons# Top BotMem) ∪ (Cons# Bot TopMem)
TopMem    =  Nil# ∪ (Cons# Top TopMem)
```

With these equations and path analysis abstract reduction is performed.

The sum example with abstract reduction:

```
sum# Bot       →   Bot
sum# Inf       =   sum# (Cons# Top Inf)
               →   Top +# (sum# Inf)
               →   Top +# Bot
               →   Bot
sum# BotMem    =   sum# ((Cons# Top BotMem) ∪ (Cons# Bot TopMem))
               →   (Top +# (sum# BotMem)) ∪ (Bot +# (sum# TopMem))
               →   (Top +# (sum# BotMem)) ∪ Bot
```

$$
\begin{aligned}
& \rightarrow && \text{Top } +^{\#} (\text{sum}^{\#} \text{ BotMem}) \\
& \rightarrow && \text{Top } +^{\#} \text{ Bot} \\
& \rightarrow && \text{Bot} \\
\text{sum}^{\#} \text{ TopMem} \quad & = && \text{sum}^{\#} (\text{Nil}^{\#} \cup (\text{Cons}^{\#} \text{ Top TopMem})) \\
& \rightarrow && 0^{\#} \cup (\text{sum}^{\#} (\text{Cons}^{\#} \text{ Top TopMem})) \\
& \rightarrow && 0^{\#} \cup (\text{Top } +^{\#} (\text{sum}^{\#} \text{ TopMem})) \\
& \rightarrow && 0^{\#} \cup (\text{Top } +^{\#} \text{ Top}) \\
& \rightarrow && 0^{\#} \cup \text{Top} \\
& = && \text{Top}
\end{aligned}
$$

Another important aspect of abstract reduction is the fact that *a priori* no restriction is imposed on the considered structures. The strictness of functions that manipulate user-defined data structures can be investigated in the same way as the strictness of lists. Furthermore, the complexity of the analysis does not explode if more complex structures are considered. There is no necessity to define more complicated domains (with more complex fixed point analysis) for such structures. An increasing complexity of the structures that are considered only leads to increasing complexity of the arguments of the abstract functions. The complexity of the path analysis and of the abstract reduction itself does not increase.

Summary

- A function is *strict* in a certain argument if evaluation of that argument does not change the termination behaviour of the program.
- Strictness analysis is a *compile-time* analysis to obtain *strictness information*. It can be used to change the reduction order in such a way that the time and space behaviour of the program is improved.
- Strictness is inherently *undecidable*. Therefore, practical strictness analysis always deals with *approximations* that must be *safe*, i.e. the termination behaviour of a program has to remain unchanged.
- Two general techniques of strictness analysis are discussed: *abstract interpretation* on *finite* abstract domains and *abstract reduction* on the powerset domain that can be *infinite*.
- Abstract interpretation approximates strictness via an analysis in a *finite domain* that is constructed with the purpose of answering a special kind of strictness question.
  – The use of continuous functions on the finite domain makes it possible to deal with recursion via a *fixed point analysis*.
  – With the *four-point domain* list structures are handled.
  – For more complex information more complex domains have to be defined; the complexity of the analysis *explodes*.

- Abstract reduction approximates strictness via a special way of reducing *sets* of expressions.
  - Pattern matching and reduction with sets is defined in a *safe* way such that the reduct of a set contains at least all reducts of all expressions in the original set.
  - In principle, abstract reduction is *non-terminating*. However, *path analysis* and practical *heuristics* make it possible to avoid this non-termination in practice at the price of making further approximations.
  - *Recursive set equations* are used to deal with pattern matching and reduction on all kinds of sets.
  - Within the general framework of abstract reduction, strictness of all kinds of structures is *relatively easily* examined since it is not necessary to construct new domains and abstract functions for every new kind of question. No explosion of complexity takes place.

EXERCISES

7.1 Give for each element a of the four-point domain (TopMem, BotMem, Inf, Bot) an element x of the concrete domain such that $x \in abs^{-1}\ a$ and for all $a' < a$ it holds that $x \notin abs^{-1}\ a'$.

7.2 The definition of the append function is:

```
append:: LD->LD->LD
append  [ ]        list    =  list
append  (h : t)    list    =  h : append t list
```

Give the defining equations of the abstract function append# in the four-point domain. Solve the equations using fixed point analysis.

7.3 Solve with the resulting definition of append# the defining equations of reverse# in Section 7.6.2. Derive the required strictness information from the resulting definition.

7.4 Derive both using abstract interpretation as well as using abstract reduction the strictness information for f, as defined below.

```
head:: LD->D
head (a : as)  =  a

f:: D->D
f   x  =  head (x : [ ])
```