Abstract Interpretation over Algebraic Data Types

Thomas P. Jensen*
CNRS-LIX
Ecole Polytechnique
91128 Palaiseau, France
Email: jensen@lix.polytechnique.fr

Abstract

This paper is concerned with the static analysis of programs over recursive data structures such as lists and trees. In particular, we consider the analysis of uniform properties i.e., *properties pertaining only to the content of a data structure. We first present an axiomatic description of properties of sum types and algebraic types and use the theory of powerdomains to construct lattices modelling the logic of the axiomatisations. In addition to a new analysis of sum types based on logic, this provides a systematic way of defining abstract lattices for arbitrary algebraic data types. We provide a detailed description of the lattice for analysing lists and show how our developments generalise existing frameworks proposed by Wadler and Nielson and Nielson. Finally, we show how abstract interpretations of well known list operations can be defined over these lattices.*

1 Introduction

This paper is concerned with the analysis of *uniform* properties of those recursive data types called algebraic data types. A property is uniform if it only describes the content, and not the structure, of a data object. Thus for lists the property *"contains an undefined element"* is uniform whereas *"the first element is undefined"* is not. For binary trees of integers, *"all nodes are greater than 5"* is a uniform property whereas *"is a heap"* is not. Otherwise said we consider properties of the multi-set of elements contained in a data structure. In this paper we restrict attention to two kinds of properties: $\Diamond\varphi$ and $\Box\varphi$, where φ is a property of the elements of the structure. The property $\Diamond\varphi$ is satisfied by those structures that contain an element satisfying φ and the property $\Box\varphi$ is satisfied by structures where all elements satisfy φ.

*This research was supported by DIKU, University of Copenhagen and ESPRIT BRA WG 6809 – SEMANTIQUE.

A commonly encountered recursive data structure is the list. The analysis of list structures has been considered by a number of people [25, 21, 26, 12, 6, 11, 16, 15, 22, 24, 19, 4]. Wadler [25] considers strictness properties of lists such as *"the list contains an undefined element"* and *"the list is partial or infinite"*. Launchbury's binding-time analysis for partial evaluation [21] operates on properties such as *"all elements of a list can be evaluated by the partial evaluator"*. Ernoult and Mycroft [11] introduced uniform ideals of domains to model uniform strictness properties of lists. The purpose of the work reported here is to describe some of these analyses in a common framework based on a simple program logic of uniform properties. This includes analyses based on the four-point domain of Wadler and generalisations thereof but not analyses of structural properties like head-strictness [26] or Burn's H_B-projection [6].

We develop program analyses by giving an axiomatic description of the properties analysed for. This description takes the form of a logical inference system defining an entailment relation $\leq$ between formulae denoting properties. For our uniform properties we have axioms such as

$$\Box(\varphi \wedge \psi) \leq \Box\varphi \wedge \Box\psi.$$

The intuition behind this axiom is that if all elements in a multi-set satisfy φ and ψ then they all satisfy φ and they all satisfy ψ. In fact we shall also have the axiom

$$\Box\varphi \wedge \Box\psi \leq \Box(\varphi \wedge \psi)$$

thus the two formulae are provably equal and therefore denote the same property. In order to obtain the set of properties axiomatised by $\leq$ we form the quotient structure where provably equal formulae are grouped into equivalence classes which are ordered by the partial order $\sqsubseteq$ defined by $\leq/=$.

We can then define the abstract interpretation of a program p as a function $[\![p]\!]$ over these partial orders of properties such that if input satisfying (the property denoted by) φ is mapped to output satisfying ψ then $[\psi] \sqsubseteq [\![p]\!]([\varphi])$. For example the length function is undefined when the input is a partial or infinite list, so the abstract interpretation of $[\![\mathsf{length}]\!]$ applied to the property ∞ denoting *"partial or infinite list"* should return the property *"undefined"*. More precisely, for the list type $\mu\alpha.\sigma \times \alpha + \mathbf{1}$ we have properties of the form $\mathsf{in}_1(\varphi_1) + \mathsf{in}_2(\varphi_2)$ where φ_1 describes the set of σ-elements in the list and φ_2 describes the set of $\mathbf{1}$-elements. This latter piece of information contains some information about the structure of the list: if the set of elements from the $\mathbf{1}$ summand is empty the list is not well terminated *i.e.*, the list is either infinite or partial. In this way we obtain some structural information about the list from the description of the content.

Section 2 reviews the type system from [18] for disjunctive strictness logic. This logic is extended in Section 3 with an axiomatisation of the uniform properties $\Diamond\varphi$ and $\Box\varphi$. In Section 4 we review the principles of deriving abstract lattices from axiomatic systems and show how techniques from powerdomain theory enable us to construct lattices of uniform properties. Section 5 defines an analysis of sum types. Section 6 introduces algebraic types and show how a logic for sums can be combined with the axiomatisation for uniform properties to yield a system of uniform properties of algebraic types. We give inference rules for reasoning about the basic operators over these types and propose a general definition of abstract lattices for algebraic data types. This definition is used in Section 7 where we construct the lattice of uniform properties over integer lists and compare it to Wadler's four-point domain and its generalisation suggested by Nielson and Nielson. Section 8 gives the abstract interpretation of some list operations and show that more precise abstract interpretation are obtained by our approach. Familiarity with the basic notions of static program analysis and abstract interpretation is assumed; for an overview see [2, 9].

2 Type systems for program analysis

This section describes a generic formal system for reasoning about static properties of programs. We define for each type

$$\sigma = \mathsf{Int} \mid \mathsf{Bool} \mid \sigma \times \tau \mid \sigma \to \tau$$

in our language a set of formulae $L(\sigma)$, denoting properties of the elements of that type, and axiomatise the logical structure on these sets of formulae. The sets of formulae are defined inductively over the type structure as follows:

Formulae

$$\mathtt{f}, \mathtt{t} \in L(\sigma) \qquad \frac{\varphi, \psi \in L(\sigma)}{\varphi \vee \psi \in L(\sigma) \quad \varphi \wedge \psi \in L(\sigma)}$$

$$\frac{\varphi \in L(\sigma), \psi \in L(\tau)}{\varphi \times \psi \in L(\sigma \times \tau)} \qquad \frac{\varphi \in L(\sigma), \psi \in L(\tau)}{\varphi \to \psi \in L(\sigma \to \tau)}$$

The formal system is *generic* because it can be used to define several analyses: in strictness analysis [7] the formula $\mathtt{f}$ denotes the property *"is undefined"* whereas for a binding time analysis [20] it represents the property "static". The formula $\mathtt{t}$ denotes the trivial property satisfied by all expressions. The formula $\phi \times \psi$ describes a pair where the first (resp. the second) component satisfies the property ϕ (resp. ψ). As a property of a function we take $\phi \to \psi$ to mean that the function maps arguments satisfying ϕ to results satisfying ψ.

The axiomatisation consists of two sets of rules: The logical rules define the generic logical connectives $\wedge, \vee, \mathtt{f}, \mathtt{t}$ in terms of the entailment relation $\leq$. The type–specific rules define the logic governing the properties of objects of product and function types. This axiomatisation defines a logical system

$$\mathcal{L}(\sigma) = (L(\sigma), \wedge, \vee, \mathtt{f}, \mathtt{t}, \leq, =)$$

where $=$ is provable equality *i.e.*, $=$ is the symmetric closure of $\leq$. The axiomatisation is shown in Figure 1. Most of the rules have a straightforward logical reading. For example, saying of a function f that it maps arguments satisfying either φ_1 or φ_2 to results satisfying ψ is the same as saying that f maps arguments satisfying φ_1 to ψ and f maps arguments satisfying φ_2 to ψ. This is the content of the rule

$$(\varphi_1 \to \psi) \wedge (\varphi_2 \to \psi) = (\varphi_1 \vee \varphi_2) \to \psi$$

In order to state a Normal Form Theorem for the axiomatisation we define the *irreducible* formulae to be those formulae where disjunctions only appear on the right-hand side of an arrow $\to$. The set of irreducible formulae of $L(\sigma)$ is denoted by $\mathsf{Irr}(\sigma)$ and can be defined inductively in the same way as $L(\sigma)$ [18]. The formal logic then satisfies the following theorem:

Theorem 2.1 (Normal Form) *For all $\varphi \in L(\sigma)$ there exist $\psi_1, \ldots \psi_n \in \mathsf{Irr}(\sigma)$ such that*

$$\varphi = \bigvee_{i=1}^{n} \psi_i \qquad n \geq 1$$

This theorem will be of importance when we describe the ordering $\leq$ on formulae in Section 4.

Logical rules

- $\varphi \leq \varphi$
- $\dfrac{\varphi \leq \psi, \psi \leq \chi}{\varphi \leq \chi}$
- $\varphi \wedge \psi \leq \varphi$
- $\dfrac{\varphi \leq \psi_1, \varphi \leq \psi_2}{\varphi \leq \psi_1 \wedge \psi_2}$
- $\varphi \wedge \psi \leq \psi$
- $\varphi \leq \psi \vee \varphi$
- $\dfrac{\varphi_1 \leq \psi, \varphi_2 \leq \psi}{\varphi_1 \vee \varphi_2 \leq \psi}$
- $\psi \leq \psi \vee \varphi$
- $\dfrac{\varphi \leq \psi, \psi \leq \varphi}{\varphi = \psi}$
- $\dfrac{\varphi = \psi}{\varphi \leq \psi, \psi \leq \varphi}$
- $\varphi \leq \mathtt{t}$
- $\mathtt{t}_{\sigma\rightarrow\tau} = \mathtt{t}_\sigma \rightarrow \mathtt{t}_\tau$
- $\mathtt{t}_{\sigma\times\tau} = \mathtt{t}_\sigma \times \mathtt{t}_\tau$
- $\mathtt{f} \leq \varphi$
- $\mathtt{t}_\sigma \rightarrow \mathtt{f}_\tau = \mathtt{f}_{\sigma\rightarrow\tau}$
- $\mathtt{f}_\sigma \times \mathtt{f}_\tau = \mathtt{f}_{\sigma\times\tau}$
- $\varphi \wedge (\psi_1 \vee \psi_2) = (\varphi \wedge \psi_1) \vee (\varphi \wedge \psi_2)$

Type–specific rules

- $(\varphi_1 \times \psi) \vee (\varphi_2 \times \psi) = (\varphi_1 \vee \varphi_2) \times \psi$
- $(\varphi \times \psi_1) \vee (\varphi \times \psi_2) = \varphi \times (\psi_1 \vee \psi_2)$
- $(\varphi_1 \times \psi_1) \wedge (\varphi_2 \times \psi_2) = (\varphi_1 \wedge \varphi_2) \times (\psi_1 \wedge \psi_2)$
- $\dfrac{\varphi_1 \leq \varphi_2, \psi_1 \leq \psi_2}{\varphi_1 \times \psi_1 \leq \varphi_2 \times \psi_2}$
- $(\varphi \rightarrow \psi_1) \wedge (\varphi \rightarrow \psi_2) = \varphi \rightarrow (\psi_1 \wedge \psi_2)$
- $(\varphi_1 \rightarrow \psi) \wedge (\varphi_2 \rightarrow \psi) = (\varphi_1 \vee \varphi_2) \rightarrow \psi$
- $\dfrac{\varphi_2 \leq \varphi_1, \psi_1 \leq \psi_2}{\varphi_1 \rightarrow \psi_1 \leq \varphi_2 \rightarrow \psi_2}$

Figure 1: Axiomatisation of properties

2.1 Assigning strictness types to terms

The type system in Figure 2 defines a strictness analysis of a simply typed lambda calculus extended with conditionals and fixed points. The logic operates with basic assertions of the form $\Gamma \vdash e : \varphi$ where Γ is an environment associating variables with formulae, e a term of type σ and φ a formula in $L(\sigma)$. As an example of a derivation we show that the function

$$\mathsf{g} \equiv \lambda x.\mathsf{if}\ B\ \mathsf{then}\ (x, 17)\ \mathsf{else}\ (17, x)$$

satisfies the property:

$$\mathtt{f} \rightarrow (\mathtt{f} \times \mathtt{t} \vee \mathtt{t} \times \mathtt{f}).$$

Let Γ be the environment $[x \mapsto \mathtt{f}]$. Using **Var, Taut, Pair, If-2** and **Abs** we get the deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \mathtt{f}} \quad \dfrac{}{\Gamma \vdash 17 : \mathtt{t}}}{\Gamma \vdash (x, 17) : \mathtt{f} \times \mathtt{t}} \qquad \dfrac{\dfrac{}{\Gamma \vdash x : \mathtt{f}} \quad \dfrac{}{\Gamma \vdash 17 : \mathtt{t}}}{\Gamma \vdash (17, x) : \mathtt{t} \times \mathtt{f}}}{\Gamma \vdash \mathsf{if}\ B\ \mathsf{then}\ (x, 17)\ \mathsf{else}\ (17, x)) : \mathtt{f} \times \mathtt{t} \vee \mathtt{t} \times \mathtt{f}}}{\vdash \lambda x.\mathsf{if}\ B\ \mathsf{then}\ (x, 17)\ \mathsf{else}\ (17, x) : \mathtt{f} \rightarrow (\mathtt{f} \times \mathtt{t} \vee \mathtt{t} \times \mathtt{f})}$$

The example demonstrates the extra power obtained by adding disjunctions to the conjunctive logic of [17]. In particular we can state and prove that at least one of the components of the result of g will be undefined. This fact cannot be expressed using only conjunctions. It also demonstrates a limitation of the logic: the identity

$$\mathtt{f} \rightarrow (\mathtt{f} \times \mathtt{t} \vee \mathtt{t} \times \mathtt{f}) = \mathtt{f} \rightarrow (\mathtt{f} \times \mathtt{t}) \vee \mathtt{f} \rightarrow (\mathtt{t} \times \mathtt{f})$$

is semantically sound but it cannot be derived in the system presented here. An axiomatisation including this axiom is considered by Benton [5].

3 Uniform properties

Uniform properties describe the content of a data structure, thus, from this point of view, a data structure is just a multiset of data objects. We introduce two kinds of properties for describing non-empty multisets of elements of a given type σ. If φ is a formula of type σ then $\Box\varphi$ holds for those sets where *all* elements

$$\textbf{Conj}\quad \frac{\Gamma \vdash e : \psi_1 \quad \Gamma \vdash e : \psi_2}{\Gamma \vdash e : \psi_1 \wedge \psi_2} \qquad \textbf{Disj}\quad \frac{\Gamma[x \mapsto \phi_1] \vdash e : \psi \quad \Gamma[x \mapsto \phi_2] \vdash e : \psi}{\Gamma[x \mapsto \phi_1 \vee \phi_2] \vdash e : \psi}$$

$$\textbf{Taut}\quad \Gamma \vdash e : \mathtt{t} \qquad \textbf{Weak}\quad \frac{\Gamma \leq \Delta \quad \Delta \vdash e : \phi \quad \phi \leq \psi}{\Gamma \vdash e : \psi}$$

$$\textbf{Var}\quad \Gamma[x \mapsto \phi] \vdash x : \phi \qquad \textbf{Pair}\quad \frac{\Gamma \vdash e_1 : \phi_1 \quad \Gamma \vdash e_2 : \phi_2}{\Gamma \vdash (e_1, e_2) : \phi_1 \times \phi_2}$$

$$\textbf{Fst}\quad \frac{\Gamma \vdash e : \varphi \times \psi}{\Gamma \vdash \mathsf{fst}(e) : \varphi} \qquad \textbf{Snd}\quad \frac{\Gamma \vdash e : \varphi \times \psi}{\Gamma \vdash \mathsf{snd}(e) : \psi}$$

$$\textbf{Abs}\quad \frac{\Gamma[x \mapsto \phi] \vdash e : \psi}{\Gamma \vdash \lambda x.e : (\phi \rightarrow \psi)} \qquad \textbf{App}\quad \frac{\Gamma \vdash e_1 : (\phi \rightarrow \psi) \quad \Gamma \vdash e_2 : \phi}{\Gamma \vdash e_1 e_2 : \psi}$$

$$\textbf{Fix}\quad \vdash \mathsf{fix} : (\varphi \rightarrow \varphi) \rightarrow \varphi$$

$$\textbf{If-1}\quad \frac{\Gamma \vdash b : \mathtt{f}}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathtt{f}} \qquad \textbf{If-2}\quad \frac{\Gamma \vdash e_1 : \phi_1 \quad \Gamma \vdash e_2 : \phi_2}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \phi_1 \vee \phi_2}$$

Figure 2: Disjunctive strictness logic

satisfy φ. Similarly $\Diamond\varphi$ holds for those sets where at least one element satisfies φ. We let $\mathcal{P}(\sigma)$ denote the type of multisets with elements of type σ and define the set $L(\mathcal{P}(\sigma))$ of formulae over $\mathcal{P}(\sigma)$ as follows:

Formulae

- $\mathtt{t}, \mathtt{f} \in L(\mathcal{P}(\sigma))$
- $\dfrac{\varphi, \psi \in L(\mathcal{P}(\sigma))}{\varphi \wedge \psi, \varphi \vee \psi \in L(\mathcal{P}(\sigma))}$
- $\dfrac{\varphi \in L(\sigma)}{\Box\varphi, \Diamond\varphi \in L(\mathcal{P}(\sigma))}$

The entailment relation $\leq$ defined in the previous section is extended to uniform properties by the axioms and rules below. The axiomatisation is adopted from the logical description of the Plotkin powerdomain by Abramsky [1] which itself has its origin in modal logic.

Axioms

$(\Box - \wedge)$	$\Box(\varphi \wedge \psi) = \Box\varphi \wedge \Box\psi$
$(\Box - \vee)$	$\Box(\varphi \vee \psi) \leq \Box\varphi \vee \Diamond\psi$
$(\Diamond - \wedge)$	$\Box\varphi \wedge \Diamond\psi \leq \Diamond(\varphi \wedge \psi)$
$(\Diamond - \vee)$	$\Diamond(\varphi \vee \psi) = \Diamond\varphi \vee \Diamond\psi$
$(\Box - \mathtt{t})$	$\Box\mathtt{t} = \mathtt{t}$
$(\Box - \mathtt{f})$	$\Box\mathtt{f} = \mathtt{f}$

Rules

$$(\Box - \leq)\quad \frac{\varphi \leq \psi}{\Box\varphi \leq \Box\psi} \qquad (\Diamond - \leq)\quad \frac{\varphi \leq \psi}{\Diamond\varphi \leq \Diamond\psi}$$

The axioms and rules can be understood intuitively in terms of the content of a multiset. The axiom $\Box - \vee$ for example states that if all elements of a set satisfies either φ or ψ then either they all satisfy φ or there exists an element satisfying ψ. Similarly, $\Diamond - \wedge$ can be justified as follows: if all elements satisfy φ and there furthermore exists an element satisfying ψ then that element satisfies both φ and ψ so the set satisfies $\Diamond(\varphi \wedge \psi)$. The formula $\Box\mathtt{t}$ expresses that all elements satisfy the trivial property $\mathtt{t}$; the axiom $\Box\mathtt{t} = \mathtt{t}$ states that this is the trivial statement about the content of a multiset. Similarly, $\Box\mathtt{f} = \mathtt{f}$ can be understood as saying that the most exact information about a multiset is obtained if we have the most exact information about every element in the multiset. As a first example of the use of the axioms we have that

- $\Box\varphi = \Box(\mathtt{f} \vee \varphi) \leq \Box\mathtt{f} \vee \Diamond\varphi = \mathtt{f} \vee \Diamond\varphi = \Diamond\varphi$
- $\mathtt{t} = \Box\mathtt{t} = \Box(\mathtt{f} \vee \mathtt{t}) \leq \Box\mathtt{f} \vee \Diamond\mathtt{t} = \mathtt{f} \vee \Diamond\mathtt{t} = \Diamond\mathtt{t}$

proving the theorems $\Box\varphi \leq \Diamond\varphi$ and $\Diamond\mathtt{t} = \mathtt{t}$.

Formulated in words these theorems state that *"if all elements in a set satisfy φ then there exists an element in that set satisfying φ"* and *"there exists an element in the set satisfying* $\mathtt{t}$*"*. Neither of these statements hold for the empty set $\emptyset$ so we see that the sets we are axiomatising are *non-empty*. We later show (Section 6) how the empty set can be accommodated in the analyses.

The following theorem extends the Normal Form Theorem from Section 2 and plays an important part in deriving the lattices modelling the uniform properties. Recall that $\mathsf{Irr}(\sigma)$ is the set of formulae built without disjunction.

Theorem 3.1 *Every formulae in $L(\mathcal{P}(\sigma))$ is provably equal to a finite disjunction of formulae of the form*

$$\Box(\bigvee_{i=1}^{n} \varphi_i) \wedge \bigwedge_{i=1}^{n}(\Diamond\varphi_i)$$

where all $\varphi_i \in \mathsf{Irr}(\sigma)$.

Proof. For a proof see *e.g.*, Abramsky [1, Prop. 3.4.8].

4 From logics to lattices

We first outline the principles for turning an axiomatic system into a domain for abstract interpretation. Then we apply this to the system of uniform properties using techniques from powerdomain theory. The implication ordering $\leq$ from the formal systems $(L(\sigma), \wedge, \vee, \mathtt{f}, \mathtt{t}, \leq, =)$ defined in the previous sections is only a preorder since there are syntactically distinct formulae that imply each other *i.e.*, are logically equivalent. As an example we have $\mathtt{t} \wedge \mathtt{t} \leq \mathtt{t}$ and $\mathtt{t} \leq \mathtt{t} \wedge \mathtt{t}$. Logically equivalent formulae denote the same *property* so we can obtain the structure of properties axiomatised by the formal systems as the quotient structures $\mathcal{LA}(\sigma) = (L(\sigma)/=, \wedge/=, \vee/=, [\mathtt{f}], [\mathtt{t}], \leq/=)$, called the Lindenbaum algebra of the formal system. We use $[\varphi]$ to denote the equivalence class containing φ and $\sqsubseteq$ to denote the partial order $\leq/=$.

The disjunctive normal form leads to a concrete representation of $\mathcal{LA}(\sigma)$ constructed as follows:

1. For each σ, let $\mathcal{J}(\sigma)$ be $(\mathsf{Irr}(\sigma)/=, \sqsubseteq)$, the partial order restricted to irreducible formulae.

2. Define $[\![\sigma]\!]$ to be the set $\mathsf{D}(\mathcal{J}(\sigma))$ of downwards closed sets of $\mathcal{J}(\sigma)$, ordered by set inclusion $\subseteq$.

Standard results [10, 8] now yields that $[\![\sigma]\!]$ is a complete lattice. For the system in Section 2 it can be shown [18] that with $\mathcal{J}(\mathsf{Int}) = \mathbf{2}$, $\mathcal{J}(\sigma \times \tau) = \mathcal{J}(\sigma) \times \mathcal{J}(\tau)$ and $\mathcal{J}(\sigma \to \tau)$ the set of monotone functions from $\mathcal{J}(\sigma)$ to $\mathcal{J}(\tau)$, $[\![\sigma]\!]$ is isomorphic to $\mathcal{LA}(\sigma)$. The element in $[\![\sigma]\!]$ corresponding to a formula φ will be denoted by $[\![\varphi]\!]$.

The upshot of having the lattice of properties $[\![\sigma]\!]$ is that we can define a denotational semantics over $[\![\sigma]\!]$, assigning to each expression e of type σ an *abstract interpretation* $[\![e]\!] \in [\![\sigma]\!]$ such that $[\![e]\!]$ is a property of e. In [18] it is shown how to define an abstract interpretation $[\![\_]\!]$ that gives the *strongest* property provable of e in the type system of Figure 2. We thus have the pleasing equivalence between logic and abstract interpretation

$$[\![e]\!] \sqsubseteq [\![\varphi]\!] \qquad \Leftrightarrow \qquad \vdash e : \varphi.$$

4.1 Uniform properties and powerdomains

In this section we apply ideas from powerdomain theory to construct abstract lattices of uniform properties in the manner outlined above. It builds on the work by Abramsky [1] on characterising powerdomains logically which again comes from Winskel's original work on the relationship between modalities and powerdomains [27]. A formula

$$\Box(\bigvee_{i\in I} \varphi_i) \wedge \bigwedge_{i\in I}(\Diamond\varphi_i) \qquad \varphi_i \in \mathsf{Irr}(L(\sigma))$$

describes those sets where every element satisfies $\bigvee_i \varphi_i$ and for every property φ_i there exists at least one element that satisfies this property. Assuming that the φ_i's denote properties $[\varphi_i]$ in some lattice $\mathcal{J}(\sigma)$ we can represent such a formula by the set of elements $\{[\varphi_i] \mid i \in I\}$. We then need to find an ordering on these sets that reflects the entailment ordering between the uniform properties represented by the sets.

Given two sets $\{\varphi_i\}_{i\in I}, \{\psi_j\}_{j\in J} \subseteq \mathsf{Irr}(\sigma)$ we have

$$\Box(\bigvee_{i\in I} \varphi_i) \wedge \bigwedge_{i\in I} \Diamond\varphi_i \leq \Box(\bigvee_{j\in J} \psi_j) \wedge \bigwedge_{j\in J} \Diamond\psi_j$$

if and only if

$$\bigvee_{i\in I} \varphi_i \leq \bigvee_{j\in J} \psi_j \quad \text{and} \quad \forall j \in J\ \exists i \in I.\ \varphi_i \leq \psi_j$$

i.e., if and only if

$$\forall i \in I\ \exists j \in J.\ \varphi_i \leq \psi_j \quad \text{and} \quad \forall j \in J\ \exists i \in I.\ \varphi_i \leq \psi_j.$$

This last condition gives rise to the *Egli-Milner ordering* on subsets of a partial order.

Definition 4.1 *The Egli-Milner ordering $\sqsubseteq_{EM}$ on non-empty subsets of a partial order $(P, \sqsubseteq)$ is defined by: $S \sqsubseteq_{EM} T$ iff*

$$\forall s \in S\ \exists t \in T.\ s \sqsubseteq t \quad \textit{and} \quad \forall t \in T\ \exists s \in S.\ s \sqsubseteq t.$$

Furthermore define

$$S \approx_{EM} T \equiv S \sqsubseteq_{EM} T \textit{ and } T \sqsubseteq_{EM} S$$

The Egli-Milner ordering is in general only a preorder on subsets. As an example take the partial order $\mathbf{3} = \{0 \sqsubseteq 1 \sqsubseteq 2\}$, where we have $\{0,2\} \sqsubseteq_{EM} \{0,1,2\}$ and $\{0,1,2\} \sqsubseteq_{EM} \{0,2\}$. In order to obtain a partial order we consider only *convex* sets.

Definition 4.2 *A subset S of a partial order P is* convex *if*

$$\forall s_1, s_2 \in S\ \forall p \in P. s_1 \sqsubseteq p \sqsubseteq s_2 \Rightarrow p \in S.$$

The convex closure *of a non-empty set S is the least (with respect to $\subseteq$) convex set that contains S. We write this set as $\overline{\overline{S}}$.*

It is straightforward to see that for a subset S we always have $S \sqsubseteq_{EM} \overline{\overline{S}}$ and $\overline{\overline{S}} \sqsubseteq_{EM} S$ From this it follows that the partial order of convex sets, ordered by $\sqsubseteq_{EM}$, is order-isomorphic to the preorder of subsets ordered by $\sqsubseteq_{EM}$, quotiented by $\approx_{EM}$

Definition 4.3 *For P a partial order define $P_C(P)$, the partial order of convex sets of P, by*

$$P_C(P) = (\{\overline{\overline{S}} \mid \emptyset \neq S \subseteq P\}, \sqsubseteq_{EM}).$$

Note that all sets in $P_C(P)$ are by definition non-empty.

As a simple example of how uniform properties are represented by convex sets we consider uniform properties of collections of integer elements. We assume that we have formulae $\mathtt{t}$ and $\mathtt{f}$ denoting properties of integers where $\mathtt{f} \leq \mathtt{t}$. If the interpretation of the type Int is the lattice $\mathcal{J}(\mathsf{Int}) = \{0 \sqsubseteq 1\}$ the set of convex sets $\mathcal{J}(\mathcal{P}(\mathsf{Int}))$ is given by

$$\mathcal{J}(\mathcal{P}(\mathsf{Int})): \quad \{0\} \sqsubseteq_{EM} \{0,1\} \sqsubseteq_{EM} \{1\}.$$

Here the top element $\{1\}$ represents $\Box\mathtt{t} \wedge \Diamond\mathtt{t}$. This contains no information: it just says that all elements satisfy the trivial property. The smallest element $\downarrow\{\{0\}\}$ representing $\Box\mathtt{f} \wedge \Diamond\mathtt{f}$ states that all elements satisfy the strongest property (*e.g.*, the property "completely undefined" in strictness analysis or "static" in binding time analysis). The middle point $\downarrow\{\{0,1\}\}$ is the interpretation of the formula $\Box(\mathtt{t} \vee \mathtt{f}) \wedge \Diamond\mathtt{t} \wedge \Diamond\mathtt{f} = \Box\mathtt{t} \wedge \Diamond\mathtt{f}$. The minimal element 0 models the $\Diamond$ part and the maximal element 1 the $\Box$ part of the formula. Notice the difference between the ordering $\sqsubseteq_{EM}$ and the subset ordering: by removing 1 from $\downarrow\{\{0,1\}\}$ we get a set that is smaller in the $\sqsubseteq_{EM}$ ordering, whereas by removing 0 from $\downarrow\{\{0,1\}\}$ we get a set that is bigger.

The logic and the interpretation can be modified according to the kind of properties we are interested in. For example we might want to consider only the $\Diamond\varphi$ set of properties. We mention briefly that the $\Diamond$-fragment of our logic can be modelled using upwards closed sets (*cf.* the Smyth powerdomain) and the $\Box$-fragment can be modelled by downwards closed sets (*cf.* the Hoare powerdomain). We return to this when we consider related approaches to analysing lists. For a recent treatment of these powerdomains and their logical charqcterisations, see [13].

5 Sum types

Sum types are essential for building recursive data structures; this section shows how they are accommodated in the analysis from Section 2. An element of sum type $\sigma_1 + \sigma_2$ is an element from either σ_1 or σ_2 with a tag indicating which summand the element belongs to. In general, we shall consider arbitrary, finite sums of types, written $\sum_{i=1}^{n} \sigma_i$. Data objects of sum type $\sum_{i \in I} \sigma_i$ are created using the constructors in_i where $i \in I$ and taken apart using the case construct. We shall here only give a restricted version of the case expression where we assume that the σ_i are of product type. The symbol $\overline{x^{\sigma}}$ then stands for a vector of type-annotated variables, one for each factor in the product. These x^{σ}'s are used to access the components of the product. The typing rules for in_j and case are

$$\frac{e : \sigma_i}{\mathsf{in}_i(e) : \sum_{i \in I} \sigma_i} \qquad \frac{e : \sum_{i \in I} \sigma_i \qquad e_i : \rho}{\mathsf{case}\ e\ \mathsf{of}\ \{in_i(\overline{x_i}^{\sigma_i})\ .\ e_i\} : \rho}$$

A case expression with two branches will be written $\mathsf{case}\ e\ \mathsf{of}\ in_1(\overline{x})\ .\ e_1\ []\ in_2(\overline{y})\ .\ e_2$.

We now present an extension of the traditional abstract interpretation of sums [8, 23]. The extension will enable us to describe not only single sum elements of the form $\mathsf{in}_j(d)$ but also collections of such elements The basic formulae of a sum type $\sum_{i \in I} \sigma_i$ will be of the form $\sum_{i \in J} \mathsf{in}_i(\varphi_i)$ where $J \subseteq I$ and φ_i is a property of type σ_i. General properties are con- and disjunctions of these basic properties. An element $\mathsf{in}_j(d)$ satisfies the property $\sum_{i \in J} \mathsf{in}_i(\varphi_i)$ if $j \in J$ and d satisfies φ_j, thus, $\sum_{i \in J} \mathsf{in}_i(\varphi_i)$ lists the summands the element can belong to and the property it satisfies if it belongs to a particular summand. A collection of elements satisfies a sum formula if each element satisfies it. The index set J limits the summands the elements can belong to: for example, the property $\mathsf{in}_k(\varphi)$ is only satisfied by elements from summand k satisfying φ. In general the smaller the index set the more precise is the information contained in the formula, hence the rule $\sum - \leq$ in the axiomatisation of sum properties in Figure 3.

We emphasise that the $\sum$ connective is different from disjunction. In general we have

$$\bigvee_{j \in J} \mathsf{in}_j(\varphi_j) \leq \sum_{j \in J} \mathsf{in}_j(\varphi_j)$$

but not the converse. The disjunction is only satisfied by those collections where all elements come from the same summand. The need for the weaker $\sum$ connective will become apparent when we consider recursive types, where we will need to state that a structure

$$\textstyle\sum - \wedge \quad \bullet \quad \sum_{j\in J_1} \mathsf{in}_j(\varphi_j) \wedge \sum_{j\in J_2} \mathsf{in}_j(\psi_j) = \sum_{j\in J_1\cap J_2} \mathsf{in}_j(\varphi_j \wedge \psi_j)$$

$$\textstyle\sum - \mathtt{f} \quad \bullet \quad \sum_{j\in\emptyset} \mathsf{in}_j(\varphi_j) = \mathtt{f} \qquad \sum - \mathtt{t} \quad \bullet \quad \sum_{j\in I} \mathsf{in}_j(\mathtt{t}) = \mathtt{t}$$

$$\textstyle\sum - \vee \quad \bullet \quad \sum_{j\in J} \mathsf{in}_j(\varphi_j) \vee \sum_{j\in J} \mathsf{in}_j(\psi_j) = \sum_{j\in J} \mathsf{in}_j(\varphi_i \vee \psi_j)$$

$$\textstyle\sum - \leq \quad \bullet \quad \dfrac{\varphi_j \leq \psi_j \qquad \forall j \in J_1 \subseteq J_2 \subseteq I}{\sum_{j\in J_1} \mathsf{in}_j(\varphi_j) \leq \sum_{j\in J_2} \mathsf{in}_j(\psi_j)}$$

Figure 3: Axiomatisation of sum properties

$$(\mathsf{in}) \quad \frac{\Gamma \vdash e : \varphi}{\Gamma \vdash \mathsf{in}_j(e) : \mathsf{in}_j(\varphi)} \qquad (\mathsf{case}\ - \mathtt{f}) \quad \frac{\Gamma \vdash e : \mathtt{f}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{in_i(\overline{x_i})\ .\ e_i\} : \mathtt{f}}$$

$$(\mathsf{case}\ - \vee) \quad \frac{\Gamma \vdash e : \bigvee_{k\in K} \sum_{i\in J_k} \mathsf{in}_i(\varphi_{ik}) \quad \Gamma[\overline{x_i} : \varphi_{ik}] \vdash e_i : \psi_{ik} \quad J_k \neq \emptyset}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{in_i(\overline{x_i})\ .\ e_i\} : \bigvee_{k\in K} \bigvee_{i\in J_k} \psi_{ik}}$$

Figure 4: Axioms and rules for sum type expressions

contains elements from some, but not all, of the summands in a sum.

The rules for reasoning about sum types, shown in Figure 4, follow the typing rules. We let Γ denote an environment of formulae that associates a formula to each free variable. Recall that we only consider the case where each of the summands is a product type. For $\overline{x}$ a vector of variables and φ a formula of the corresponding product type we write $\overline{x} : \varphi$ for the environment obtained by matching each variable in $\overline{x}$ with the corresponding component of φ. In the rule for case , the formula $\sum_{i\in J} \mathsf{in}_i(\varphi_i)$ tells us that one of the branches with index $i \in J$ will be chosen. So we analyse each branch and take the disjunction of the results obtained.

To construct a lattice of sum properties we reason as follows: we can write a sum of properties, $\sum_{i\in J} \mathsf{in}_i(\varphi_i)$, of type $\sum_{i\in I} \sigma_i$ as an I–tuple of properties if we add an extra symbol that expresses *"no property of this summand in the sum"*. As already argued, a formula stating that the value is definitely not in summand j, say, logically implies a formula saying that *if* the value is from summand j then it satisfies some property φ. This means that the new symbol is least in the implication ordering. If we are given a lattice modelling the properties of each summand σ_i we can add this extra symbol by lifting the lattice, adding a new bottom element $\perp$. The abstract lattice of a sum type then becomes (*cf. Section 4*):

$$[\![\textstyle\sum_{i\in I} \sigma_i]\!] = \mathcal{LA}(\sum_{i\in I} \sigma_i) = \mathsf{D}(\prod_{i\in I}(\mathcal{J}(\sigma_i))_\perp)$$

where $\prod$ is the cartesian product of lattices. We introduce a way of writing a tuple where some components are described explicitly and the rest are defined to be $\perp$. For a cartesian product of lattices $\prod_{i=1,\dots,n}(D_i)_\perp$ and elements $\{d_j\}_{j\in J}$ with $J \subseteq \{1,\dots,n\}$ we define

$$\textstyle\prod_{j\in J} d_j \equiv (e_1,\dots,e_n) \quad \text{where } e_i = \begin{cases} d_i & \text{if } i \in J \\ \perp & \text{othw.} \end{cases}$$

In particular, $\prod_{j\in\emptyset} d_j$ is the tuple $(\perp,\dots,\perp)$.

6 Algebraic types

In this section we synthesise the developments from the previous sections into a general definition of properties of algebraic data types. The definition will be completely in terms of properties of sums and multisets so there will be no need for a new axiomatisation. First we define the syntax of the data types with which we shall be concerned. These types are often referred to as *algebraic* types, or "sum–of–products" types and have the format

$$T = \mu\alpha.\sigma_1 \times \alpha^{n_1} + \ldots + \sigma_k \times \alpha^{n_k} = \mu\alpha.\sum_i \sigma_i \times \alpha^{n_i}$$

$$\mathsf{fold}\ : \mathsf{in}_i(\varphi \times \textstyle\sum_{j\in J} \mathsf{in}_j(\chi_j)) \rightarrow \mathsf{in}_i(\Box\varphi \wedge \Diamond\varphi) + \textstyle\sum_{j\in J} \mathsf{in}_j(\chi_j) \qquad \text{if } i \notin J$$

$$\begin{aligned}\mathsf{fold}\ :\ & \mathsf{in}_i(\varphi \times \mathsf{in}_i(\Box \bigvee_{k\in K_i} \varphi_{ik} \wedge \bigwedge_{k\in K_i} \Diamond\varphi_{ik}) + \sum_{j\in J} \mathsf{in}_j(\chi_j)) \\ \rightarrow\ & \mathsf{in}_i(\Box(\varphi \vee \bigvee_{k\in K_i} \varphi_{ik}) \wedge \Diamond\varphi \wedge \bigwedge_{k\in K_i} \Diamond\varphi_{ik}) + \sum_{j\in J} \mathsf{in}_j(\chi_j)\end{aligned}$$

$$\frac{\mathsf{fold}\ : \psi_i \rightarrow \sum_{k\in K_i} \chi_k \qquad (i \in I)}{\mathsf{fold}\ : \sum_{i\in I} \psi_i \rightarrow \bigvee_{i\in I} \sum_{k\in K_i} \chi_k} \qquad \mathsf{unfold} : \textstyle\sum_{j\in\emptyset} \mathsf{in}_j(\chi_j) \rightarrow \mathtt{f} = \mathtt{f} \rightarrow \mathtt{f}$$

$$\begin{aligned}\mathsf{unfold}\ :\ & \sum_{j\in J} \mathsf{in}_j(\Box \bigvee_{i\in I_j} \varphi_{ij} \wedge \bigwedge_{i\in I_j} \Diamond\varphi_{ij}) \quad \rightarrow \qquad\qquad (j \neq \emptyset) \\ & \bigvee_{j\in J} \bigvee_{\emptyset\neq I\subseteq I_j} \mathsf{in}_j(\bigwedge_{i\in I} \varphi_{ij} \times \mathsf{in}_j(\Box \bigvee_{i\in I_j} \varphi_{ij} \wedge \bigwedge_{i\in I_j\setminus I} \Diamond\varphi_{ij}) + \sum_{J\setminus\{j\}} \mathsf{in}_j(\Box \bigvee_{i\in I_j} \varphi_{ij} \wedge \bigwedge_{i\in I_j} \Diamond\varphi_{ij}))\end{aligned}$$

Figure 5: Rules for fold and unfold

where σ_i is a simple type and n_i is an integer value, possibly zero. Binary trees with boolean values at the leaves have type $\mu\alpha.\mathtt{Bool} + \mathbf{1} \times \alpha \times \alpha$. and the type of lists of integers is defined by $\mu\alpha.\mathtt{Int} \times \alpha + \mathbf{1}$. The operations for constructing and decomposing recursive data structures are called fold and unfold. The typing rules for fold and unfold are

$$\frac{e : F(\mu\alpha.F(\alpha))}{\mathsf{fold}\ e : \mu\alpha.F(\alpha)} \qquad \frac{e : \mu\alpha.F(\alpha)}{\mathsf{unfold}\ e : F(\mu\alpha.F(\alpha))}.$$

Assume that e is of type lists of integers. If e evaluates to the list $[2,3]$ then the expression fold $\mathsf{in}_1(1,e)$ evaluates to the list $[1,2,3]$. Unfolding that value would yield the tagged pair $\mathsf{in}_1(1,[2,3])$.

Recall that we are interested in those properties that only depend on the content, and not on the structure, of the data structure hence we consider a recursive structure as a multiset of elements and reason about it as we would reason about multisets. We can use the type structure to refine this description so that a structure is viewed as a collection of multisets, one for each type the elements can have. For example, the elements in the type lists of integers given by $\mu\alpha.\mathtt{Int} \times \alpha + \mathbf{1}$ will be described by two multisets: the multiset of integers in the list and the multiset of values terminating the list. The latter will either be the one-point set, in case the list is finite, or the empty set, in case the list is partial or infinite.

We can use the formal system of uniform properties as a basis for a logic of recursive data structures but we have to deal with the empty set explicitly since this is not handled by the uniform properties. With the logic for sum we can express that there is definitely no value in a summand, so by organising our collection of sets as a sum we can express that the set of elements of a given type is empty by stating that there is no set for this type in the sum. In other words, the logic for sums allows us to express that the elements in the structure are drawn from only some of the summands from the type expression. This was the motivation for developing the more refined logic of sums and provides the intuition behind the following definition of formulae of recursive type. Define the set $L(\mu\alpha.\sum_i \sigma_i \times \alpha^{n_i})$ of formulae denoting properties of recursive type as follows:

Definition 6.1 (Formulae for recursive types)

$$L(\mu\alpha.\sum_i \sigma_i \times \alpha^{n_i}) = L(\sum_i \mathcal{P}(\sigma_i)).$$

Some examples from strictness analysis over the list of integers type $\mu\alpha.\mathtt{Int} \times \alpha + \mathbf{1}$: assuming that we have two properties for integer values, $\mathtt{f}_{\mathtt{Int}}$ meaning that the value is undefined and $\mathtt{t}_{\mathtt{Int}}$ meaning that the value might be defined, we can form properties like the following for lists of integers.

- $\mathtt{f}_{\mathcal{P}(\mathtt{Int})+\mathcal{P}(\mathbf{1})}$ This property states that there is no content from either summand. It is only satisfied by the undefined list $\perp_{\mu\alpha.\mathtt{Int}\times\alpha+\mathbf{1}}$.
- $\mathsf{in}_1(\Diamond\mathtt{f}_{\mathtt{Int}})$ This describes all the lists that contain an undefined element *and* whose ending is undefined.
- $\mathsf{in}_1(\Box\mathtt{f}_{\mathtt{Int}}) + \mathsf{in}_2(\Box\mathtt{t}_{\mathbf{1}})$ This property is satisfied by all finite, infinite or partial lists, all the elements of which are undefined.

- $\mathsf{in}_1(\Box \mathtt{t}_{\mathtt{Int}}) + \mathsf{in}_2(\Box \mathtt{t}_{\mathbf{1}})$ The weakest property that holds for all lists. It says nothing about the content and nothing about the ending.

The rules in Figure 5 define how the operations fold and unfold interact with the properties for recursive structures. The argument to fold is described by a formula consisting of a property of the element to be "folded" into the structure and a property of the structure itself. When adding an element from σ_j satisfying property φ to a data structure we have to modify that part of the formula that describes the set of elements from summand σ_j. Assume this set satisfies the formula

$$\Box \bigvee_{i \in I} \varphi_i \wedge \bigwedge_{i \in I} \Diamond \varphi_i.$$

Inserting an element satisfying φ creates a set that satisfies

$$\Box(\varphi \vee \bigvee_{i \in I} \varphi_i) \wedge \Diamond \varphi \wedge \bigwedge_{i \in I} \Diamond \varphi_i.$$

The fold-operation can only change the set of σ_i-elements, hence the formulae describing the other summands are left unchanged. In the rules we let χ range over those formulae of $\mathcal{L}(\mathcal{P}(\sigma_k))$ that do not change.

The property describing the argument to unfold will be built from properties of the form

$$\sum_{j \in J} \mathsf{in}_j(\Box(\bigvee_{i \in I_j} \varphi_{ij}) \wedge \bigwedge_{i \in I_j} \Diamond \varphi_{ij})$$

where J lists the set of non-empty summands. Unfolding such a structure gives a tagged pair $\mathsf{in}_j(d, e')$ where d belongs to the set of elements described by the j'th summand in the formula above. We want to determine the properties that d and the set of elements in e' satisfy. Since we have no information about the order in which the elements appear in the data structure this corresponds directly to choosing at random an element from a set. The element d will satisfy a subset of the properties, $\{\varphi_{ij}\}_{i \in J \subseteq I_j}$. The remaining elements will still satisfy $\Box(\bigvee_{i \in I_j} \varphi_{ij})$ since we haven't added any elements; however, d could have been the only element satisfying the φ_{ij}'s so we cannot guarantee that there remain elements satisfying these. The remaining structure therefore only satisfies the weaker

$$\Box(\bigvee_{i \in I_j} \varphi_{ij}) \wedge \bigwedge_{i \in I_j \setminus J} \Diamond \varphi_{ij}.$$

We cannot predict exactly which properties d will satisfy so we have to consider all possible subsets of I_j; hence the disjunction in the rule for unfold.

We construct abstract lattices for algebraic types using the lattices modelling sums and multisets. By definition we have that $[\![\mu\alpha. \sum_i \sigma_i \times \alpha^{n_i}]\!] = [\![\sum_i \mathcal{P}(\sigma_i)]\!]$ so by modelling the last sum by a product we can extend the lower-set interpretation $[\![\ ]\!]$ of types to algebraic types. When lifting a lattice of convex sets we shall use the empty set $\emptyset$ as the bottom element so we extend the ordering $\sqsubseteq_{EM}$ by stipulating that $\emptyset \sqsubseteq_{EM} C$ for all convex sets C. We could have worked with an abstract bottom element for $P_C(\mathcal{J}(\sigma_i))_\perp$ but introducing the concrete set $\emptyset$ enables us to use some of the usual set operations as operations on the lattices. Furthermore $\emptyset$ is a natural choice of symbol to represent that there is no element from that summand in a structure.

Definition 6.2 (Lattices for algebraic data types)

$$[\![\mu\alpha. \textstyle\sum_i \sigma_i \times \alpha^{n_i}]\!] = \mathsf{D}(\prod_{i \in I} P_C(\mathcal{J}(\sigma_i))_\emptyset).$$

So we model a data type $\mu\alpha. \sum_i D_i \times \alpha^{n_i}$ by an I-tuple $(P_1, \ldots, P_n)$ where P_i is either a convex set describing the multiset of elements of type σ_i or is the empty set $\emptyset$ meaning that there are no elements from the i-th summand.

7 Abstract lattices of list structures

The definition of abstract lattices specialised to a σ-list, $\mu\alpha.\sigma \times \alpha + \mathbf{1}$ yields the following abstract lattice:

$$[\![\mu\alpha.\sigma \times \alpha + \mathbf{1}]\!] = \mathsf{D}(P_C(\mathcal{J}(\sigma))_\emptyset \times P_C(\mathcal{J}(\mathbf{1})))$$

where $\mathcal{J}(\sigma)$ is the partial order modelling the implication order between the disjunction-free formulae of type σ. Thus the lattice will be constructed as the lower sets of a partial order whose elements are pairs of convex or empty sets (C, E). The set C describes the σ elements in the lists and E describes the set of elements from the summand $\mathbf{1}$. There is only one property of the one-element type $\mathbf{1}$, the trivial $\mathtt{t}_{\mathbf{1}}$, so $\mathcal{J}(\mathbf{1})$ is the one-point partial order $\{\bullet\}$. The only non-empty convex subset of $\mathcal{J}(1)$ is $\{\bullet\}$, hence there are only two possibilities for E: either E is $\emptyset$ or E is $\{\bullet\}$ itself. $E = \emptyset$ means that there are no elements from $\mathbf{1}$ *i.e.*, the list is not well terminated meaning that it is either infinite or partial. On the other hand, $E = \{\bullet\}$ means that *if* there are elements from $\mathbf{1}$ *then* they satisfy the trivial property *i.e.*, $E = \{\bullet\}$ means that the list can be anything of finite, infinite or partial. Thus we see that information about the content allows us to derive important structural information about the list.

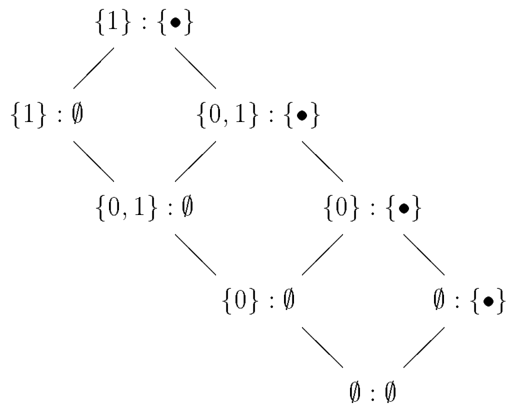

Figure 6: The partial order $(P_C(\mathbf{2}))_\emptyset \times (P_C(\{\bullet\}))_\emptyset$

As an example of a lattice we shall look at the lattice describing the type for integer lists defined by $\mathsf{Int\ List} \equiv \mu\alpha.\mathsf{Int}{\times}\alpha + \mathbf{1}$. The lattice modelling integer lists is built up from the lattices modelling sets of integers and sets of elements from the unit type 1. Sets of integers are modelled by $P_C(\mathbf{2})$ and were considered in Section 4.1. The lattice modelling integer lists is

$$[\![\mathsf{Int\ List}]\!] = \mathsf{D}((P_C(\mathbf{2}))_\emptyset \times (P_C(\{\bullet\}))_\emptyset).$$

We have depicted $(P_C(\mathbf{2}))_\emptyset \times (P_C(\{\bullet\}))_\emptyset$ in Figure 6. We write the pair (C, E) as $C : E$ akin to the way lists are usually written. For example, from the pair $\{1\} : \emptyset$ we read that the set of integers in the list satisfies the property $\Box\mathtt{t} \wedge \Diamond\mathtt{t}$ and there are no elements from the $\mathbf{1}$ summand; a property describing all partial or infinite lists. As a more complicated example we have an element like

$$\downarrow\{\{1\} : \emptyset, \{0,1\} : \{\bullet\}\}$$

representing the disjunctive property that either the list is infinite or partial or it contains an undefined element. This is the property corresponding to the property $\bot\in$ in the four-point domain of Wadler.

In his original work on strictness analysis of lists [25] Wadler suggested the following four-point domain for the analysis of programs over lists of integers. With each point in the domain we list the set of lists that the point represents. [1]

[1] Wadler writes $\top\in$ and $\bot\in$ where we write $1\in$ and $0\in$.

$1\in$	all lists
$0\in$	lists containing an undefined element and all partial or infinite lists
∞	all partial or infinite lists
$\bot$	the undefined list

As mentioned, Wadler's $0\in$ is in our lattice the point $\downarrow\{\{1\} : \emptyset, \{0,1\} : \{\bullet\}\}$. The $\bot$ property in the four-point domain corresponds to $\downarrow\{\emptyset : \emptyset\}$, the $1\in$ property to $\downarrow\{\{1\} : \{\bullet\}\}$ and finally the property ∞ corresponds to $\downarrow\{\{1\} : \emptyset\}$. The abstract interpretation of the list operations are defined over this domain such that *e.g.*, we have $\mathsf{cons}^\#\ 0\ \ 1\in\ =\ \ 0\in$ and $\mathsf{tail}^\#\ \infty\ =\ \infty$.

Nielson and Nielson [24] have suggested a generalisation of Wadler's lattices for lists. Their lattice for σ-lists contains, in addition to the two elements $\bot$ and ∞, an element $Y\in$ for each upwards closed subset Y of the abstract lattice $[\![\sigma]\!]$. Such an element represents the property *"either the list is partial or infinite, or there exists for each $y \in Y$ an element of the list satisfying the property y"*. This property is expressible in our system of uniform properties. For each $y \in Y$, let φ_y be a formula such that $[\![\varphi_y]\!] = y$ *i.e.*, φ_y denotes the property y. Then the formula

$$\mathsf{in}_1(\mathtt{t}) \vee (\mathsf{in}_1(\bigwedge_{y\in Y} \Diamond\varphi_y) + \mathsf{in}_2(\mathtt{t})) \qquad (*)$$

corresponds to the property $Y\in$. More precisely, by restricting the set of formulae to those of the form $(*)$ we obtain a set of properties isomorphic to that of Nielson and Nielson. To see this we note that the normal form of the formula $\bigwedge_{y\in Y} \Diamond\varphi_y$ is

$$\Box(\bigvee_{y\in Y} \varphi_y \vee \mathtt{t}) \wedge \bigwedge_{y\in Y} \Diamond\varphi_y \wedge \Diamond\mathtt{t}$$

which is modelled by the convex closure of $Y \cup \{[\![\mathtt{t}]\!]\}$ (*cf.* Section 4.1). This is exactly the construction from [24] because a convex set is upwards closed exactly when it contains the top element $[\![\mathtt{t}]\!]$[2]. Thus our lattice contains the elements from that of Nielson and Nielson. Our additional elements come from the $\Box$-properties that allow us to talk about properties of all the elements in the lists; properties not considered in [24].

[2] This argument assumes that there *is* a top element in the partial order which is always the case here (since for all types we have the trivial property $[\![\mathtt{t}]\!]$)

A drawback of our construction is that it can result in lattices where different elements denote the same property in the concrete semantics. In [11] it is pointed out that in strictness analysis, where properties are taken to be Scott-closed sets, the two elements $\downarrow\{\{1\}:\emptyset\}$ and $\downarrow\{\{0,1\}:\emptyset\}$ represent the same Scott-closed set of lists. Although a drawback of the method, this redundancy does not in itself jeopardise the correctness of the analysis, and a lattice without duplicate elements can be obtained by the method of [8].

8 List operations

In this section we show how our lattice for analysing lists can be used in the abstract interpretation of list operations. We consider the operations

$$\begin{array}{lll} \mathsf{cons} & : & \lambda p.\mathsf{fold}\ \mathsf{in}_1(p) \\ \mathsf{head} & : & \lambda l.\mathsf{case}\ \mathsf{unfold}\ l\ \mathsf{of}\ \mathsf{in}_1(x,xs)\ .\ x\ []\ \mathsf{in}_2(y)\ .\ \bot \\ \mathsf{tail} & : & \lambda l.\mathsf{case}\ \mathsf{unfold}\ l\ \mathsf{of}\ \mathsf{in}_1(x,xs)\ .\ xs\ []\ \mathsf{in}_2(y)\ .\ \bot \end{array}$$

where we use undefined ($\bot$) to model the occurrence of an error.

It is possible to calculate the abstract interpretations of these operations from those of case, in, unfold and fold and obtain an abstract interpretation that is sound and complete with respect to the logic defined by the inference rules for these operators (*cf.* Section 4,). We do not have space to do this (see [18] for details); instead, we shall just define the abstract interpretations and show part of their tabulations. When defining a function f over a lattice which consists of downwards closed sets of partial orders, we define it on downwards closures of single elements, $\downarrow\{p\}$, and extend it to arbitrary downwards closed sets by setting

$$f(\downarrow\{p_1,\ldots,p_n\} = f(\downarrow\{p_1\})\cup\ldots\cup f(\downarrow\{p_n\})$$

The abstract functions are then defined by

$$\begin{aligned} [\![\mathsf{cons}]\!]\downarrow\{(c,(C,E))\} &= \downarrow\{(\overline{C\cup\{c\}},E)\} \\ [\![\mathsf{head}]\!]\downarrow\{(C,E)\} &= \{0_{\mathsf{Int}}\}\cup\bigcup_{c\in C}\downarrow\{c\} \\ [\![\mathsf{tail}]\!]\downarrow\{(C,E)\} &= \{(\emptyset,\emptyset)\}\cup\bigcup_{c\in C}\{(\max(C,C\setminus\{c\}),E)\} \end{aligned}$$

The following table gives the value of the abstract functions at the arguments corresponding to the elements of four-point domain. The abstract values of the list argument are listed vertically and the abstract values of the element being added to the list are listed horizontally. This is exactly the table for head and tail from Wadler's article [25].

l	$[\![\mathsf{head}]\!]$	$[\![\mathsf{tail}]\!]$
$\downarrow\{\{1\}:\{\bullet\}\}$	$\downarrow\{1\}$	$\downarrow\{\{1\}:\{\bullet\}\}$
$\downarrow\{\{1\}:\emptyset,\{0,1\}:\{\bullet\}\}$	$\downarrow\{1\}$	$\downarrow\{\{1\}:\{\bullet\}\}$
$\downarrow\{\{1\}:\emptyset\}$	$\downarrow\{1\}$	$\downarrow\{\{1\}:\emptyset\}$
$\downarrow\{\emptyset:\emptyset\}$	$\downarrow\{0\}$	$\downarrow\{\emptyset:\emptyset\}$

The next table shows what happens when adding an undefined element to an integer list described by the elements from the four-point domain.

$[\![\mathsf{cons}]\!]$	$\downarrow\{0\}$
$\downarrow\{\{1\}:\{\bullet\}\}$	$\downarrow\{\{1\}:\emptyset,\{0,1\}:\{\bullet\}\}$
$\downarrow\{\{1\}:\emptyset,\{0,1\}:\{\bullet\}\}$	$\downarrow\{\{1\}:\emptyset,\{0,1\}:\{\bullet\}\}$
$\downarrow\{\{1\}:\emptyset\}$	$\downarrow\{\{1\}:\emptyset\}$
$\downarrow\{\emptyset:\emptyset\}$	$\downarrow\{\{0\}:\emptyset\}$

This is the table reported by Wadler [25] except for the bottom row. When adding an undefined element to an undefined list, Wadler can only say that the ending is undefined, *i.e.*, that the result is ∞. With our $\Box$-properties we are able to express the additional fact that all elements of the resulting list are undefined.

9 Conclusion

This paper has provided a framework for defining analyses of algebraic data structures. The analyses were developed from a simple axiomatic system of uniform properties based on familiar axioms from modal logic. This extends existing non-standard type systems for program analysis to treat algebraic data structures. From the axiomatisation we then derived lattices of properties over which abstract interpretation of programs over algebraic types can be formulated. The lattices encompasses some of the earlier structures proposed in the literature and can therefore lead to more precise analysers. This was demonstrated by analysing some of the standard list operations.

Other work has dealt with relevant, but non-uniform, properties of lists have been considered, including Wadler and Hughes' head-strictness [26] and Burn's H_B-strictness [6]. Interesting methodologies for constructing abstract domains have been proposed by Hughes and Ferguson [12] and Benton [4] but lack of space prevents us from a deeper comparison.

There are several possibilities for extending the work. The logic contains a large number of properties even for integer lists. Subfragments should be studied, *e.g.*, a logic only involving $\Box$-properties and conjunction which could be combined with the conjunctive systems considered in [3, 17]. This should allow a type system formulation of Launchbury's uniform binding-time analysis [21] that only considers properties pertaining to all elements of a structure thus integrating this analysis into a higher-order framework.

The relation to the standard semantics should be defined rigorously through a concretisation function from the lattice of abstract properties to a suitable set of properties of the concrete semantics. A theory for constructing these concretisation functions systematically from the definition of the algebraic type given the concretisation functions for the underlying types would be a worthwhile goal to pursue. One approach is reported by Benton [4].

The problem of redundant elements in the lattices deserves a closer examination. In particular it should be investigated whether treating terminating elements of a structure differently from elements with a recursive component could yield a more accurate construction

The size of the lattices constructed by our method makes a conventional implementation based on fixpoint iteration infeasible. It would therefore be desirable to extend the promising work by Hankin and Le Metayer [14] on implementing analyses formulated as type systems to cover the analyses proposed here.

Acknowledgements. Thanks to Chris Hankin and the anonymous referees for comments on the paper. Diagrams were drawn using Paul Taylor's macros.

References

[1] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51, 1991.

[2] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[3] P. N. Benton. Strictness logic and polymorphic invariance. In A. Nerode and M. Taitslin, editors, *Proc. of the 2. International Symposium on Logical Foundations of Computer Science, Tver, Russia*, LNCS vol. 620. Springer, 1992.

[4] P. N. Benton. Strictness properties of lazy algebraic datatypes. In P. Cousot et. al., editor, *Static Analysis.* Springer, LNCS vol. 724, 1993.

[5] P.N. Benton. *Strictness Analysis and Optimising Transformations for Lazy Functional Programs.* PhD thesis, University of Cambridge, 1992.

[6] G.L. Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing. Pitman, 1991.

[7] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. *Science of Computer Programming*, 7:249–278, 1986.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM Symposium on Principles of Programming Languages*, 1979.

[9] P. Cousot and R. Cousot. Abstract interpretation frameworks. Technical Report LIX/RR/92/10, Ecole Polytechnique, 1992.

[10] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[11] C. Ernoult and A. Mycroft. Uniform ideals and strictness analysis. In *Proc. of 18th Int. Colloq. on Automata, Languages and Programming*, LNCS vol. 510. Springer, 1991.

[12] A.B. Ferguson and J. Hughes. An iterative powerdomain construction. In K. Davis and J. Hughes, editors, *Functional Programming. Proc. of the 1989 Glasgow workshop.* Springer Verlag, Workshops in Computing Series, 1989.

[13] C. Gunter and D. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. North-Holland, 1990.

[14] C. Hankin and D. Le Metayer. Deriving algorithms from type inference systems: Applications to strictness analysis. In *Proc. of 21st Conf. on Principles of Programming Languages.* ACM Press, 1994. To appear.

[15] L.S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* PhD thesis, Imperial College, Univ. of London, 1991.

[16] S. Hunt and D. Sands. Binding time analysis: A new PERspective. In *Proc. ACM Symposium on Partial Evaluation and Semantics–Based Program Manipulation*, 1991.

[17] T. P. Jensen. Strictness analysis in logical form. In J. Hughes, editor, *Proc. of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS vol. 523. Springer Verlag, 1991.

[18] T. P. Jensen. *Abstract Interpretation in Logical Form.* PhD thesis, Imperial College, Univ. of London, November 1992.

[19] T. P. Jensen. Axiomatising uniform properties of recursive data structures. In M. Billaud et. al., editor, *Proc. of 2nd Bordeaux Workshop on Static Analysis*, Bigre no. 81—82, 1992.

[20] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Series in Computer Science. Prentice-Hall International, 1993.

[21] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, Department of Computing, University of Glasgow, 1989.

[22] A. Leung and P. Mishra. Reasoning about simple and exhaustive demand in higher–order lazy languages. In J. Hughes, editor, *Proc. of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS vol. 523. Springer Verlag, 1991.

[23] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69:117–242, 1989.

[24] H. R. Nielson and F. Nielson. The tensor product in Wadler's analysis of lists. In B. Krieg–Brückner, editor, *Proc. of 4th European Symposium on Programming*, LNCS vol. 582. Springer Verlag, 1992.

[25] P. Wadler. Strictness analysis in non–flat domains (by abstract interpretation over finite domains). In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages.* Ellis Horwood, 1987.

[26] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Conf. on Functional Programming and Computer Architecture*, LNCS vol. 274. Springer, 1987.

[27] G. Winskel. Powerdomains and modality. In M. Karpinski, editor, *Foundations of Computation Theory*, LNCS vol. 158. Springer, 1983.