The Tensor Product in Wadler's Analysis of Lists

Flemming Nielson & Hanne Riis Nielson,
Dept. of Comp. Science, Aarhus University,
DK-8000 Aarhus C, Denmark.

E-mail: fnielson@daimi.aau.dk

We consider abstract interpretation (in particular strictness analysis) for pairs and lists. We begin by reviewing the well-known fact that the best known description of a pair of elements is obtained using the tensor product rather than the cartesian product. We next present a generalisation of Wadler's strictness analysis for lists using the notion of open set. Finally, we illustrate the intimate connection between the case analysis implicit in Wadler's strictness analysis and the precision that the tensor product allows for modelling the inverse **cons** operation.

1 Introduction

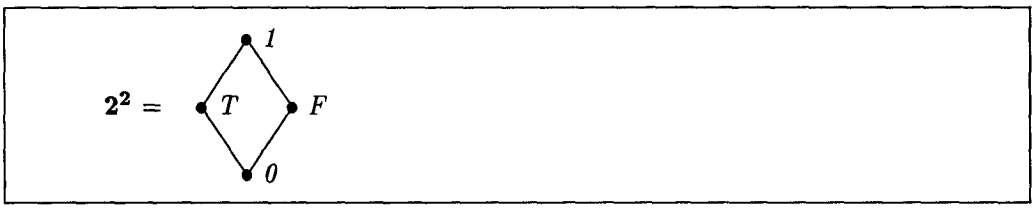
Let us begin with pairs. It is common belief that to describe a pair one must use a pair of descriptions. As an example consider a pair (true,false) and an analysis for detecting constants (see the figure). It is immediate that T is the best description of 'true' and F is the best description of 'false' so that it is natural to use (T,F) as the description of (true,false).

It is well-known (but perhaps to too few!) that in general this approach does not give the best description possible. As an example consider the pair $(\mathbf{x},\mathbf{x})$ where $\mathbf{x}$ is either 'true' or 'false' and thus is described by 1. Here the above strategy would call for using $(1,1)$. A similar description would arise for the pair $(\mathbf{x},\neg\mathbf{x})$ and if the use of the pair was to test for equality of the two booleans we will obviously not obtain precise information: it would appear that the result of the test is $(1=1)$ which clearly is 1.

The solution is immediate: we will describe $(\mathbf{x},\mathbf{x})$ by (T,T) *or* (F,F) and $(\mathbf{x},\neg\mathbf{x})$ by (T,F) *or* (F,T) — assuming of course that $\mathbf{x}$ is described by 1. Then the test will always yield T in case of $(\mathbf{x},\mathbf{x})$ and always F in case of $(\mathbf{x},\neg\mathbf{x})$.

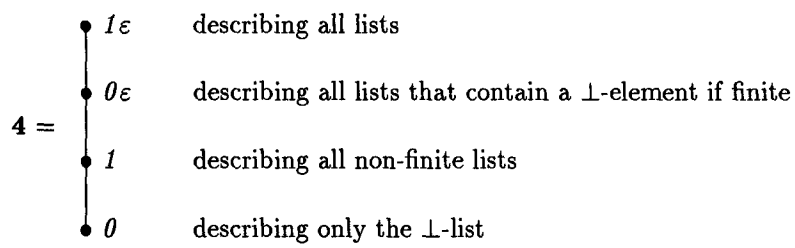
This observation is by no means novel. It dates back (at least) to [9] that distinguished between independent attribute analyses (the first kind) and relational analyses (the second kind). The first systematic treatment was given in [11] and the highlights are also presented in [12, 13, 14]. It amounted to the following identifications:

$$\begin{aligned} \text{independent attribute method} &\equiv \text{cartesian product} \\ \text{relational method} &\equiv \text{tensor product} \end{aligned}$$



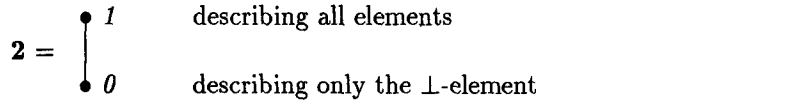
The notion of tensor product is a very general notion from category theory [10]. One has to be specific about the category (complete lattices) and the property (additivity or distributivity) in order to home in on the concept. An early reference to tensor products of complete lattices is [2] and [11] gave a direct construction that was closer to motivating **why** the tensor product would be useful for the relational method; the construction we give in Section 2 is a cut-down version that applies to finite complete lattices only. (Hence the reader can happily forget about compact elements, consistently complete cpo's, algebraicity, ideal completions etc. for the duration of this paper.)

Let us now turn to lists. Here the difficulty is not to find a general description of lists but to find one that is useful for the analysis of lazy languages. The first remarkable success in this area was Wadler's strictness analysis for lists [17]. For lists of base types, like `Int list`, it used a four-point domain:



Here a list is finite if it is of the form v_1:...:v_n:NIL, and is non-finite if it is either infinite, i.e. v_1:v_2:..., or else partial, i.e. v_1:...:v_n:⊥.

Much work has been directed at generalizing Wadler's construction to other recursive data structures (e.g. [5]). In a sense this is not hard; however, it would seem that no one has been able to obtain a generalisation that is equally natural. (Almost all generalisations contain far too many descriptive elements and more or less ad-hoc ways have to be found to throw some of them out again.) Here we consider the more mundane task of generalising Wadler's analysis from lists of base types to arbitrary lists. One easy approach (discovered by many) is to note that Wadler's construction amounts to the double lifting of the two-point domain



used to describe the strictness properties of base types. However, this does not give the desired descriptive power when the elements of the lists have more structure. This was also observed in [5] and in Section 3 we shall see how to do better — without first introducing many more descriptive elements and next making sure that only the interesting ones are retained.

The success of Wadler's analysis is not only due to the use of a four-point domain but rests at least as much on the (implicit) use of case analysis when analysing function definitions. In Section 4 we then show that case analysis amounts to nothing but the use of an inverse cons operation — provided that the range of the inverse cons operation is modelled using tensor product. This amounts to a formalisation of Wadler's remark that the case analysis is performed by using the abstraction of cons "in a backward manner".

2 Tensor products for pairs

Let us consider a small *lazy* functional language with types given by[1]

$$t ::= \mathtt{Int} \mid \mathtt{Bool} \mid t \times t \mid t \to t \mid t \otimes t \mid t\ \mathtt{list}$$

The first step in describing an analysis by means of abstract interpretation is to describe the complete lattice $\mathbf{A}(t)$ associated with each type t. For strictness analysis it is common to model the base types using the two-point lattice **2** described above. However, to illustrate how lists of structured types are handled we shall be a bit more ambitious in some of our choices:

$$\mathbf{A}(\mathtt{Int}) = \mathbf{2}$$
$$\mathbf{A}(\mathtt{Bool}) = \mathbf{2}^{\mathbf{2}}$$

Thus our modelling of Int is a proper strictness analysis whereas our modelling of Bool amounts to an analysis for detecting constants; however, *if only the 0 and 1 elements are retained we have a proper strictness analysis* corresponding to the use of $\mathbf{A}(\mathtt{Bool}) = \mathbf{2}$.

For composite types our starting point will be the following definitions:

$$\mathbf{A}(t_1 \times t_2) = (\mathbf{A}(t_1) \times \mathbf{A}(t_2))_\perp$$
$$\mathbf{A}(t_1 \to t_2) = (\mathbf{A}(t_1) \to \mathbf{A}(t_2))_\perp$$

The basic idea is that a property of a pair of elements is a pair of properties (one for each component) and that a property of a function is a function that maps properties of arguments to properties of results. There is the additional twist that we use an outer lifting. This is in order to distinguish between the undefined element of a product or function space and the least "defined" element (a pair of $\perp$-properties or the function mapping any property to the $\perp$-property). The choice corresponds to the choice made in the standard semantics (see the Appendix) and is invaluable in order to model the behaviour of a lazy functional language.

Example 1 Consider the function eq : Bool × Bool → Bool that tests for equality of its two arguments. In the analysis **A** it will be natural to set

$$\mathbf{A}(\mathtt{eq}) = up(eq \circ dn)$$

[1] In a realistic language one would have only one of × and ⊗, say ×. Some occurrences of × will then be interpreted as we interpret × and others as we interpret ⊗. The actual choice will depend on the precision wanted and the context of the occurrence.

where $eq : \mathbf{2^2} \times \mathbf{2^2} \to \mathbf{2^2}$ is given by

$$eq(T,T) = T,\ eq(F,F) = T,\ eq(T,F) = F,\ eq(F,T) = F,\ eq(1,1) = 1, \cdots$$

and where *up* and *dn* are the standard "polymorphic operators" that transform between domains D and $D_\perp$, i.e.

$$up : D \to D_\perp$$
$$dn : D_\perp \to D$$

with $dn(\perp) = \perp$ as well as $dn(up(\perp)) = \perp$; thus $dn \circ up$ is the identity, *id*, but $up \circ dn$ is greater than the identity because it sends $\perp$ to $up(\perp)$. □

Returning to the example of the Introduction let us consider the behaviour of a function

```
f(x) = eq(x,x)
```

upon an element **x** that can be either 'true' or 'false'. Here *1* describes **x** and using **A(eq)** as specified in Example 1 we obtain *1* as the result, even though **we** know that the result must be 'true' so that one would have hoped for T as the result of the analysis.

The crux of the problem is that

$$up((T,T)) \sqcup up((F,F)) = up((T,F)) \sqcup up((F,T))$$

and that we are therefore not able to describe the difference between the pairs (**x**,**x**) and (**x**,¬**x**) where **x** is described by *1*. The solution we propose is to use lifted *tensor product* rather than lifted cartesian product. This will enable us to achieve

$$up(cross(T,T)) \sqcup up(cross(F,F)) \neq up(cross(T,F)) \sqcup up(cross(F,T))$$

for a suitable function *cross*. However, to be able to compare the possibilities we shall keep the interpretation of $\mathbf{A}(t_1 \times t_2)$ and instead interpret $\mathbf{A}(t_1 \otimes t_2)$ as stated.

To conduct this development we need a few auxiliary notions. A function $f{:}L \to M$ is (binary) *additive* if $f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$ holds for all l_1 and l_2 in L. A function $f{:}L \times L' \to E$ is *separately* (binary) *additive* if

$$f(l_1 \sqcup l_2, l') = f(l_1,l') \sqcup f(l_2,l'), \text{ and } f(l,l'_1 \sqcup l'_2) = f(l,l'_1) \sqcup f(l,l'_2)$$

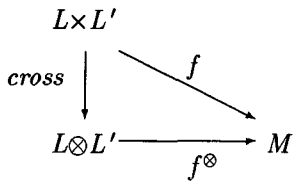
for all choices of l_1, l_2, l, l'_1, l'_2 and l'. It is easy to show that if $f{:}L \times L' \to M$ is additive then it is also separately additive but the converse does not hold. The tensor product may then be regarded as a way of turning separately additive functions into additive ones. To be more precise consider complete lattices L and L'.

Definition 2 A pair $(L \otimes L', cross)$ is a tensor product of L and L' (with respect to additivity) provided that

- $L \otimes L'$ is a complete lattice,
- *cross*: $L \times L' \to L \otimes L'$ is a continuous function that is separately additive,

- for all complete lattices M and for all continuous functions $f{:}L{\times}L'{\to}M$ that are separately additive the following universal property holds: there exists precisely one continuous function $f^{\otimes}$: $L{\otimes}L'{\to}M$ that is additive and satisfies the equation $f^{\otimes}{\circ}cross{=}f$. □

This may all be illustrated by the following diagram:



More precisely we have "defined" the tensor product (with respect to additivity) within the category **CL** of complete lattices (as objects) and continuous functions (as morphisms). We have as yet no guarantee that the tensor product always exists in **CL** or in subcategories. However, it follows from general categorical reasoning that the tensor product — if it exists — is unique up to isomorphism; this means that if $(M_1,cross_1)$ and $(M_2,cross_2)$ are both tensor products of L and L' there is an isomorphism θ from M_1 to M_2 such that $\theta{\circ}cross_1{=}cross_2$. (Here an isomorphism θ is a bijection such that it and its inverse are both morphisms of the category in question.)

Example 3 The lower powerdomain $\mathcal{P}(D)$ of an algebraic cpo D is

$$\mathcal{P}(D) = (\{Y{\subseteq}\mathrm{B}_D \mid Y{\neq}\emptyset \wedge Y{=}\mathrm{LC}(Y)\},{\subseteq})$$

where B_D is the set of compact elements of D and

$$\mathrm{LC}(Y) = \{d \mid \exists y{\in}Y{:}\ d{\sqsubseteq}y\}$$

is the left-closure of a subset Y. (If D is finite one has $\mathrm{B}_D{=}D$.) Then

$$(\mathcal{P}(D{\times}D'),\ \lambda(Y,Y').\{(y,y') \mid y{\in}Y \wedge y'{\in}Y'\})$$

is a tensor product of $\mathcal{P}(D)$ and $\mathcal{P}(D')$. □

The above example shows that the tensor product always exists in the category of distributive and finite lattices and monotonic (hence continuouos) functions; this follows from [6, Section 7] that in effect shows that L is a finite and distributive lattice if and only if $L{=}\mathcal{P}(D)$ for a finite cpo D. A much more general result may be found in [2] but note that the notion of tensor product studied there is slightly different[2]. Here we shall be content with demonstrating the existence of the tensor product within the category **FCL** of finite complete lattices and monotonic (hence continuous) functions.

The elements of $L{\otimes}L'$ will be be certain subsets Y of $L{\times}L'$. To this end we shall say that a set Y is *left-closed* when $Y = \mathrm{LC}(Y)$ and where $\mathrm{LC}(Y)$ is as above. We shall say that a set Y is *closed in both components* when $Y = \mathrm{CC}_1(Y)$ and $Y = \mathrm{CC}_2(Y)$ and where

[2]It is the tensor product with respect to complete additivity. Which tensor product is the more adequate depends on the setting at hand. We believe that the tensor product studied in this paper is well suited for lazy languages whereas that of [2] is well suited for eager languages.

$$\mathrm{CC}_1(Y) = \{(l_1 \sqcup l_2, l') \mid (l_1,l'),(l_2,l') \in Y\}$$
$$\mathrm{CC}_2(Y) = \{(l, l'_1 \sqcup l'_2) \mid (l,l'_1),(l,l'_2) \in Y\}$$

denote the closure in the first and second components, respectively.

Fact 4 For each subset $Y \subseteq L \times L'$ the set

$$\mathrm{TC}(Y) = \bigcap \{ Y' \subseteq L \times L' \mid Y \subseteq Y' \wedge Y' = \mathrm{LC}(Y') \wedge Y' = \mathrm{CC}_1(Y') \wedge Y' = \mathrm{CC}_2(Y') \}$$

is the least left-closed set that contains Y and that is closed in both components. □

Proposition 5 The following data

$$L \otimes L' = (\{ Y \subseteq L \times L' \mid Y \neq \emptyset \wedge Y = \mathrm{LC}(Y) = \mathrm{CC}_1(Y) = \mathrm{CC}_2(Y) \}, \subseteq)$$
$$cross = \lambda(l,l').\ \mathrm{LC}(\{(l,l')\})$$
$$f^{\otimes} = \lambda Y.\ \bigsqcup \{ f(l,l') \mid (l,l') \in Y \}$$

constructs a tensor product (with respect to additivity) in the category **FCL** of finite complete lattices and monotonic (hence continuous) functions. □

Proof: See [15, Chapter 7] or possibly [11] or [14]. □

We can now return to the definition of the analysis **A** where we have already hinted at the desire to use

$$\mathbf{A}(t_1 \otimes t_2) = (\mathbf{A}(t_1) \otimes \mathbf{A}(t_2))_{\perp}$$

Example 6 In Example 1 we considered the analysis of the function `eq`. Now consider the similar function $\mathtt{eq}' : \mathtt{Bool} \otimes \mathtt{Bool} \to \mathtt{Bool}$. For this it is natural to set

$$\mathbf{A}(\mathtt{eq}') = up(\lambda a.\ \bigsqcup \{ eq(l,l') \mid (l,l') \in dn(a) \})$$

In this way $\mathbf{A}(\mathtt{eq}')$ will give F when applied to $up(cross(T,F)) \sqcup up(cross(F,T))$ and will give T when applied to $up(cross(T,T)) \sqcup up(cross(F,F))$. Thus the required precision has been obtained. □

3 Wadler-like analysis of general lists

To prepare for our analysis of lists we need some terminology. Given a partially ordered set D and a subset $Y \subseteq D$ we define the right-closure of Y, or upwards closure of Y, as

$$\mathrm{RC}(Y) = \{ d \in D \mid \exists y \in Y: y \sqsubseteq d \}$$

(In the literature this is sometimes written $\uparrow Y$.) A subset $Y \subseteq D$ is Scott-open, or open in the Scott-topology, if and only if

- Y is right-closed, and
- for all chains $(d_n)_n$: if $\bigsqcup_n d_n \in Y$ then $d_n \in Y$ for some n.

Given our restriction to finite lattices the second condition is trivial and Scott-open just means right-closed throughout this paper. It is immediate that RC(Y) is the least right-closed set that contains Y. We now define

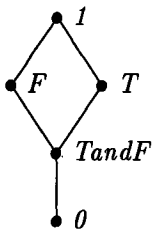
$$\mathcal{O}(D) = (\{Y \subseteq D \mid Y = \mathrm{RC}(Y) \wedge Y \neq \emptyset\}, \supseteq)$$

as the partially ordered set of non-empty right-closed sets.

Fact 7 If D is a finite complete lattice then $\mathcal{O}(D)$ is a finite complete lattice with least element D, greatest element $\{\top_D\}$ where $\top_D$ is the greatest element of D, and least upper bounds and greatest lower bounds given by $\bigcap$ and $\bigcup$, respectively. □

Example 8 $\mathcal{O}(\mathbf{A}(\mathtt{Int}))$ has elements $\{0,1\}$ and $\{1\}$ with $\{0,1\} \sqsubseteq \{1\}$; thus $\mathcal{O}(\mathbf{A}(\mathtt{Int}))$ is isomorphic to **2**.

$\mathcal{O}(\mathbf{A}(\mathtt{Bool}))$ has elements $\{1\}$, $\{F,1\}$, $\{T,1\}$, $\{T,F,1\}$, and $\{0,T,F,1\}$. The partial order may be depicted as follows



where 1 denotes $\{1\}$, T denotes $\{T,1\}$, $TandF$ denotes $\{T,F,1\}$ etc. (Clearly it is isomorphic to $(\mathbf{2}^{\mathbf{2}})_\perp$.)

Since $\mathbf{A}(\mathtt{Int} \times \mathtt{Int})$ is isomorphic to $\mathbf{A}(\mathtt{Bool})_\perp$ it follows that $\mathcal{O}(\mathbf{A}(\mathtt{Int} \times \mathtt{Int}))$ is as above but with an additional least element. □

For our analysis of lists we shall then use the following generalisation of Wadler's construction:

$$\mathbf{A}(t\ \mathtt{list}) = (\mathcal{O}(\mathbf{A}(t))_\perp)_\perp$$

To overcome the growing notational complexity it is helpful to write

0	for $\perp$,
1	for $up(\perp)$,
$Y\varepsilon$	for $up(up(Y))$,
$y\varepsilon$	for $\mathrm{RC}(\{y\})\varepsilon$, that is $up(up(\mathrm{RC}(\{y\})))$.

The intended meaning of these elements is as follows:

0	describes the $\perp$-list,
1	additionally describes all infinite lists and all partial lists,
$\top\varepsilon$	describes all lists.

Finally, let $Y\varepsilon \in \mathbf{A}(t\ \mathtt{list})$ satisfy $Y \neq \mathrm{RC}(\{\top\})$. We may write $Y = \{a_1, \cdots, a_k\}$ and we know that k>0. Then

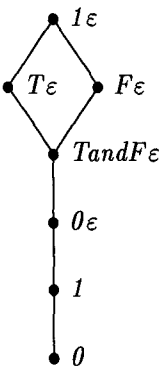
$Y\varepsilon$ describes all infinite lists and all partial lists and some finite lists; a finite list $[v_1,\cdots,v_n] = v_1{:}\cdots{:}v_n{:}\text{NIL}$ is described if there are k values $j_1,\cdots,j_k$ such that the property a_i describes the element v_{j_i}.

A more precise definition of the relationship between properties of lists and concrete lists may be found in the Appendix.

Example 9 For the strictness analysis we get $\mathbf{A}(\texttt{Int list}) = \mathbf{4}$ as in Wadler's approach because $\mathcal{O}(\mathbf{A}(\texttt{Int}))$ is isomorphic to $\mathbf{A}(\texttt{Int})$ (cf. Example 8). A similar remark applies to lists of lists of base types etc.

The difference between Wadler's approach and ours arises when the elements have more structure as for lists of booleans. Here we get:

$\mathbf{A}(\texttt{Bool list}) =$



The element $TandF\varepsilon$ really is $\text{RC}(\{T,F\})\varepsilon = \{1,T,F\}\varepsilon$ and it is the only element of $(\mathcal{O}(\mathbf{A}(\texttt{Bool}))_\perp)_\perp$ that is not accounted for in $(\mathbf{A}(\texttt{Bool})_\perp)_\perp$. Thus we are able to give a more precise description of a list like [true,false] than Wadler is, because we can record that both 'true' and 'false' are present in the list.

As follows from Example 8 the analysis of $\mathbf{A}((\texttt{Int}\times\texttt{Int})\ \texttt{list})$ would be very similar except that there would be an additional element between 1 and 0ε. The list [true,false] now corresponds to the list $[up((1,\perp)),up((\perp,2))]$ which does not have an adequate description in Wadler's approach (cf. the discussion of [5] where the inadequacy of a generalisation of Wadler's approach is described). The Appendix contains a slightly more detailed discussion of these matters. □

Example 10 Returning to Example 9 the concrete lists described by these properties may be described as follows:

0 describes $\perp$,

1 additionally describes all infinite lists and all partial lists (e.g. true:false:$\perp$),

0ε additionally describes all finite lists with at least one $\perp$-element (e.g. [$\perp$]),

$TandF\varepsilon$	additionally describes all finite lists with at least one 'true'-element and at least one 'false'-element (e.g. [true,false]),
$T\varepsilon$	additionally describes all finite lists with at least one 'true'-element (e.g. [true]),
$F\varepsilon$	additionally describes (wrt. $TandF\varepsilon$) all finite lists with at least one 'false'-element (e.g. [false]),
1ε	additionally describes [].

To illustrate the benefit of using $(\mathcal{O}(\mathbf{A}(\texttt{Bool}))_\perp)_\perp$ rather than $(\mathbf{A}(\texttt{Bool})_\perp)_\perp$ consider the (Miranda-like) function

```
g l = isnil (filter (=true) l) ∨ isnil(filter (=false) l)
```

Here `isnil` tests for a list being `nil` and `filter (=x) l` denotes the list consisting of those elements of `l` that equal `x`. Thus

$\mathbf{g}$([true,false]) = false

$\mathbf{g}$([true,$\perp$]) = $\perp$

$\mathbf{g}$(false:$\perp$) = $\perp$

$\mathbf{g}$([true]) = true

$\mathbf{g}$([false]) = true

If we let g denote the optimal analysis of $\mathbf{g}$ it follows that

$g(0\varepsilon) = 0$

$g(TandF\varepsilon) = F$

$g(T\varepsilon) = g(F\varepsilon) = 1$

Now consider the list [true,false]. It is described by all of $T\varepsilon$, $F\varepsilon$ and $TandF\varepsilon$ but not by 0ε; hence $TandF\varepsilon$ is the optimal description of [true,false]. Thus the inclusion of $TandF\varepsilon$ allows the analysis of $\mathbf{g}$([true,false]) to give a better result than if $TandF\varepsilon$ had not been included.

An analogous example can be constructed for the type (`Int`×`Int`) `list`. □

4 Tensor products and case analysis for lists

So far we have not shown how to interpret the functions associated with the tensor product, nor the notion of case analysis implicit in Wadler's analysis. This is all rectified in this section where the connection is also demonstrated. When doing so we shall introduce the required language primitives as the need arises; we do not have the space to give a full interpretation, nor do we have the space to explain the intricacies of the fixed point operator.

Example 11 Turning to operations on lists of integers we recall the strictness properties 0, 1, 0ε and 1ε and consider the functions `hd` that takes the head of a list, `length` that

computes the lenght of a list and sum that adds a list of integers. The *optimal analysis* of these functions are $up(hd)$, $up(length)$ and $up(sum)$ where hd, $length$ and sum are given by

	0	1	0ε	1ε
hd	0	1	1	1
$length$	0	0	1	1
sum	0	0	0	1

We shall regard hd as a primitive of the functional language whereas length and sum will be programs; as we shall see it will cost some effort to obtain a compositional and optimal analysis **A** of length and sum. □

To account for Wadler's notion of case-analysis we shall assume that there is a case construct. Informally, the meaning of case(e_1,e_2) should be "equivalent" to[3] cond(isnil, e_1, $e_2$otuple(hd,tl)) where cond is the familiar conditional, isnil is the test for whether a list is empty or not, hd and tl are the selection functions for lists and tuple(e_1,e_2) is intended to map v to $(e_1(v),e_2(v))$. By incorporating case as a language primitive we will be able to specify the strictness properties of case freely.

Example 12 Using the case construct we may define the following version of the length and sum programs:

$\texttt{length}_1$: Int list → Int
$\texttt{length}_1$ = fix(λf. case(zero, addotuple(one, fosnd)))

$\texttt{sum}_1$: Int list → Int
$\texttt{sum}_1$ = fix(λf. case(zero, addotuple(fst, fosnd)))

We shall return to their analysis, when we have defined the analysis **A**. □

Most of the definitions needed are rather straightforward:

$\mathbf{A}(\texttt{zero}) = up(\lambda a.\ 1)$

$\mathbf{A}(\texttt{one}) = up(\lambda a.\ 1)$

$\mathbf{A}(\texttt{add}) = up(\lambda a.\ a_1 \sqcap a_2 \text{ where } (a_1,a_2)=dn(a))$

$\mathbf{A}(\texttt{tuple}) = strict(\lambda h_1.\ strict(\lambda h_2.\ up(\lambda a.$
$\quad up((dn(h_1)(a),dn(h_2)(a))))))$

$\mathbf{A}(\texttt{fst}) = up(\lambda a.\ a_1 \text{ where } (a_1,a_2) = dn(a))$

$\mathbf{A}(\texttt{snd}) = up(\lambda a.\ a_2 \text{ where } (a_1,a_2) = dn(a))$

$\mathbf{A}(\texttt{cons}) = strict(\lambda h_1.\ strict(\lambda h_2.\ up(\lambda a.$
$\quad \begin{cases} 1 & \text{if } dn(h_2)(a) \sqsubseteq 1 \\ Y\varepsilon \sqcap (dn(h_1)(a))\varepsilon & \text{if } dn(h_2)(a) = Y\varepsilon \end{cases})))$

$\mathbf{A}(\texttt{nil}) = up(\lambda a.\ \top\varepsilon)$

[3] They will be equivalent in the semantics **S** of the Appendix but the whole point is that they will not be equivalent in the analysis **A**.

$$\mathbf{A}(\mathtt{hd}) = up(\lambda a.\ \begin{cases} \bot & \text{if } a{=}0 \\ \top & \text{if } a \sqsupseteq 1 \end{cases})$$

$$\mathbf{A}(\mathtt{tl}) = up(\lambda a.\ \begin{cases} 0 & \text{if } a{=}0 \\ 1 & \text{if } a{=}1 \\ \top\epsilon & \text{if } a{=}Y\epsilon \end{cases})$$

$$\mathbf{A}(\mathtt{isnil}) = up(\lambda a.\ \begin{cases} 0 & \text{if } a{=}0 \\ F & \text{if } a{\neq}0 \wedge a{\neq}\top\varepsilon \\ 1 & \text{if } a{=}\top\varepsilon \end{cases})$$

$$\mathbf{A}(\mathtt{fix}) = \mathrm{FIX}' \text{ where } \mathrm{FIX}' = \lambda H.\ \bigsqcup_{n\geq 1} H^n(up(\bot))$$

Here we have used the notation

$$strict(H) = \lambda h.\ \begin{cases} H\ h & \text{if } h{\neq}\bot \\ \bot & \text{otherwise} \end{cases}$$

Note that if $\mathbf{A}(\mathtt{Bool}) = \mathbf{2}$ was used instead of $\mathbf{A}(\mathtt{Bool}) = \mathbf{2^2}$ then we would have to let $\mathbf{A}(\mathtt{isnil})$ return *1* rather than *F* and thus loose precision. The more interesting construct is **case**:

$$\mathbf{A}(\mathtt{case}) = strict(\lambda h_1.\ strict(\lambda h_2.\ up(\lambda a.\ \begin{cases} \bot_{\mathbf{A}(t_1)} & \text{if } a{=}0 \\ dn(h_2)(up((\top_{\mathbf{A}(t_0)}, 1))) & \text{if } a{=}1 \\ \bigsqcup\{dn(h_2)(up((\sqcap Y', (Y \ominus Y')\varepsilon))) \mid Y' \subseteq Y\} & \text{if } a{=}Y\varepsilon \neq \top\varepsilon \\ dn(h_2)(up((\top_{\mathbf{A}(t_0)}, \top\varepsilon))) \sqcup dn(h_1)(\top\varepsilon) & \text{if } a{=}\top\varepsilon \end{cases}))))$$

where we have used the notation

$$Y \ominus Y' = \mathrm{RC}(\mathrm{Y}\backslash\mathrm{RC}(Y')) \cup \{\top\}$$

The two first and the last clause should be fairly straightforward. In the third clause Y' is the set of possible descriptions of the first element of the list and then $Y \ominus Y'$ will be the corresponding description of the tail of the list. All choices of $Y' \subseteq Y$ are possible so we join the results. The correctness of this definition is demonstrated in the Appendix.

Example 13 In the case of lists of base types the above definition of $\mathbf{A}(\mathtt{case})$ amounts to the following:

$$\mathbf{A}(\mathtt{case}) = strict(\lambda h_1.\ strict(\lambda h_2.\ up(\lambda a.\ \begin{cases} 0 & \text{if } a{=}0 \\ dn(h_2)(up((1,1))) & \text{if } a{=}1 \\ dn(h_2)(up((1,0\varepsilon))) \sqcup dn(h_2)(up((0,1\varepsilon))) & \text{if } a{=}0\varepsilon \\ dn(h_2)(up((1,1\varepsilon))) \sqcup dn(h_1)(1\varepsilon) & \text{if } a{=}1\varepsilon \end{cases}))))$$

We shall explain the definition in the case where $a{=}0\varepsilon$. Here we use that $a{=}0\varepsilon$ really stands for $a{=}Y\varepsilon$ with $Y{=}\{0,1\}$. The subsets Y' of Y are $\emptyset$, $\{0\}$, $\{1\}$ and $\{0,1\}$ but we only need to consider $\{1\}$ and $\{0,1\}$. Since

$$\sqcap\{0,1\} = 0 \text{ and } \{0,1\} \ominus \{0,1\} = \{1\}$$

$$\sqcap\{1\} = 1 \text{ and } \{0,1\} \ominus \{1\} = \{0,1\}$$

this gives the contribution

$$dn(h_2)(up((1,0\varepsilon))) \sqcup dn(h_2)(up((0,1\varepsilon)))$$

as stated. Thus our general definition of $\mathbf{A}(\mathtt{case})$ specializes to Wadler's notion of case analysis for lists of base types. □

Example 14 As we have already said `hd` is a primitive and thus there is no need to analyse it here. We may now perform the following analysis of `length`:

$$\begin{aligned}[\![\mathtt{length}_1]\!](\mathbf{A}) = up(&\lambda a.\text{case } a \text{ of}\\ &0\colon 0\\ &1\colon 0\\ &0\varepsilon\colon 1\\ &1\varepsilon\colon 1)\end{aligned}$$

Thus $dn([\![\mathtt{length}_1]\!](\mathbf{A}))$ equals the optimal result of Example 11. Turning to `sum` we may perform the following analysis:

$$\begin{aligned}[\![\mathtt{sum}_1]\!](\mathbf{A}) = up(&\lambda a.\text{case } a \text{ of}\\ &0\colon 0\\ &1\colon 0\\ &0\varepsilon\colon 0\\ &1\varepsilon\colon 1)\end{aligned}$$

Thus also $dn([\![\mathtt{sum}_1]\!](\mathbf{A}))$ equals the optimal result of Example 11. □

We shall now see how to obtain a similar effect by using the tensor product and then dispensing with the `case` construct. We begin by considering the operators `tuple'`, `fst'` and `snd'` associated with (tensor) products:

$$\begin{aligned}&\mathbf{A}(\mathtt{tuple}') \in \mathbf{A}(t_0{\to}t_1) \to \mathbf{A}(t_0{\to}t_2) \to \mathbf{A}(t_0{\to}t_1{\otimes}t_2)\\ &\mathbf{A}(\mathtt{tuple}') = strict(\lambda h_1.\ strict(\lambda h_2.\ up(\lambda a.\\ &\qquad up(cross(dn(h_1)(a),\ dn(h_2)(a))))))\\ &\mathbf{A}(\mathtt{fst}') \in \mathbf{A}(t_1{\otimes}t_2{\to}t_1)\\ &\mathbf{A}(\mathtt{fst}') = up(\lambda a.\ \bigsqcup\{l|(l,l')\in dn(a)\})\\ &\mathbf{A}(\mathtt{snd}') \in \mathbf{A}(t_1{\otimes}t_2{\to}t_2)\\ &\mathbf{A}(\mathtt{snd}') = up(\lambda a.\ \bigsqcup\{l'|(l,l')\in dn(a)\})\end{aligned}$$

Furthermore,

$$\begin{aligned}&\mathbf{A}(\mathtt{add}') \in \mathbf{A}(\mathtt{Int}{\otimes}\mathtt{Int}{\to}\mathtt{Int})\\ &\mathbf{A}(\mathtt{add}') = up(\lambda a.\ \bigsqcup\{\ a_1{\sqcap}a_2 \mid (a_1,a_2){\in}dn(a)\ \})\end{aligned}$$

However, the weak point is that `tuple'` is the only operator that constructs an element of the tensor product and that this element is of the form $cross(\cdots,\cdots)$ and so does not exploit the additional precision of the tensor product. This can be rectified by letting the interpretation of `tuple'` consider the atoms or the irreducible elements of the argument a (or $dn(a)$); references to approaches following these ideas may be found in the

Conclusion. Here we shall take a shortcut and introduce special operators for exploiting the tensor product. One is `split` which is the inverse `cons` operation and it is supposed to be "equivalent" to `tuple'(hd,tl)`. The other is `pair`$(e_1,\ e_2)$ that is supposed to be "equivalent" to `tuple'`$(e_1$`ofst'`$,\ e_2$`osnd'`$)$. For the analysis we then have

$$\mathbf{A}(\texttt{split}) \in \mathbf{A}(t\ \texttt{list} \to t \otimes (t\ \texttt{list}))$$

$$\mathbf{A}(\texttt{split}) = up(\lambda a.\ up(\begin{cases} cross(\bot,0) & \text{if } a=0 \\ cross(\top,1) & \text{if } a=1 \\ \bigsqcup\{cross(\sqcap Y',\ (Y \ominus Y')\varepsilon) \mid Y' \subseteq Y\} & \text{if } a=Y\varepsilon \neq \top\varepsilon \\ cross(\top,\top\varepsilon) & \text{if } a=\top\varepsilon \end{cases}))$$

$$\mathbf{A}(\texttt{pair}) \in \mathbf{A}(t_1 \to t_3) \to \mathbf{A}(t_2 \to t_4) \to \mathbf{A}(t_1 \otimes t_2 \to t_3 \otimes t_4)$$

$$\mathbf{A}(\texttt{pair}) = strict(\lambda h_1.\ strict(\lambda h_2.\ up(\lambda a.\ up(\bigsqcup\{cross(dn(h_1)(l),\ dn(h_2)(l')) \mid (l,l') \in dn(a)\}))))$$

Note the similarities between the definition of $\mathbf{A}$(`split`) and that of $\mathbf{A}$(`case`).

Example 15 In the case of lists of base types the above definition of $\mathbf{A}'$(`split`) amounts to the following:

$$\mathbf{A}'(\texttt{split}) = up(\lambda a.up(\begin{cases} cross(0,0) & \text{if } a=0 \\ cross(1,1) & \text{if } a=1 \\ cross(0,1\varepsilon) \sqcup cross(1,0\varepsilon) & \text{if } a=0\varepsilon \\ cross(1,1\varepsilon) & \text{if } a=1\varepsilon \end{cases}))$$

Naturally this has many similarities to the simplification of $\mathbf{A}$(`case`) obtained in Example 13. □

We also need

$$\mathbf{A}(\texttt{cond}) \in \mathbf{A}(t_0 \to \texttt{Bool}) \to \mathbf{A}(t_0 \to t_1) \to \mathbf{A}(t_0 \to t_1) \to \mathbf{A}(t_0 \to t_1)$$

$$\mathbf{A}(\texttt{cond}) = strict(\lambda h_1.\ strict(\lambda h_2.\ strict(\lambda h_3.\ up(\lambda a.\ \begin{cases} \bot & \text{if } dn(h_1)(a)=0 \\ dn(h_2)(a) & \text{if } dn(h_1)(a)=T \\ dn(h_3)(a) & \text{if } dn(h_1)(a)=F \\ dn(h_2)(a) \sqcup dn(h_3)(a) & \text{if } dn(h_1)(a)=1 \end{cases}))))$$

Example 16 Using `split` and `sum` we may now consider the following definitions of `length` and `sum`:

$$\texttt{length}_2 : \texttt{Int list} \to \texttt{Int}$$

$$\texttt{length}_2 = \texttt{fix}(\lambda \texttt{f}.\ \texttt{cond(isnil, zero, add'opair(one,f)osplit)})$$

$$\texttt{sum}_2 : \texttt{Int list} \to \texttt{Int}$$

$$\texttt{sum}_2 = \texttt{fix}(\lambda \texttt{f}.\ \texttt{cond(isnil, zero, add'opair(id,f)osplit)})$$

Again there is no need to redefine `hd` and thus no need to analyse it. We may then perform the following analysis of `length`:

$$\begin{aligned}[\![\texttt{length}_2]\!](\mathbf{A}) = up(&\lambda a.\text{case } a \text{ of}\\ &0\colon 0\\ &1\colon 0\\ &0\varepsilon\colon 1\\ &1\varepsilon\colon 1)\end{aligned}$$

Thus $dn([\![\texttt{length}_2]\!](\mathbf{A}))$ equals the optimal result of Example 11. This is not due to the use of tensor product but more to the use of $\mathbf{A}(\texttt{Bool}) = \mathbf{2}^{\mathbf{2}}$ instead of $\mathbf{A}(\texttt{Bool}) = \mathbf{2}$ (cf. the definition of $\mathbf{A}(\texttt{isnil})$ given earlier).

Turning to sum we may perform the following analysis:

$$\begin{aligned}[\![\texttt{sum}_2]\!](\mathbf{A}) = up(&\lambda a.\text{case } a \text{ of}\\ &0\colon 0\\ &1\colon 0\\ &0\varepsilon\colon 0\\ &1\varepsilon\colon 1)\end{aligned}$$

Thus also $dn([\![\texttt{sum}_2]\!](\mathbf{A}))$ equals the optimal result of Example 11. This is in contrast to what would happen if cartesian product was used instead of tensor product. □

5 Conclusion

Judging from the development of the previous section we can obtain optimal results for key functions using either case-analysis or the tensor product. Admittedly our treatment had a few special operators but these may be dispensed with at the price of a more complex theory: [11] contains formulations for a general tuple′-construct where the *join-irreducible elements* are used for case analysis. A similar development but using *atoms* is contained in [12]. This all relates to the study of so-called *expected forms* [11, 14].

One should take care, however, to note that there is a certain "duality" in the sets considered. For lists we are using right-closed sets whereas for tensor products we are using left-closed sets (that are additionally closed in each component). The use of left-closed sets is rather natural for abstract interpretation as is evidenced by the central role the lower powerdomain plays in many formulations of abstract interpretation. The use of right-closed sets for lists seems to be necessary to capture the essence of Wadler's insight: the ability to describe long finite lists that may have arbitrary elements except that one of these has to be $\perp$. In the terminology of [1] one might say that the Wadler-like analysis of lists necessitates a formulation of liveness aspects in addition to the safety aspects.

We should like to investigate the relationship between our use of open sets and the use of least Moore families in [4] for extending abstraction lattices with additional elements. It is important to note that (unlike [5]) our construction specialises to that of [17] in the case where the abstraction lattice for the element type is a chain. One may view our use of $\mathbf{A}(t\ \texttt{list}) = ((\mathcal{O}(\mathbf{A}(t)))_\perp)_\perp$ as opposed to $\mathbf{A}(t\ \texttt{list}) = ((\mathbf{A}(t))_\perp)_\perp$ as a way of introducing the required 'meets'.

A final note concerns the exclusion of the empty set in the definition of $\mathcal{O}(\cdots)$. We believe that a more "uniform" development would result if the empty set was admitted; in particular the correctness predicate *val* of the Appendix could then be defined in a more "natural" way on the top-element. However, for lists of base types we would then get a five-point domain rather than Wadler's four-point domain.

Acknowledgement

The Semantique meeting at Barra gave the impetus for writing this paper. The present research is part of The DART-Project which is funded by The Danish Research Councils.

References

[1] S.Abramsky: Abstract Interpretation, Logical Relations and Kan Extensions, *Journal of Logic and Computation* **1** *1* (1990), 5–40.

[2] H.-J.Bandelt: The tensorproduct of continuous lattices, *Mathematische Zeitschrift* **172** (1980) 89–96.

[3] G.L.Burn, C.Hankin, S.Abramsky: Strictness analysis for higher-order functions, *Science of Computer Programming* **7** (1986) 249–278.

[4] P.Cousot, R.Cousot: Systematic Design of Program Analysis Frameworks, *Procedings POPL 1979.*

[5] A.B.Ferguson, R.J.M.Hughes: An Iterative Powerdomain Construction, *Functional Programming, Glasgow 1989*, K.Davis and J.Hughes (eds.), Springer-Verlag (1989) 41–55.

[6] G.Grätzer: *Lattice Theory: First concepts and distributive lattices*, W.H.Freeman and Company (1971).

[7] J.Hughes: Strictness detection in non-flat domains, *Proc. Programs as Data Objects*, Springer Lecture Notes in Computer Science **217** (1986) 112–135.

[8] J.Hughes: Backwards Analysis of Functional Programs, *Partial Evaluation and Mixed Computation*, D.Bjørner, A.P.Ershov and N.D.Jones (eds.), North-Holland (1988) 187–208.

[9] N.D.Jones, S.S.Muchnick: Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra, *Program Flow Analysis: Theory and Applications*, S.S.Muchnick and N.D.Jones (eds.), Prentice-Hall (1981).

[10] S. Mac Lane: *Categories for the Working Mathematician*, Springer-Verlag (1971).

[11] F.Nielson: Abstract Interpretation using Domain Theory, *Ph.D.-thesis CST-31-84, University of Edinburgh, Scotland* (1984).

[12] F.Nielson: Tensor Products Generalize the Relational Data Flow Analysis Method, *Proceedings of the 4'th Hungarian Computer Science Conference* (1985) 211–225.

[13] F.Nielson: Towards a Denotational Theory of Abstract Intepretation, *Abstract Interpretation of Declarative Languages*, S.Abramsky and C.Hankin (eds.), Ellis Horwood (1987) 219–245.

[14] F.Nielson: Two-Level Semantics and Abstract Interpretation, *Theoretical Computer Science — Fundamental Studies* **69** *2* (1989) 117–242.

[15] F.Nielson, H.R.Nielson: *Two-Level Functional Languages*, Cambridge University Press (to appear 1992).

[16] G.D.Plotkin: Lambda definability in the full type hierarchy, *To H.B.Curry: Essays on Combinatorial Logic, Lambda Calculus and Formalism*, Academic Press (1980).

[17] P.Wadler: Strictness analysis on non-flat domains (by abstract interpretation over finite domains), *Abstract Interpretation of Declarative Languages*, S.Abramsky and C.Hankin (eds.), Ellis Horwood (1987) 266–275.

[18] P.Wadler, R.J.M.Hughes: Projections for Strictness Analysis, *Proceedings Functional Programming Languages and Computer Architecture*, Springer Lecture Notes in Computer Science **274** (1987), 385–407.

Appendix: Correctness

So far we have only given informal explanations of the intended meaning of the (strictness) properties in $\mathbf{A}(t)$. Since the definition of the analysis presupposes a clear understanding of these meanings we shall begin by explaining some facets of the semantics, $\mathbf{S}$, and then define a safety predicate val_t.

The semantics of types, $\mathbf{S}(t)$, is given by the following definitions where $\mathbf{T}$ is the flat cpo of truth values ('true', 'false' and $\bot$) and $\mathbf{Z}$ is the flat cpo of integers ($\cdots$,-1,0,1,$\cdots$ and $\bot$):

$$\mathbf{S}(\texttt{Int}) = \mathbf{Z},\ \mathbf{S}(\texttt{Bool}) = \mathbf{T},\ \mathbf{S}(t_1 \times t_2) = (\mathbf{S}(t_1) \times \mathbf{S}(t_2))_\bot,$$
$$\mathbf{S}(t_1 \to t_2) = (\mathbf{S}(t_1) \to \mathbf{S}(t_2))_\bot,\ \mathbf{S}(t_1 \otimes t_2) = \mathbf{S}(t_1 \times t_2),\ \mathbf{S}(t_0\ \texttt{list}) = \mathbf{S}(t_0)^\infty$$

Here we note that the lifting used for product allows to distinguish between the completely undefined value ($\bot$) of a product type and the value being a pair of undefined values ($up((\bot,\bot))$). Similarly for functions we distinguish between the undefined function ($\bot$) and the function ($up(\lambda v.\bot)$) that always yields an undefined result when applied.

To explain the semantics of lists consider a partially ordered set $(D,\sqsubseteq)$ and how to define the partially ordered set $(D^\infty,\sqsubseteq)$ of *potentially infinite lists*. Let us say that a set of positive integers is *convex* if it equals $\{1,2,\cdots\}$ or if it equals $\{1,2,\cdots,n\}$ for some $n \geq 0$; we shall say that the set has supremum n exactly when it equals $\{1,2,\cdots,n\}$. Assuming that $\star$ is an element not in D we define

$$D^\infty = \{\ l{:}K \to D \cup \{\star\} \mid (K \text{ is a } \textit{convex} \text{ set of positive integers}) \wedge (\forall n \in K{:}\ l(n) = \star \Rightarrow n \text{ is the supremum of } K)\ \}$$

We shall feel free to write $\mathrm{dom}(l) = K$ when $l{:}K \to D \cup \{\star\}$ and we define $\mathrm{dom}^\star(l) = \{i \in \mathrm{dom}(l) \mid l(i) \neq \star\}$. Next define

$$l \sqsubseteq l' \text{ if and only if } (\ (\mathrm{dom}(l) \subseteq \mathrm{dom}(l')) \wedge (\forall n \in \mathrm{dom}(l){:}\ l(n) \sqsubseteq l'(n))\ )$$

where $l(\mathrm{n}) \sqsubseteq l'(\mathrm{n})$ implies that if one of $l(\mathrm{n})$ or $l'(\mathrm{n})$ is $\star$ then so is the other. — To allow for a more convenient notation for the elements of D^∞ we shall write the least element of D^∞ as $\bot$ and allow the usual notation involving ':', i.e. infix **cons**, for constructing elements.

Turning now to the issue of correctness we shall define a safety relation

$$val_t : \mathbf{S}(t) \times \mathbf{A}(t) \to \{\text{true,false}\}$$

by structural induction over types t. This technique is commonly called *logical relations* [16] although the use of tensor product gives a twist:

$$val_{\mathrm{Int}}(v,a) \equiv (a{=}0 \Rightarrow v{=}\bot)$$

$$val_{\mathrm{Bool}}(v,a) \equiv (a{=}0 \Rightarrow v{=}\bot) \wedge (a{=}T \Rightarrow v \sqsubseteq \mathrm{true}) \wedge (a{=}F \Rightarrow v \sqsubseteq \mathrm{false})$$

$$\begin{aligned} val_{t_1 \times t_2}(v,a) \equiv\ & (a{=}\bot \Rightarrow v{=}\bot) \wedge \\ & val_{t_1}(v_1,a_1) \wedge val_{t_2}(v_2,a_2) \\ & \text{where } (v_1,v_2) = dn(v) \\ & \text{and } (a_1,a_2) = dn(a) \end{aligned}$$

$$\begin{aligned} val_{t_1 \to t_2}(f,h) \equiv\ & (h{=}\bot \Rightarrow f{=}\bot) \wedge \\ & \forall v{\in}\mathbf{S}(t_1){:}\ \forall a{\in}\mathbf{A}(t_1){:} \\ & \quad val_{t_1}(v,a) \Rightarrow val_{t_2}(dn(f)(v),dn(h)(a)) \end{aligned}$$

$$\begin{aligned} val_{t_1 \otimes t_2}(v,a) \equiv\ & (a{=}\bot \Rightarrow v{=}\bot) \wedge \\ & (\exists (a_1,a_2){\in}dn(a){:}\ val_{t_1}(v_1,a_1) \wedge val_{t_2}(v_2,a_2) \\ & \text{where } (v_1,v_2) = dn(v)) \end{aligned}$$

$$\begin{aligned} val_{t\ \mathrm{list}}(vl,al) \equiv\ & (al{=}0 \Rightarrow vl{=}\bot) \wedge \\ & (al{=}1 \Rightarrow \forall \mathrm{i}{\in}\mathrm{dom}(vl){:}\ vl(\mathrm{i}){\neq}\star) \wedge \\ & (al \notin \{0,1,\top\varepsilon\} \wedge (\exists \mathrm{i}{\in}\mathrm{dom}(vl){:}\ vl(\mathrm{i}){=}\star) \\ & \Rightarrow \forall a{\in}dn(dn(al)){:}\ \exists \mathrm{i}{\in}\mathrm{dom}^*(vl){:}\ val_t(vl(\mathrm{i}),a)) \end{aligned}$$

The cases of base types, product types and function types should be rather straightforward. In the case of tensor product we use an existential quantifier to reflect that an element of the tensor product is a set of possible properties where only one of them needs to hold. The clause for lists formalises the meaning of properties of the form $al{=}Y\varepsilon$. It is here important to realise that for each property a of Y there must be some element of the concrete lists that enjoys that property. In the case where the type t of the elements has $\mathbf{A}(t)$ to be a chain (as for t one of `Int`, `Int list` etc.) this is equivalent to Wadler's requirement that the 'meet' of all the elements in the concrete list must be described by a. As was also observed in [5] this does not make sense in general, hence our use of a universal quantifier.

The safety predicate val_t enjoys a number of properties that are indicative of what one would expect to hold for an arbitrary analysis. (Just think of $val_t(v,a)$ as a shorthand for $\beta_t(v) \sqsubseteq a$ for a sufficiently well-behaved function β_t, i.e. one that is strict and continuous and maps compact elements to compact elements[4].)

[4]**Note on terminology:** Our β_t corresponds to *abs* of [3]; in an analogous way the α and γ of [11, 14] correspond to *Abs* and *Conc*, respectively, of [3]. (The use of α and γ is motivated by the notation in the original papers on abstract interpretation, e.g. [4].)

Lemma 17 The above clauses define an admissible (or inductive) predicate

$$val_t : \mathbf{S}(t) \times \mathbf{A}(t) \to \{\text{true,false}\}$$

that enjoys the following properties:

$$\forall a{\in}\mathbf{A}(t){:}\ val_t(\bot_{\mathbf{S}(t)},a)$$
$$\forall v{\in}\mathbf{S}(t){:}\ val_t(v,\top_{\mathbf{A}(t)})$$
$$\forall v{\in}\mathbf{S}(t){:}\ \forall a_1,a_2{\in}\mathbf{A}(t){:}\ val_t(v,a_1) \wedge a_1 \sqsubseteq a_2 \Rightarrow val_t(v,a_2)$$
$$\forall v{\in}\mathbf{S}(t){:}\ \forall a_1,a_2{\in}\mathbf{A}(t){:}\ val_t(v,a_1) \wedge val_t(v,a_2) \Rightarrow val_t(v,a_1 \sqcap a_2)$$
$$\forall v_1,v_2{\in}\mathbf{S}(t){:}\ \forall a{\in}\mathbf{A}(t){:}\ v_1 \sqsubseteq v_2 \wedge val_t(v_2,a) \Rightarrow val_t(v_1,a)$$ □

The safety predicate val_t also enjoys another property that only holds because we were careful to use lifting when interpreting $\times$ and $\to$. It is a key result for the analysis **A** to be useful for optimizations based on strictness analysis. (In terms of the function β_t mentioned above just suppose that it reflects $\bot$, i.e. $\beta_t(v){=}\bot \Rightarrow v{=}\bot$.)

Lemma 18 $\forall v{\in}\mathbf{S}(t){:}\ val_t(v,\bot_{\mathbf{A}(t)}) \Rightarrow v{=}\bot$ □

The correctness of the analysis **A** with respect to the standard semantics **S**, in the sense of the *val* predicate, is demonstrated in [15, Chapter 7]. One of the more interesting ingredients in this proof is the correctness of **A**(case).

Lemma 19 (Correctness of **A**(case)) Whenever

$$val_{t_0\text{list}\to t_1}(f_1,h_1)$$
$$val_{(t_0\times(t_0\text{list}))\to t_1}(f_2,h_2)$$

we also have

$$val_{(t_0\text{list})\to t_1}(\mathbf{S}(\mathsf{case})\ f_1\ f_2,\ \mathbf{A}(\mathsf{case})\ h_1\ h_2)$$

(for an 'obvious' definition of **S**(case)). □

Proof: For the proof we shall assume that

$$val_{t_0\text{list}\to t_1}(f_1,h_1)$$
$$val_{(t_0\times(t_0\text{list}))\to t_1}(f_2,h_2)$$
$$val_{t_0\text{list}}(v,a)$$

and that none of f_1, f_2, h_1 or h_2 equals $\bot$. The definition of $\mathbf{A}(\mathsf{case})(h_1)(h_2)$ applied to a then amounts to a case analysis upon the strictness property a.

If $a{=}0$ we know that the list v is $\bot$ so that $\mathbf{S}(\mathsf{case})(f_1)(f_2)$ applied to v gives $\bot_{\mathbf{S}(t_1)}$. It is therefore natural to use the strictness property $\bot_{\mathbf{A}(t_1)}$.

If $a{=}1$ we know that the list v is infinite or partial. Hence any element v' of $\mathbf{S}(t_0)$ may be the head of v (unless v is $\bot$) and the tail v'' of v will still be infinite or partial. Hence $\top_{\mathbf{A}(t_0)}$ aptly describes v' and 1 aptly describes v'' so that

$$dn(h_2)(up((\top_{\mathbf{A}(t_0)},1)))$$

aptly describes $dn(f_2)(up((v',v'')))$ as well as $\bot_{\mathbf{S}(t_1)}$ (in case v is $\bot$).

If $a{=}\top\varepsilon$ we know nothing about the list v; it may be the empty list [], its head v' may be any element of $\mathbf{S}(t_0)$ and its tail v'' may be any list of $\mathbf{S}(t_0\ \texttt{list})$. Thus

$$dn(h_1)(\top\varepsilon)$$

aptly describes $dn(f_1)([])$ and

$$dn(h_2)(up((\top,\top\varepsilon)))$$

aptly describes $dn(f_2)(up((v',v'')))$ as well as $\bot_{\mathbf{S}(t_1)}$. By using the least upper bound we obtain a strictness property that aptly describes both possibilities.

Finally consider the case where $a{=}Y\varepsilon$ and $Y\varepsilon \neq \top\varepsilon$; we then know that the list v cannot be []. It therefore might be natural to use the strictness property

$$dn(h_2)(up((\top,\top\varepsilon)))$$

since indeed the head v' of the list v may be any element of $\mathbf{S}(t_0)$. However, the snag is that the tail v'' cannot necessarily be any list of $\mathbf{S}(t_0\ \texttt{list})$ because there are certain constraints from Y that may still have to be satisfied. Thus while $dn(h_2)(up((\top,\top\varepsilon)))$ would not be incorrect we shall be able to do better.

Consider the situation where v is a finite list; since v is not $\bot$ it will be of the form $v{=}v'{:}v''$. We then have a mapping

$$\jmath : Y \to \mathrm{dom}^*(v)$$

such that $val_{t_0}(v(\jmath(a)),a)$ holds for all $a{\in}Y$. We now have a number of possibilities concerning

$$Y' = \{a{\in}Y \mid \jmath(a){=}1\}$$

For each of these we shall argue that

$$\forall a{\in}Y'\colon\ val_{t_0}(v',a)$$
$$val_{t_0\texttt{list}}(v'',(Y{\ominus}Y')\varepsilon)$$

The first of these is immediate and gives

$$val_{t_0}(v',{\sqcap}Y')$$

using Lemma 17 where we set ${\sqcap}\emptyset{=}\top$. The second of these is immediate if $Y{\ominus}Y' = \{\top\}$; so assume that $Y{\ominus}Y'{\neq}\{\top\}$ and note that $\mathrm{RC}(Y')$ then is a proper subset of Y. For each $a{\in}Y{\setminus}\mathrm{RC}(Y')$ we have $a{\notin}Y'$ and hence $\jmath(a){\neq}1$. Thus

$$\jmath'(a) = \jmath(a)\text{-}1$$

defines a mapping

$$\jmath' : (Y{\setminus}\mathrm{RC}(Y')) \to \mathrm{dom}^*(v'')$$

such that $val_{t_0}(v''(\jmath'(a)),a)$ holds for all $a{\in}Y{\setminus}\mathrm{RC}(Y')$. This mapping may be extended (in at least one way) to a mapping

$$j'' : Y \ominus Y' \to \mathrm{dom}^*(v'')$$

such that $val_{t_0}(v''(j''(a)),a)$ holds for all $a \in Y \ominus Y'$.

Returning to each choice of $Y' \subseteq Y$ we now have a contribution

$$dn(h_2)(up((\sqcap Y', (Y \ominus Y')\varepsilon)))$$

and by taking the least upper bound of all of these we aptly describe all possibilities. Actually we may assume that Y' is nonempty, or $Y' \ni \top$, as $\sqcap\emptyset = \sqcap\{\top\}$ and $Y \ominus \emptyset = Y \ominus \{\top\}$, and therefore no contributions will be missed. Furthermore one may assume that Y' is right-closed as $\sqcap Y' = \sqcap \mathrm{RC}(Y')$ and $Y \ominus Y' = Y \ominus \mathrm{RC}(Y')$, and therefore no contributions will be missed. In summary we only need to consider those $Y' \in \mathcal{O}(\mathbf{A}(t_0))$ such that $Y' \subseteq Y$. □