Disjunctive Program Analysis for Algebraic Data Types

THOMAS JENSEN
IRISA/CNRS

We describe how binding-time, data-flow, and strictness analyses for languages with higher-order functions and algebraic data types can be obtained by instantiating a generic program logic and axiomatization of the properties analyzed for. A distinctive feature of the analyses is that disjunctions of program properties are represented exactly. This yields analyses of high precision and provides a logical characterization of abstract interpretations involving tensor products and uniform properties of recursive data structures. An effective method for proving properties of a program based on fixed-point iteration is obtained by grouping logically equivalent formulae of the same type into equivalence classes, obtaining a lattice of properties of that type, and then defining an abstract interpretation over these lattices. We demonstrate this in the case of strictness analysis by proving that the strictness abstract interpretation of a program is the equivalence class containing the strongest property provable of the program in the strictness logic.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Program Verification—*assertion checkers*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*; *procedures, functions and subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; E.1 [**Data**]: Data Structures—*lists*; *trees*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions; invariants; logics of programs; mechanical verification; pre- and postconditions*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Abstract interpretation, binding-time and data-flow analysis, lambda calculus, nonstandard type systems, strictness analysis

1. INTRODUCTION

A fundamental issue in the design of data-flow analyses of programs is how to combine several pieces of information that can reach a particular program point along different execution paths. As a simple example consider a program with two integer variables s, t where two different paths lead to the same point p. Along one path we can establish that variable s will be 0 when execution reaches p, and along the other that t will be 0 at p. Assuming that we cannot determine which of the paths are taken during execution we cannot say anything about s or t in isolation,

Author's address: IRISA, Campus de Beaulieu, 35042 Rennes, France. Email: jensen@irisa.fr.

so if we insist on combining the two pieces of information $(0, \mathsf{Int})$ and $(\mathsf{Int}, 0)$ into one single pair we obtain the trivial information $(\mathsf{Int}, \mathsf{Int})$. This is clearly weaker than the information that *can* be retained (i.e., that *either* s *or* t will be 0), but modeling this disjunctive information requires a more elaborate description of (s, t), e.g., as a set of pairs $\{(0, \mathsf{Int}), (\mathsf{Int}, 0)\}$. Jones and Muchnick [1981] coined the terms *independent attribute* and *relational* for these two ways of combining information about tuples. Modeling disjunctive properties can yield more precise analyses at the price of more complex domains of program properties: a relational analysis with disjunctive properties would for example be able to predict that multiplying s and t gives 0.

The purpose of this article is to study the use of disjunctions in analysis of programs with higher-order functions and recursive data structures and in particular to develop a program analysis framework for a language with these features where disjunctive properties are modeled exactly. The following example from strictness analysis shows where such properties are of interest. Consider the following function:

$$\mathsf{h\ x} = +(\mathsf{if\ B\ then\ (x, 17)\ else\ (17, x)}),$$

where $+$ denotes the function that accepts as argument a pair of integers and returns their sum. Even without information about which branch of the if will be taken, we can see that if the value of x is undefined, then the value of the if-statement is a pair where either the first or the second component is undefined. Applying $+$ to such a pair gives an undefined result, so h is a strict function. However, in the standard strictness analysis [Burn et al. 1986] or in the equivalent strictness logic with conjunctive properties [Benton 1992b; Jensen 1991] the disjunction describing the pair cannot be expressed, so the only safe description permitted by these analyses is that the if-statement returns a pair where both components may be defined. Applying $+$ to such a pair may give a well-defined result, and so the logic has failed to detect that h is strict. The framework developed here leads to the construction of a strictness analysis where disjunctions can be expressed.

An analysis that only considers tuples is, however, of limited use, and we want to define properties of a larger class of data structures. For structures such as tuples and sums we can include a property of each component in the description of the whole data object without having to introduce infinite descriptions. This is no longer feasible for unbounded data structures such as lists. For these structures we have to identify properties that have a finite description and still carry useful information. To this end we introduce *uniform properties*: a property of a data structure 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 multiset of elements contained in a data structure. In this article we restrict attention to two kinds of properties that we write $\Diamond\varphi$ 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 leads to a generalization of a number of analyses proposed in the literature.

The fundamental part of our program analysis framework consists of an axiomatic description of disjunctive properties, given in Section 2. This description takes the form of a logical inference system defining an entailment relation $\leq$ between formulae denoting properties. The axiomatization is divided into a generic and an analysis-specific part, and we show how to instantiate it to yield data-flow, binding-time, and strictness analyses. As an example of a formula involving function and product type we have

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

With $\mathtt{f}$ representing the property of being undefined and $\mathtt{t}$ signifying "no information," this formula describes the strictness property of function $\mathtt{h}$ from above. The entailment relation is defined via axioms and rules. For the uniform properties we have generic 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 data structure satisfy φ and ψ. then they all satisfy φ and they all satisfy ψ. We also have the axiom

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

so the two formulae are provably equal and therefore denote the same property. Using this axiomatization we define (Section 3) a program logic for reasoning about properties of expressions in our language. Similar to the axiomatization, the logic consists of a generic part that is extended with a collection of analysis-specific inference rules to constitute a complete program analysis. The formulation as a program logic provides a clear description of the analysis, free of implementation issues.

An effective method for determining properties of programs based on fixed-point iteration can then be obtained in two steps:

—For each type we group the formulae into equivalence classes, written $[\![\varphi]\!]$. Two formulae are equivalent if they entail each other in the axiomatization (like the preceding $\Box\varphi \wedge \Box\psi$ and $\Box(\varphi \wedge \psi)$). The resulting structure will be a lattice: the lattice of properties of that type.

—Each language construct is given an *abstract interpretation* as an element of the lattice of properties corresponding to its type.

We can then define the abstract interpretation of a program p compositionally as a function $[\![p]\!]$ over these lattices of properties such that if input satisfying (the property denoted by) φ is mapped to output satisfying ψ then $[\![\psi]\!] \sqsubseteq [\![p]\!]([\![\varphi]\!])$. This establishes the equivalences

$$\begin{array}{rcl} \text{Formulae + axiomatization} & \sim & \text{Lattices of properties} \\ \text{Logic} & \sim & \text{Abstract interpretation.} \end{array}$$

For an example, we define an abstract interpretation for strictness analysis and prove that this abstract interpretation is a sound and complete model of the disjunctive strictness logic from Section 3 in the sense that the abstract interpretation of a program is the strongest property provable in the strictness logic for this program. We emphasize that we are working solely at the level of abstract program properties and that we do *not* develop a machinery for relating the analyses to a standard semantics. This issue has been studied by others and is discussed briefly in Section 6.

Section 4 describes the lattices of properties and defines the abstract interpretation of our programming languages over these lattices. Section 5 examines in more detail the lattices of uniform properties of list structures and gives examples of analyses of operations on lists. Section 6 gives an account of related work. Proofs of the main theorems can be found in Appendices A.1–A.6.

2. AXIOMATIZATION

This section describes a formal system for reasoning about properties of data structures. First we define our set of types by

$$\sigma, \tau = \mathsf{Int} \mid \mathsf{Bool} \mid 1 \mid \sigma \times \tau \mid \sigma \to \tau \mid \sigma + \tau \mid \ \mu\alpha.\sum_{k=1}^{n} \sigma_k \times \alpha^{n_k}.$$

Int and Bool are the usual types of integer and Boolean values, and 1 is the one-point type. Product and function types are constructed with $\times$ and $\to$, respectively. 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 consider arbitrary finite sums of types, written $\sum_{i=1}^{n} \sigma_i$. Recursive data types $\mu\alpha.T$ have been restricted so that T must be a sum of products, possibly containing the recursion variable as one or more factors. For example, the integer list data type is written $\mu\alpha.(\mathsf{Int} \times \alpha) + 1$. We do not consider mutually recursive data types.

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

Formulae

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

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

$$\frac{\varphi \in L(\sigma_i)}{\mathsf{in}_i(\varphi) \in L(\sum_{i \in I} \sigma_i)} \qquad \frac{\varphi \in L(\sum_{i \in I} \sigma_i)}{\Box\varphi, \Diamond\varphi \in L(\ \mu\alpha.\sum_{i \in I} \sigma_i \times \alpha^{n_i})}$$

We sometimes write φ^σ to make explicit that the formula φ belongs to $L(\sigma)$. The formula $\mathtt{f}^\sigma$ denotes different properties according to the actual analysis (*"undefined"* in data-flow and strictness, *"static"* in binding-time analysis). The formula $\mathtt{t}^\sigma$ denotes the trivial property satisfied by all expressions of type σ. 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. A property $\mathsf{in}_i(\varphi)$ describes a sum element with tag i satisfying φ.

Logical rules

$$\bullet\ \varphi \leq \varphi \qquad \bullet\ \frac{\varphi \leq \psi, \psi \leq \chi}{\varphi \leq \chi}$$

$$\bullet\ \varphi \wedge \psi \leq \varphi \qquad \bullet\ \frac{\varphi \leq \psi_1, \varphi \leq \psi_2}{\varphi \leq \psi_1 \wedge \psi_2} \qquad \bullet\ \varphi \wedge \psi \leq \psi$$

$$\bullet\ \varphi \leq \psi \vee \varphi \qquad \bullet\ \frac{\varphi_1 \leq \psi, \varphi_2 \leq \psi}{\varphi_1 \vee \varphi_2 \leq \psi} \qquad \bullet\ \psi \leq \psi \vee \varphi$$

$$\bullet\ \varphi \leq \mathtt{t} \qquad \bullet\ \frac{\varphi \leq \psi, \psi \leq \varphi}{\varphi = \psi} \qquad \bullet\ \frac{\varphi = \psi}{\varphi \leq \psi, \psi \leq \varphi} \qquad \bullet\ \mathtt{f} \leq \varphi$$

$$\bullet\ \varphi \wedge (\psi_1 \vee \psi_2) = (\varphi \wedge \psi_1) \vee (\varphi \wedge \psi_2)$$

Type-specific rules

$$(\vee - \times) \quad \bullet\ (\varphi_1 \times \psi) \vee (\varphi_2 \times \psi) = (\varphi_1 \vee \varphi_2) \times \psi$$

$$(\times - \vee) \quad \bullet\ (\varphi \times \psi_1) \vee (\varphi \times \psi_2) = \varphi \times (\psi_1 \vee \psi_2)$$

$$(\times - \wedge) \quad \bullet\ (\varphi_1 \times \psi_1) \wedge (\varphi_2 \times \psi_2) = (\varphi_1 \wedge \varphi_2) \times (\psi_1 \wedge \psi_2)$$

$$(\times - \mathtt{t}) \quad \bullet\ \mathtt{t}^{\sigma \times \tau} = \mathtt{t}^{\sigma} \times \mathtt{t}^{\tau} \qquad (\times - \leq) \quad \bullet\ \frac{\varphi_1 \leq \varphi_2, \psi_1 \leq \psi_2}{\varphi_1 \times \psi_1 \leq \varphi_2 \times \psi_2}$$

$$(\rightarrow - \wedge) \quad \bullet\ (\varphi \rightarrow \psi_1) \wedge (\varphi \rightarrow \psi_2) = \varphi \rightarrow (\psi_1 \wedge \psi_2)$$

$$(\vee - \rightarrow) \quad \bullet\ (\varphi_1 \rightarrow \psi) \wedge (\varphi_2 \rightarrow \psi) = (\varphi_1 \vee \varphi_2) \rightarrow \psi$$

$$(\rightarrow - \leq) \quad \bullet\ \frac{\varphi_2 \leq \varphi_1, \psi_1 \leq \psi_2}{\varphi_1 \rightarrow \psi_1 \leq \varphi_2 \rightarrow \psi_2} \qquad (\rightarrow - \mathtt{t}) \quad \bullet\ \mathtt{t}^{\sigma \rightarrow \tau} = \mathtt{t}^{\sigma} \rightarrow \mathtt{t}^{\tau}$$

$$(+ - \wedge) \quad \bullet\ \mathsf{in}_i(\varphi) \wedge \mathsf{in}_i(\psi) = \mathsf{in}_i(\varphi \wedge \psi) \qquad (+ - \mathtt{t}) \quad \bullet\ \bigvee_{i \in I} \mathsf{in}_i(\mathtt{t}^{\sigma_i}) = \mathtt{t}^{\Sigma_{i \in I} \sigma_i}$$

$$(+ - \leq) \quad \bullet\ \frac{\varphi \leq \psi}{\mathsf{in}_i(\varphi) \leq \mathsf{in}_i(\psi)} \qquad (+ - \vee) \quad \bullet\ \mathsf{in}_j(\varphi) \vee \mathsf{in}_j(\psi) = \mathsf{in}_j(\varphi \vee \psi)$$

$$(\Box - \wedge) \quad \bullet\ \Box(\varphi \wedge \psi) = \Box\varphi \wedge \Box\psi \qquad (\Box - \vee) \quad \bullet\ \Box(\varphi \vee \psi) \leq \Box\varphi \vee \Diamond\psi$$

$$(\Diamond - \wedge) \quad \bullet\ \Box\varphi \wedge \Diamond\psi \leq \Diamond(\varphi \wedge \psi) \qquad (\Diamond - \vee) \quad \bullet\ \Diamond(\varphi \vee \psi) = \Diamond\varphi \vee \Diamond\psi$$

$$(\Box - \leq) \quad \bullet\ \frac{\varphi \leq \psi}{\Box\varphi \leq \Box\psi} \qquad (\Diamond - \leq) \quad \bullet\ \frac{\varphi \leq \psi}{\Diamond\varphi \leq \Diamond\psi} \qquad (\Box - \mathtt{t}) \quad \bullet\ \Box\mathtt{t} = \mathtt{t}$$

Strictness-specific rules

$$(\times - \mathtt{f}) \quad \bullet\ \mathtt{f}^{\sigma} \times \mathtt{f}^{\tau} = \mathtt{f}^{\sigma \times \tau} \qquad (+ - \mathtt{f}) \quad \bullet\ \frac{i \neq j}{\mathsf{in}_i(\varphi) \wedge \mathsf{in}_j(\psi) = \mathtt{f}}$$

$$(\rightarrow - \mathtt{f}) \quad \bullet\ \mathtt{t}^{\sigma} \rightarrow \mathtt{f}^{\tau} = \mathtt{f}^{\sigma \rightarrow \tau} \qquad (\Diamond - \mathtt{f}) \quad \bullet\ \Diamond\mathtt{f} = \mathtt{f} \qquad (\Box - \mathtt{f}) \quad \bullet\ \Box\mathtt{f} = \mathtt{f}$$

Fig. 1. Axiomatization of strictness properties.

The axiomatization shown in Figure 1 consists of three 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 composite types. Finally, there is a set of rules that are specific to each analysis (here strictness). We emphasise that $\leq$ only relates formulae of the same type. The axiomatization 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 $\leq$. Most of the rules have a straightforward logical reading. For example, saying that a function f 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 that f maps arguments satisfying φ_2 to ψ. This is the content of the axiom

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

We note that for a sum type with two summands we can express the trivial property of this type as a disjunction

$$\mathsf{in}_1(\mathtt{t}^\sigma) \vee \mathsf{in}_2(\mathtt{t}^\tau) = \mathtt{t}^{\sigma+\tau}$$

because it describes neither the tag nor the tagged value. More generally, the trivial property in a sum type with n summands is equivalent to $\mathsf{in}_1(\mathtt{t}) \vee \ldots \vee \mathsf{in}_n(\mathtt{t})$. As mentioned in the introduction the formulae $\Box\varphi$ and $\Diamond\varphi$ are used to represent properties of recursive data structures, but note that for recursive data structures we do not have that the structure of the type is the same as the structure of the formulae. The axiomatization can be understood in terms of elements of data structures. For example, the axiom

$$(\Box - \vee) \quad \Box(\varphi \vee \psi) \leq \Box\varphi \vee \Diamond\psi$$

expresses that if all elements of a structure satisfy either φ or ψ, then either they all satisfy φ, or there is at least one element satisfying ψ. As for all other types, we have $\mathtt{f}$ representing the completely undefined structure of a particular type. The strictness-specific axioms $\Box - \mathtt{f}$ and $\Diamond - \mathtt{f}$ are due to the fact that all elements in a recursive structure carry a tag indicating the summand they come from; hence a structure that contains an element with no tag can only be the completely undefined structure. From the axioms we can deduce corollaries such as

$$\Diamond\mathtt{t} = \mathtt{t} \quad \text{and} \quad \Box\varphi \leq \Diamond\varphi.$$

2.1 Normal Form

In order to state a Normal Form Theorem for the axiomatization we define the *irreducible* formulae to be those formulae that cannot be expressed as a disjunction of two logically stronger formulae. The subset of irreducible formulae of $L(\sigma)$ is denoted by $\mathsf{Irr}(\sigma)$ and is defined via the following predicate Irr on formulae such that if $\mathsf{Irr}(\varphi)$ for formula $\varphi \in L(\sigma)$ then $\varphi \in \mathsf{Irr}(\sigma)$.

Irreducible formulae $(F(\alpha) = \sum_{i=1}^{k} \sigma_i \times \alpha^{n_i})$

$$\bullet\ \ \mathsf{Irr}(\mathtt{f}^\sigma) \quad \bullet\ \ \mathsf{Irr}(\mathtt{t}^{\mathsf{Int}}) \quad \bullet\ \ \mathsf{Irr}(\mathtt{t}^{\mathsf{Bool}}) \quad \bullet\ \ \mathsf{Irr}(\mathtt{t}^{1})$$

$$\bullet\ \ \frac{\mathsf{Irr}(\mathtt{t}^\sigma)}{\mathsf{Irr}(\mathtt{t}^{\sigma\to\tau})} \qquad \bullet\ \ \frac{\mathsf{Irr}(\mathtt{t}^\sigma) \quad \mathsf{Irr}(\mathtt{t}^\tau)}{\mathsf{Irr}(\mathtt{t}^{\sigma\times\tau})}$$

$$\bullet\ \ \frac{\mathsf{Irr}(\varphi) \quad \mathsf{Irr}(\psi)}{\mathsf{Irr}(\varphi \times \psi)} \qquad \bullet\ \ \frac{\mathsf{Irr}(\varphi_i) \quad i \in I}{\mathsf{Irr}(\bigwedge_{i\in I}(\varphi_i \to \psi_i))} \qquad \bullet\ \ \frac{\mathsf{Irr}(\varphi)}{\mathsf{Irr}(\mathsf{in}_i(\varphi))}$$

$$\bullet\ \ \mathsf{Irr}(\mathtt{t}^{\mu\alpha.F(\alpha)}) \qquad \bullet\ \ \frac{\mathsf{Irr}(\varphi_i) \quad i \in I}{\mathsf{Irr}(\Box(\bigvee_{i\in I}\varphi_i) \wedge \bigwedge_{i\in I}(\Diamond\varphi_i))}$$

Not all $\mathtt{t}$ formulae are irreducible as witnessed by the $(+ - \mathtt{t})$ axiom: hence the special axioms and rules for $\mathtt{t}$. Disjunctions can appear in irreducible formulae, but only on the right-hand side of an arrow $\to$ and under a $\Box$. In Corollary 4.5.5 we prove that if $\mathsf{Irr}(\varphi)$ and $\varphi = \varphi_1 \vee \varphi_2$, then $\varphi = \varphi_1$ or $\varphi = \varphi_2$, justifying the word irreducible.

The irreducible formulae describing recursive data structures can intuitively be read as "each element satisfies $\bigvee_{i\in I}\varphi_i$, and for each $i \in I$ there is an element satisfying φ_i." The following equivalences concerning recursive structures are variations of the axioms $\Box - \vee$ and $\Diamond - \wedge$:

$$\Box(\varphi_1 \vee \varphi_2) = \Box\varphi_2 \vee (\Box(\varphi_1 \vee \varphi_2) \wedge \Diamond\varphi_1)$$
$$\Box\varphi_1 \wedge \Diamond\varphi_2 = \Box\varphi_1 \wedge \Diamond(\varphi_1 \wedge \varphi_2).$$

For example, if all elements of a structure satisfy φ_1, and there exists an element satisfying φ_2, then there there exists an element satisfying $\varphi_1 \wedge \varphi_2$. They are used to prove the following lemma.

LEMMA 2.1.1. *A formula of the form*

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

with $\mathsf{Irr}(\varphi_i)$ *and* $\mathsf{Irr}(\psi_j)$ *is equivalent to a disjunction of irreducible formulae.*

PROOF. The proof follows that of Abramsky [1991a, Theorem 4.5]. By repeated application of the first equivalence from above and by using distributivity we can rewrite the formula into an equivalent disjunction of formulae of the form

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

with $I' \subseteq I$ and $J' \subseteq J$ such that to every φ_i there exists a ψ_j satisfying $\psi_j \leq \varphi_i$. Furthermore, by using the second equivalence on each disjunct we can obtain an equivalent formula

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

which also satisfies that to every ψ_j there exists a φ_i satisfying $\psi_j \leq \varphi_i$. This implies that

$$\bigwedge_{j\in J''} \Diamond\psi_j = \bigwedge_{j\in J''} \Diamond\psi_j \wedge \bigwedge_{i\in I''} \Diamond\varphi_i \text{ and } \Box(\bigvee_{i\in I''} \varphi_i) = \Box(\bigvee_{i\in I''} \varphi_i \vee \bigvee_{j\in J''} \psi_j),$$

so the original formula is equivalent to

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

which is a disjunction of irreducibles. $\Box$

The irreducible formulae are almost closed under conjunction—only the formulae in $\mathsf{Irr}(\mu\alpha.\sum_{i\in I}\sigma_i \times \alpha^{n_i})$ do not have this property. The following lemma is what can be said in general.

LEMMA 2.1.2. *For any $\varphi_1, \varphi_2 \in \mathsf{Irr}(\sigma)$ there exists $\{\psi_i\}_{i\in I} \subseteq \mathsf{Irr}(\sigma)$ such that*

$$\varphi_1 \wedge \varphi_2 = \bigvee_{i\in I} \psi_i.$$

PROOF. By induction over σ. Most of the cases are straightforward. For sums, we have that if $i \neq j$, then $\mathsf{in}_i(\varphi) \wedge \mathsf{in}_j(\psi) = \mathtt{f} \in \mathsf{Irr}(\sigma)$. Otherwise, we have

$$\mathsf{in}_i(\varphi_1) \wedge \mathsf{in}_i(\varphi_2) = \mathsf{in}_i(\varphi_1 \wedge \varphi_2) = \mathsf{in}_i(\bigvee_{j\in J} \varphi_j) = \bigvee_{j\in J} \mathsf{in}_i(\varphi_j)$$

and $\mathsf{Irr}(\mathsf{in}_i(\varphi_j))$ when $\mathsf{Irr}(\varphi_i)$. For recursive structures we have

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

and Lemma 2.1.1 applied to this formula gives a disjunction of the desired form. $\Box$

In general we have the following Normal Form Theorem for the axiomatization in Figure 1.

THEOREM 2.1.3 (NORMAL FORM). *For all $\varphi \in L(\sigma)$ there exist $\psi_1, \ldots, \psi_n \in \mathsf{Irr}(\sigma)$ such that*

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

PROOF. See Appendix A.1. $\Box$

2.2 Other Program Properties

We now show how two other analyses—binding-time analysis and a simple data-flow analysis—can be accommodated in the framework. Binding-time analysis [1985; 1993] is used in partial evaluation to determine how much of an expression can be evaluated given that only part of the input is available. The flow analysis that we consider is one of the classical examples; it traces the sign (positive, negative, or zero) of integer values [Sintzoff 1972] and the value of Boolean expressions. Although too simplistic for practical purposes, it illustrates how the more sophisticated interval-based analyses [Cousot and Cousot 1976] can be handled.

In binding-time analysis, an expression whose value is certain to be determined by the available input is classified as "static." If there is not sufficient input available to evaluate an expression it is called dynamic. At the base types Int and Bool we have that $\mathtt{f}$ denotes static, and $\mathtt{t}$ denotes dynamic. An axiom such as $\times - \mathtt{f}$ expresses that a pair is static if the value of each component is completely determined. At higher types, following Hunt and Sands [1991], we let $\mathtt{f}^{\sigma\to\tau}$ represent the constant functions. This is embodied in the axiom

$$\mathtt{f}^{\sigma\to\tau} = \mathtt{t}^{\sigma} \to \mathtt{f}^{\tau}$$

which states that even if the value of the argument to the function cannot be determined, the result can still be calculated, that is, the function is constant.

For sum types $\sum_{i\in I}\sigma_i$, the rule $+ - \mathtt{f}$ means that the property $\mathtt{f}^{\Sigma\sigma_i}$ denotes the undefined value, as in strictness analysis. The property "static" for sum types is the disjunction $\bigvee_{i\in I}\mathsf{in}_i(\varphi_i)$ with φ_i being the "staticness" predicate at type σ_i. It is important to distinguish between the completely dynamic expression $\mathtt{t}^{\Sigma\sigma_i}$ and the expression where we can determine the tag but not the value under the tag. The latter is described by the disjunction

$$\bigvee_{i\in I}\mathsf{in}_i(\mathtt{t}^{\sigma_i}),$$

so the distinction is obtained by removing the axiom $+ - \mathtt{t}$ from the axiomatization. Since $\mathtt{f}^{\Sigma}$ denotes undefinedness we have that the strictness axioms $\Box - \mathtt{f}$ and $\Diamond - \mathtt{f}$ are still valid for binding-time analysis. As for sum types, it is important to distinguish between the completely dynamic structure and the structure where only the elements are dynamic. This is achieved through our previously explained treatment of sum types, since it allows us to distinguish between a set whose sum elements are completely dynamic and a set of sum elements the tags of which can be evaluated. The axiom $\Box\mathtt{t} = \mathtt{t}$ now identifies the completely dynamic structure with the structure where none of the sum tags can be determined. In Section 5.3 we examine the set of binding-time properties of lists that results from this axiomatization.

A simple data-flow analysis for determining the sign of integer values and calculating with exact Boolean values can be obtained by adding the three formulae $\mathtt{pos}$, $\mathtt{neg}$, and $\mathtt{zero}$ to the set of integer formulae $L(\mathsf{Int})$ and the two formulae $\mathtt{true}$ and $\mathtt{false}$ to $L(\mathsf{Bool})$ (these should not be confused with $\mathtt{t}^{\mathsf{Bool}}$ and $\mathtt{f}^{\mathsf{Bool}}$). The axiomatization must be extended accordingly with the axioms

$$\mathtt{t}^{\mathsf{Int}} = \mathtt{pos} \vee \mathtt{neg} \vee \mathtt{zero} \qquad \text{and} \qquad \mathtt{t}^{\mathsf{Bool}} = \mathtt{true} \vee \mathtt{false}.$$

From the general axiomatization follows that $\mathtt{f}^{\mathsf{Int}}$ (which represents "undefined") implies all three properties, and we obtain properties like $\mathtt{pos} \vee \mathtt{neq}$ and $\mathtt{pos} \vee \mathtt{zero}$, expressing nonzero and nonnegative values, respectively. In order to obtain a normal form theorem for the axiomatization we must add each of the new formulae to the list of irreducibles, since none of them can be expressed as a disjunction of others. On the other hand, we note that $\mathtt{t}^{\mathsf{Int}}$ need no longer be irreducible, since it can be expressed as $\mathtt{pos} \vee \mathtt{neg} \vee \mathtt{zero}$; hence if uniqueness of normal forms is required it must be expelled from the list of irreducibles. Similar considerations hold for the Boolean formulae.

From here on a lot of other properties of tuples of integers, sums of Booleans, lists of integers etc. follow automatically from the framework. For example, the property of integer pairs from the introduction stating that at least one element is zero is written $(\mathtt{t} \times \mathtt{zero}) \vee (\mathtt{zero} \times \mathtt{t})$. In Section 5.3, list properties built of these basic properties are studied further.

3. DISJUNCTIVE PROGRAM ANALYSIS

3.1 Language

For our programming language we take an explicitly typed lambda calculus extended with products, sums, and recursive types and equipped with a conditional and a fixed-point operator. The language constructs together with their typing rules are given in the following. We assume that the language contains binary strict operations such as $+ : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$ and $= : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$. Objects of sum type $\sum_{i \in I} \sigma_i$ are created using the constructors in_i, where $i \in I$, and are taken apart using the case construct. A case expression with two branches is written as

$$\mathsf{case}\ e\ \mathsf{of}\ \mathsf{in}_1(x^{\sigma_1})\ .\ e_1\ []\ \mathsf{in}_2(x^{\sigma_2})\ .\ e_2.$$

The operations for constructing and decomposing recursive data structures are called fold and unfold. If e is an expression of type lists of integers that evaluates to the list $[2,3]$, then the expression $\mathsf{fold}\ \mathsf{in}_1(1,e)$ evaluates to the list $[1,2,3]$. Unfolding that value would yield the tagged pair $\mathsf{in}_1(1,[2,3])$.

Typing rules

$$x^\sigma \in \sigma \qquad n \in \mathsf{Int} \qquad \mathsf{true}, \mathsf{false} \in \mathsf{Bool} \qquad \bullet \in 1$$

$$\frac{e_1 \in \sigma \quad e_2 \in \tau}{(e_1, e_2) \in \sigma \times \tau} \qquad \frac{e \in \sigma \times \tau}{\mathsf{fst}(e) \in \sigma} \qquad \frac{e \in \sigma \times \tau}{\mathsf{snd}(e) \in \tau}$$

$$\frac{e \in \tau}{\lambda x^\sigma . e \in \sigma \to \tau} \qquad \frac{e_1 \in \sigma \to \tau \quad e_2 \in \sigma}{e_1 e_2 \in \tau}$$

$$\frac{b \in \mathsf{Bool} \quad e_1 \in \sigma \quad e_2 \in \sigma}{\mathsf{if}\ b\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \in \sigma} \qquad \frac{e \in \sigma \to \sigma}{\mathsf{fix}\ e \in \sigma}$$

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

$$\frac{e \in \sum_{i=1}^n \sigma_i \times (\mu\alpha. \sum_{i=1}^n \sigma_i \times \alpha^{n_i})^{n_i}}{\mathsf{fold}\ e \in \mu\alpha. \sum_{i=1}^n \sigma_i \times \alpha^{n_i}}$$

$$\frac{e \in \mu\alpha. \sum_{i=1}^n \sigma_i \times \alpha^{n_i}}{\mathsf{unfold}\ e \in \sum_{i=1}^n \sigma_i \times (\mu\alpha. \sum_{i=1}^n \sigma_i \times \alpha^{n_i})^{n_i}}$$

3.2 Strictness Logic

The axiomatization of the disjunctive formulae from Section 2 is now used to formulate a logic for reasoning about properties of expressions from the language. As

Generic rules

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

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

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

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

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

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

$$\textbf{If} - \vee\quad \frac{\Gamma \vdash e_1 : \phi_1 \quad \Gamma \vdash e_2 : \phi_2}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \phi_1 \vee \phi_2} \qquad \textbf{In}\quad \frac{\Gamma \vdash e : \varphi}{\Gamma \vdash \mathsf{in}_j(e) : \mathsf{in}_j(\varphi)}$$

$$\textbf{Case} - \vee\quad \frac{\Gamma \vdash e : \bigvee_{i \in I} \mathsf{in}_i(\varphi_i) \qquad \forall i \in I.\, \Gamma[x_i : \varphi_i] \vdash e_i : \psi_i}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{\mathsf{in}_i(x_i)\ .\ e_i\} : \bigvee_{i \in I} \psi_i}$$

$$\textbf{Fold} - \Box\quad \vdash \mathsf{fold} : \mathsf{in}_j(\varphi \times \Box\psi_1 \times \ldots \times \Box\psi_{n_j}) \to \Box(\mathsf{in}_j(\varphi) \vee \psi_1 \vee \ldots \vee \psi_{n_j})$$

$$\textbf{Fold} - \Diamond\quad \vdash \mathsf{fold} : \mathsf{in}_j(\varphi \times \Diamond\psi_1 \times \ldots \times \Diamond\psi_{n_j}) \to \Diamond\mathsf{in}_j(\varphi) \wedge \Diamond\psi_1 \wedge \ldots \wedge \Diamond\psi_{n_j}$$

$$\textbf{Unfold} - \Box\quad \vdash \mathsf{unfold} : \Box \bigvee_{i \in I} \bigvee_{j \in J_i} \mathsf{in}_i(\varphi_{ij}) \to \bigvee_{k \in I} \mathsf{in}_k((\bigvee_{j \in J_k} \varphi_{kj}) \times (\Box \bigvee_{i \in I} \bigvee_{j \in J_i} \mathsf{in}_i(\varphi_{ij}))^{n_i})$$

$\textbf{Unfold} - \Diamond\quad \vdash \mathsf{unfold}$:

$$\Box \bigvee_{i \in I} \bigvee_{j \in J_i} \mathsf{in}_i(\varphi_{ij}) \wedge \bigwedge_{i \in I} \bigwedge_{j \in J_i} \Diamond\, \mathsf{in}_i(\varphi_{ij}) \to \bigvee_{k \in I} \mathsf{in}_k((\bigvee_{j \in J_k} \varphi_{kj}) \times (\bigwedge_{i \in I \setminus \{k\}} \bigwedge_{j \in J_i} \Diamond\, \mathsf{in}_i(\varphi_{ij}))^{n_i})$$

Strictness rules

$$\textbf{Base}\quad \frac{\Gamma \vdash e : (\mathtt{t} \times \mathtt{f}) \vee (\mathtt{f} \times \mathtt{t})}{\Gamma \vdash bop(e) : \mathtt{f}}$$

$$\textbf{If} - \mathtt{f}\quad \frac{\Gamma \vdash b : \mathtt{f}}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathtt{f}} \qquad \textbf{Case} - \mathtt{f}\quad \frac{\Gamma \vdash e : \mathtt{f}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{\mathsf{in}_i(x_i)\ .\ e_i\} : \mathtt{f}}$$

Fig. 2. Disjunctive strictness logic.

with the axiomatization, the logic contains generic rules and rules that deal specifically with strictness. In Section 3.3 we show how to modify these rules to obtain other analyses.

The logic operates with judgments of the form

$$\Gamma \vdash e : \varphi,$$

where Γ is an environment mapping variables to formulae; e is a term of type σ; and φ a formula in $L(\sigma)$. We write $\Gamma \leq \Delta$ to say that $Dom(\Gamma) \supseteq Dom(\Delta)$, and for all $x \in Dom(\Delta)$ we have $\Gamma(x) \leq \Delta(x)$. We use *bop* to range over the binary strict operators. The logic is presented in Figure 2. In the following we comment on some of its features.

In the rule for if we combine the results of analyzing each branch by a disjunction to obtain the strongest property that holds for both branches. This is demonstrated on the example from the introduction.

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

We verify that h satisfies the property $\mathtt{f} \to \mathtt{f}$, i.e., that it is strict. With Γ denoting the environment $[x : \mathtt{f}]$ we get the deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{x} : \mathtt{f}}\,\textbf{Var} \quad \dfrac{}{\Gamma \vdash 17 : \mathtt{t}}\,\textbf{Taut}}{\Gamma \vdash (\mathsf{x}, 17) : \mathtt{f} \times \mathtt{t}}\,\textbf{Pair} \qquad \dfrac{\dfrac{}{\Gamma \vdash \mathsf{x} : \mathtt{f}}\,\textbf{Var} \quad \dfrac{}{\Gamma \vdash 17 : \mathtt{t}}\,\textbf{Taut}}{\Gamma \vdash (17, \mathsf{x}) : \mathtt{t} \times \mathtt{f}}\,\textbf{Pair}}{\dfrac{\Gamma \vdash \mathsf{if\ B\ then\ (x, 17)\ else\ (17, x)} : \mathtt{f} \times \mathtt{t} \vee \mathtt{t} \times \mathtt{f}}{\Gamma \vdash +(\mathsf{if\ B\ then\ (x, 17)\ else\ (17, x)}) : \mathtt{f}}\,\textbf{Base}}\,\textbf{If} - \vee}{\vdash \lambda \mathsf{x}. + (\mathsf{if\ B\ then\ (x, 17)\ else\ (17, x)}) : \mathtt{f} \to \mathtt{f}.}\,\textbf{Abs}$$

The fixed-point operator is axiomatized as a higher-order function. The reason for this is economical; by axiomatizing it as a function we do not have to specify the behavior of fix on more complex formulae, since this is determined by the axioms for functions. For example, it holds generally that

$$\bigwedge_{i \in I} ((\varphi_i \to \varphi_i) \to \varphi_i) \;\leq\; \bigwedge_{i \in I} ((\varphi_i \to \varphi_i) \to (\bigvee_{i \in I} \varphi_i)) \;\leq\; \bigvee_{i \in I} (\varphi_i \to \varphi_i) \to \bigvee_{i \in I} \varphi_i.$$

By repeatedly using the rule **Conj**, fix can be shown to satisfy the formula to the left, so by using **Weak** we get the following derived fact about fix :

$$\vdash \mathsf{fix} \;:\; \bigvee_{i \in I} (\varphi_i \to \varphi_i) \to \bigvee_{i \in I} \varphi_i$$

The formula $\bigvee_{i \in I} (\varphi_i \to \varphi_i)$ represents the knowledge that a function satisfies (at least) one of the properties $\varphi_i \to \varphi_i$. Hence the fixed point will satisfy (at least) one of the properties φ_i.

Disjunctions can occur in the environment describing free variables in the expression. An environment where x is bound to the formula $\varphi \vee \psi$ indicates that x satisfies either φ or ψ for the whole of the evaluation of this expression. When analyzing the components of the expression we must take care to ascribe the same property to x throughout the analysis. This is made possible by the rule **Disj**. As an example consider analyzing the expression (x, x) in an environment where x is bound to a disjunction $L \vee R$ (L and R could, for example, be the properties $\mathtt{f} \times \mathtt{t}$ and $\mathtt{t} \times \mathtt{f}$ if x itself were of product type). Without the **Disj** rule we can only prove

$$[x : L \vee R] \vdash (x, x) : (L \vee R) \times (L \vee R),$$

which has normal form

$$(L \times L) \vee (L \times R) \vee (R \times L) \vee (R \times R).$$

However, by considering each disjunct in turn we can prove

$$\dfrac{\dfrac{}{[x : L] \vdash (x, x) : L \times L}\,\textbf{Pair} \qquad \dfrac{}{[x : R] \vdash (x, x) : R \times R}\,\textbf{Pair}}{[x : L \vee R] \vdash (x, x) : (L \times L) \vee (R \times R)}\,\textbf{Disj}$$

which is a stronger property.

The rule $\textbf{Case} - \mathtt{f}$ is specific to strictness analysis by stating that if the value on which to branch is undefined, then the value of the whole case expression is undefined. In the rule $\textbf{Case} - \vee$, the formula $\bigvee_{i \in J} \mathsf{in}_i(\varphi_i)$ indicates that one of the branches with index $i \in J$ will be chosen. So we analyze each branch, assigning the bound variable the property φ_i, and take the disjunction of the results obtained. Notice that this is the natural generalization of the treatment of the conditional if.

The rules defining how the operations fold and unfold interact with the properties for recursive structures can be explained as follows. The argument to fold is described by a property of the element to be "folded" into the structure and a property for each of the substructures with which it has to be folded. Folding an element satisfying property φ together with a set of data structures means that an element of the resulting structure now also might satisfy $\mathsf{in}_j(\varphi)$ and that there exists an element in the structure satisfying $\mathsf{in}_j(\varphi)$. Assuming that the ith substructure satisfies the formula $\Box\psi_i$ the resulting structure will satisfy

$$\Box(\mathsf{in}_j(\varphi) \vee \bigvee_{i \in I} \psi_i).$$

Likewise, if in the ith substructure there exists an element satisfying ψ_i then the whole structure satisfies

$$\Diamond\ \mathsf{in}_j(\varphi) \wedge \bigwedge_{i \in I} \Diamond\psi_i.$$

The two axioms $\textbf{Fold} - \Box$ and $\textbf{Fold} - \Diamond$ are equivalent to the following axiom:

Fold

$$\begin{aligned}\vdash \mathsf{fold}\ :\ & \mathsf{in}_j(\varphi \times (\Box \bigvee_{i \in I_1} \psi_{1i} \wedge \bigwedge_{i \in I_1} \Diamond\psi_{1i}) \times \ldots \times (\Box \bigvee_{i \in I_{n_j}} \psi_{n_j i} \wedge \bigwedge_{i \in I_{n_j}} \Diamond\psi_{n_j i})) \\ \rightarrow\ & \Box(\mathsf{in}_j(\varphi) \vee \bigvee_{k=1}^{n_j} \bigvee_{i \in I_k} \psi_{ki}) \wedge \Diamond\mathsf{in}_j(\varphi) \wedge \bigwedge_{k=1}^{n_j} \bigwedge_{i \in I} \Diamond\psi_{ki}\end{aligned}$$

It is straightforward to see that **Fold** follows from $\textbf{Fold} - \Box$ and $\textbf{Fold} - \Diamond$. For the other way round we note that the argument in $\textbf{Fold} - \Box$ (resp. $\textbf{Fold} - \Diamond$) can be rewritten as

$$\Box \bigvee_{i\in I} \varphi_i = \bigvee_{I'\subseteq I} (\Box \bigvee_{i\in I'} \varphi_i \wedge \bigwedge_{i\in I'} \Diamond\varphi_i) \quad \text{resp.,} \quad \Diamond\varphi = \Diamond\mathtt{t} \wedge \Diamond\varphi = \Box(\mathtt{t} \vee \varphi) \wedge \Diamond\mathtt{t} \wedge \Diamond\varphi$$

(cf. the proof of Lemma 2.1.1). Applying **Fold** to the first kind of formulae gives that the result will satisfy a disjunction of formulae of the form

$$\Box(\bigvee_{i\in I'_1} \varphi_i \vee \ldots \vee \bigvee_{i\in I'_{n_j}} \varphi_i) \wedge \Diamond \ldots$$

where $I'_k \subseteq I_k, k = 1,\ldots,n_j$ and where each disjunct implies the result in the **Fold** $-\ \Box$. To derive **Fold** $-\ \Diamond$ we observe that applying **Fold** to the second kind of formulae yields a conjunction and that among the conjuncts is each of the $\Diamond\varphi_k, k = 1,\ldots,n_j$. The rules **Fold** $-\ \Box$ and **Fold** $-\ \Diamond$ are simpler than **Fold** and therefore handier for proving properties of programs. The interest of **Fold** is that it applies to arguments described by irreducible formulae; this is helpful when relating the logic to an abstract interpretation. Similar to fixed points, the behavior of fold on an argument described by a disjunctive formula $\varphi_1 \vee \varphi_2$ can be obtained with a deduction of the form

$$\cfrac{\cfrac{\mathsf{fold} : \varphi_1 \to \psi \qquad\qquad \mathsf{fold} : \varphi_2 \to \psi}{\mathsf{fold} : \varphi_1 \to \psi \wedge \varphi_2 \to \psi}\ \textbf{Conj}}{\mathsf{fold} : (\varphi_1 \vee \varphi_2) \to \psi.}\ \textbf{Weak}$$

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

$$\Box \bigvee_{i\in I} \bigvee_{j\in J_i} \mathsf{in}_i(\varphi_{ij}) \wedge \bigwedge_{i\in I} \bigwedge_{j\in J_i} \Diamond\ \mathsf{in}_i(\varphi_{ij}),$$

where disjuncts with the same sum tag have been grouped together. Unfolding such a structure yields a sum element $\mathsf{in}_k(d, e)$, where $k \in I$, d is the element being taken out, and e is the remaining part of the structure. Since we have no information about the order in which the elements appear in the data structure we only know that the element will have one of the tags in_k for $k \in I$. The formula describing the result must therefore be a disjunction over $i \in I$. Given that the element has tag k it must satisfy the property $\bigvee_{j\in J_k} \varphi_{kj}$. The remaining elements still satisfy $\Box(\bigvee_{i\in I} \bigvee_{j\in J_i} \mathsf{in}_i(\varphi_{ij}))$, since we have not added any elements (the **Unfold** $-\ \Box$ axiom). For the "exists"-part, however, d could have been the element that validated all of the $\Diamond\mathsf{in}_j(\varphi_{kj})$ for $j \in J_k$, so we cannot guarantee that there remain elements satisfying any of these properties. The remaining structures therefore only satisfy the weaker

$$\Box(\bigvee_{i\in I} \bigvee_{j\in J_i} \mathsf{in}_i(\varphi_{ij})) \wedge \bigwedge_{i\in I\setminus\{k\}} \bigwedge_{j\in J_i} \Diamond\ \mathsf{in}_i(\varphi_{ij}),$$

as expressed by the **Unfold** $-\ \Diamond$ axiom.

3.3 Binding-Time and Data-Flow Analyses

Minor modifications to the inference rules suffice to obtain a binding-time analysis. First, the value of constant expressions can always be determined, so these are static. This is expressed with the rule

$$\textbf{Const} - \textbf{BTA} \qquad \Gamma \vdash c : \texttt{f}.$$

Second, expressions built using base operators such as + and = require that all their arguments are completely determined in order for the value to be determined; thus the rule governing these operators is changed to

$$\textbf{Base} - \textbf{BTA} \qquad \frac{\Gamma \vdash e : (\texttt{f} \times \texttt{f})}{\Gamma \vdash \mathit{bop}(e) : \texttt{f}}.$$

Third, evaluation of an if-expression requires that the Boolean condition can be evaluated. In terms of inference rules, this means that the rules $\textbf{If} - \texttt{f}$ and $\textbf{If} - \vee$ must be replaced with

$$\textbf{If} - \textbf{BTA} \qquad \frac{\Gamma \vdash b : \texttt{f}}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \texttt{f}}.$$

The fact that a completely dynamic data structure of sum type cannot be decomposed using a **case**-statement is expressed by the *absence* from the logic of a rule treating this particular case; the only property provable of such an expression is $\texttt{t}$ by the general rule **Taut**.

A data-flow analysis using the properties of integer and Boolean values given in Section 2.2 is obtained by adding a set of inference rules describing the behavior of the base operators and the if-expression on these properties, e.g., that $\Gamma \vdash *(e) : \texttt{neg}$ if $\Gamma :\vdash e : \texttt{pos} \times \texttt{neg}$ where $*$ is multiplication. This is standard and is not detailed further here. Similarly, rules such as

$$\textbf{If} - \texttt{true} \qquad \frac{\Gamma \vdash b : \texttt{true} \quad \Gamma \vdash e_1 : \varphi}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \varphi}$$

must be added so that the treatment of conditional expressions now takes into account that there are cases where we can decide the value of the conditional expression at analysis time and therefore only have to analyze one of the branches in the conditional.

4. PARTIAL-ORDER SEMANTICS

The implication ordering $\leq$ from the formal systems $(L(\sigma), \leq)$ defined in the previous sections is only a preorder, since there are syntactically distinct formulae that imply each other (any of the equalities in Figure 1 can serve as an example). Logically equivalent formulae denote the same *property* so we can obtain the structure of properties axiomatized by the formal systems as the quotient structures $\mathcal{LA}(\sigma) = (L(\sigma)/=, \leq/=)$, called the Lindenbaum algebra of the formal system. The purpose of this section is to describe the lattice structure of the Lindenbaum algebra $\mathcal{LA}(\sigma)$ for each type σ (summarized in Figure 3) and use these lattices to give a denotational semantics that assigns properties to the expressions in our language (Figure 5). This provides us with a method for proving properties of expressions by calculating their denotation (their abstract interpretation) in these lattices. The theorems are formulated for the strictness logic, but the generality of the framework means that small changes allow similar results for the binding-time and data-flow analyses from Section 3.3.

4.1 Order-Theoretic Preliminaries

The Normal Form Theorem (Theorem 2.1.3) states that every property is the finite disjunction of irreducible properties. This splits the task of describing $\mathcal{LA}(\sigma)$ into two:

(1) For each σ, find $\mathcal{J}(\sigma) = (\mathsf{Irr}(\sigma)/{=}, \sqsubseteq)$, the partial order on irreducible formulae.
(2) Define $[\![\sigma]\!]$ to be the set $\mathsf{D}(\mathcal{J}(\sigma))$ of downward closed sets of $\mathcal{J}(\sigma)$, ordered by set inclusion $\subseteq$.

In the following we explain for each type the partial order $\mathcal{J}(\sigma)$ and define an interpretation of the strictness properties of Section 2 as elements in $[\![\sigma]\!]$. We overload the semantic brackets and use them for both interpretations, i.e., if $\varphi \in L(\sigma)$ then $[\![\varphi]\!] \in [\![\sigma]\!]$. We prove that each $[\![\sigma]\!]$ is (isomorphic to) the Lindenbaum algebra of σ by showing that the interpretation of formulae induces an order isomorphism of lattices. First, we need some order-theoretic notions.

Definition 4.1.1. A lower subset of a partial order $(P, \sqsubseteq)$ is a finite nonempty subset S of P that is downward closed, i.e., it satisfies that if $p_1 \sqsubseteq p_2$ and $p_2 \in S$, then $p_1 \in S$. We let $\downarrow\{p_1, \ldots, p_n\}$ denote the set $\{p \mid p \sqsubseteq p_i \text{ for some } p_i\}$.

The set of lower subsets of P is denoted by $\mathsf{D}(P)$. Under certain conditions, this provides a technique for building distributive lattices from partial orders.

LEMMA 4.1.2. *If P is a finite partial order with least element $\perp$, then $(\mathsf{D}(P), \cup, \cap, \subseteq)$ is a lattice with least element the lower set $\{\perp\}$ and greatest element P. The function $p \mapsto \downarrow\{p\}$ is an order embedding.*

PROOF. We observe that the existence of a least element in P guarantees that the intersection of two lower sets is nonempty; thus $\cap$ yields an element of $\mathsf{D}(P)$. The rest of the verification is straightforward. □

The lemma is a variation of a general result from lattice theory known as Birkhoff's Representation Theorem [Davey and Priestley 1990, Theorem 8.17] stating that every distributive lattice can be represented as a lattice of lower sets of the partial order of its join-irreducible elements.

We recall that a function $f : (A, \leq) \to (B, \sqsubseteq)$ between partial orders is called *monotone* if $a_1 \leq a_2 \;\Rightarrow\; f(a_1) \sqsubseteq f(a_2)$. If A and B are lattices with least upper bound operators $\vee$ and $\sqcup$, we say that f is *linear* if $f(a_1 \vee a_2) = f(a_1) \sqcup f(a_2)$. We write $A \to B$ and $A \multimap B$ for the set of monotone, (resp., linear) functions from A to B. A function $f : (A \times B) \to C$ is *bilinear* if it is linear in each component of the argument separately, i.e., $f((a_1 \sqcup a_2), b) = f((a_1, b)) \sqcup f((a_2, b))$ and similarly for the second component.

4.2 Base Types, Products, and Sums

An irreducible formula of integer or Boolean type is equivalent to either `t` or `f`, so the two-point lattice $\mathbf{2} = \{0 \sqsubseteq 1\}$ suffices. Thus $[\![\mathsf{Int}]\!] = [\![\mathsf{Bool}]\!] = \mathsf{D}(\mathbf{2})$. The unit type 1 has one property `t` that is modeled by the one-point lattice $\mathbf{1} = \{\bullet\}$. An irreducible formula of type $\sigma \times \tau$ is of form $\varphi \times \psi$ with φ, ψ irreducible. We therefore define

$$\mathcal{J}(\sigma \times \tau) = \mathcal{J}(\sigma) \times \mathcal{J}(\tau);$$

$$\begin{array}{lcl}
\mathcal{J}(1) & = & \mathbf{1} \\
\mathcal{J}(\mathsf{Int}) & = & \mathbf{2} \\
\mathcal{J}(\mathsf{Bool}) & = & \mathbf{2} \\
\mathcal{J}(\sigma \times \tau) & = & \mathcal{J}(\sigma) \times \mathcal{J}(\tau) \\
\mathcal{J}(\sigma + \tau) & = & \mathcal{J}(\sigma) \oplus \mathcal{J}(\tau) \\
\mathcal{J}(\sigma \to \tau) & = & [\![\sigma]\!] \multimap [\![\tau]\!] \\
\mathcal{J}(\mu\alpha. \sum_{i\in I} \sigma_i \times \alpha^{n_i}) & = & (P_C(\bigcup_{i\in I} \mathcal{J}(\sigma_i)))_\emptyset \\
 & & \\
[\![\sigma]\!] & = & \mathsf{D}(\mathcal{J}(\sigma))
\end{array}$$

Fig. 3. Types modeled as lower sets.

thus the lattice modeling $\sigma \times \tau$ consists of lower sets of pairs where the elements of the pairs correspond to irreducible properties:

$$[\![\sigma \times \tau]\!] = \mathsf{D}(\mathcal{J}(\sigma) \times \mathcal{J}(\tau)) = \mathsf{D}(\{(a, b) \mid {\downarrow}\{a\} \in [\![\sigma]\!], {\downarrow}\{b\} \in [\![\tau]\!]\}).$$

The interpretation of a formula $\varphi \times \psi$ is defined by

$$[\![\varphi \times \psi]\!]_{\sigma\times\tau} = \{(a, b) \mid a \in [\![\varphi]\!]\,,\, b \in [\![\psi]\!]\}.$$

When $[\![\varphi]\!], [\![\psi]\!]$ are downward closed this construction yields a downward closed set. The first and second projections π_1 and π_2 that take out the first (resp., the second) component of a pair are extended to lower subsets of pairs by applying the relevant projection to each pair in the set. Due to the ordering on pairs this gives a lower set. Formally we define $\pi_1 : [\![\sigma \times \tau]\!] \to [\![\sigma]\!]$ by

$$\pi_1(\bigcup_{i=1}^{n} {\downarrow}\{(a_i, b_i)\}) = \bigcup_{i=1}^{n} {\downarrow}\{a_i\}$$

and similarly for π_2.

The use of tensor products in program analysis has received some attention, so it is worth pointing out that the lattice $[\![\sigma \times \tau]\!]$ is a concrete representation of the tensor product $[\![\sigma]\!] \otimes [\![\tau]\!]$. The embedding from the Cartesian to the tensor product is defined by

$$\begin{array}{l}
i : \mathsf{D}(\mathcal{J}(\sigma)) \times \mathsf{D}(\mathcal{J}(\tau)) \to \mathsf{D}(\mathcal{J}(\sigma) \times \mathcal{J}(\tau)) \\
i : (\bigcup_{i\in I} {\downarrow}\{s_i\}, \bigcup_{j\in J} {\downarrow}\{t_j\}) \mapsto \bigcup_{(i,j)\in I\times J} {\downarrow}\{(s_i, t_j)\}
\end{array}$$

and the extension of a bilinear function $f : [\![\sigma]\!] \times [\![\tau]\!] \to L$ to a linear $\hat{f} : [\![\sigma]\!] \otimes [\![\tau]\!] \to L$ by

$$\hat{f}(\bigcup_{i\in I} {\downarrow}\{(s_i, t_i)\}) = \bigsqcup_{i\in I} f(({\downarrow}\{s_i\}, {\downarrow}\{t_i\})),$$

which obviously makes $\hat{f}$ linear. It remains to verify that $\hat{f}$ really does extend f, i.e., $f = \hat{f} \circ i$ for bilinear f. This is shown by the following calculations.

$$\begin{aligned} \hat{f} \circ i((\bigcup_{i \in I} \downarrow\{s_i\}, \bigcup_{j \in J} \downarrow\{t_j\})) &= \hat{f}(\bigcup_{(i,j) \in I \times J} \downarrow\{(s_i, t_j)\}) \\ = \bigsqcup_{(i,j) \in I \times J} f((\downarrow\{s_i\}, \downarrow\{t_j\})) &= \bigsqcup_{i \in I} (\bigsqcup_{j \in J} f((\downarrow\{s_i\}, \downarrow\{t_j\}))) \\ = \bigsqcup_{i \in I} f((\downarrow\{s_i\}, \bigcup_{j \in J} \downarrow\{t_j\})) &= f((\bigcup_{i \in I} \downarrow\{s_i\}, \bigcup_{j \in J} \downarrow\{t_j\})) \end{aligned}$$

We comment further on tensor products in Section 6 on related works.

For sums, we have that an irreducible of a sum type is either $\mathtt{f}$ or is of the form $\mathsf{in}_i(\varphi)$ with φ irreducible. Two tagged formulae $\mathsf{in}_i(\varphi)$ and $\mathsf{in}_j(\psi)$ are comparable if and only if $i = j$. The formula $\mathtt{f}$ is smallest in the ordering and different from all other formulae. We can thus use the well-known separated sum construction $\oplus$ [Davey and Priestley 1990] to obtain a partial order modeling the implication between irreducibles of sum types. If in_i is the tagging operation for the separated sum, then we interpret sum formulae by

$$[\![\mathsf{in}_i(\varphi)]\!] = \{in_i(d) : d \in [\![\varphi]\!]\} \cup \{\bot_\Sigma\},$$

where $\bot_\Sigma$ is the new bottom element added by the separate sum construction.

4.3 Functions

An irreducible formula of function type $\sigma \to \tau$ has the form $\bigwedge(\varphi_i \to \psi_i)$, with every φ_i irreducible. Intuitively, a property $\varphi \to \psi$ of a function means that if the argument to the function satisfies φ, then the result satisfies ψ. With the interpretation of φ_i and ψ_i as elements of a lattice of properties, we can interpret $\varphi \to \psi$ as a set of functions f on the relevant lattices that satisfy $f([\![\varphi]\!]) \subseteq [\![\psi]\!]$. Since all the φ_i are irreducible this only specifies f's behavior on downward closures of single elements. The axiom

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

enforces that the extension to arbitrary lower sets must make f a linear function. This motivates the definition:

$$\mathcal{J}(\sigma \to \tau) = [\![\sigma]\!] \multimap [\![\tau]\!]$$

The following lemma establishes a connection between formulae of function type and linear functions. It is used later in an inductive proof of the fact that the interpretation of an irreducible formula is a downward closure of a single element.

LEMMA 4.3.1. *Given $\{\varphi_i\}_{i \in I} \subseteq L(\sigma), \{\psi_i\}_{i \in I} \subseteq L(\tau)$ such that $[\![\varphi_i]\!]$ is of the form $\downarrow\{c_i\}$ for all $i \in I$, let $f : [\![\sigma]\!] \to [\![\tau]\!]$ be the function defined by*

$$f(\downarrow\{a_1, \ldots, a_n\}) = \mathcal{J}(\tau) \cap \bigcap\{[\![\psi_j]\!] : \downarrow\{a_1, \ldots, a_n\} \subseteq [\![\varphi_j]\!]\}.$$

Then f is linear, and we have

$$[\![\bigwedge_{i \in I}(\varphi_i \to \psi_i)]\!] = \downarrow\{f\}.$$

PROOF. This defines a linear function, since $\downarrow\{a_1, \ldots, a_n\} \subseteq [\![\varphi_j]\!] = \downarrow\{c_j\}$ if and only if $\downarrow\{a_i\} \subseteq [\![\varphi_j]\!]$ for each $i = 1, \ldots, n$. Hence $\downarrow\{a_1, \ldots, a_n\}$ is mapped to the same intersection as each of the $\downarrow\{a_i\}$, so

$$f(\downarrow\{a_1,\ldots,a_n\}) = \bigcup_{i=1}^{n} f(\downarrow\{a_i\}).$$

Furthermore, any g satisfying $g([\![\varphi_i]\!]) \subseteq [\![\psi_i]\!]$ for all $i \in I$ satisfies

$$\begin{aligned} g(\downarrow\{a_1,\ldots,a_n\}) &\subseteq \bigcap\{g([\![\varphi_j]\!]) : \downarrow\{a_1,\ldots,a_n\} \subseteq [\![\varphi_j]\!]\} \\ &\subseteq \bigcap\{[\![\psi_j]\!] : \downarrow\{a_1,\ldots,a_n\} \subseteq [\![\varphi_j]\!]\} \\ &\subseteq f(\downarrow\{a_1,\ldots,a_n\}), \end{aligned}$$

showing that $g \in \downarrow\{f\}$. $\square$

4.4 Recursive Data Structures

The modeling of uniform properties of recursive data structures builds on the work by Abramsky [1991b] on characterizing powerdomains logically, which in turn is inspired by Winskel's work on the relationship between modalities and powerdomains [Winskel 1983]; see also Gunter and Scott [1990]. A formula

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

describes those structures where every element satisfies $\bigvee_i \varphi_i$, and for every property φ_i there exists at least one element that satisfies this property. Those irreducible φ_i are interpreted as $\downarrow\{d_i\}$ for some $d_i \in \mathcal{J}(\sigma)$, so we can represent such a formula by the set of elements $\{d_i : [\![\varphi_i]\!] = \downarrow\{d_i\}\} \subseteq \mathcal{J}(\sigma)$. 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

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

Transferring this to subsets of $\mathcal{J}(\sigma)$ with $\downarrow\{d_i\} = [\![\varphi_i]\!]$ and $\downarrow\{e_j\} = [\![\psi_j]\!]$ we arrive at

$$\forall i \in I\ \exists j \in J.\ d_i \sqsubseteq e_j \quad \text{and} \quad \forall j \in J\ \exists i \in I.\ d_i \sqsubseteq e_j.$$

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

Definition 4.4.1. The Egli-Milner ordering $\sqsubseteq_{EM}$ on nonempty subsets of a partial order $(P, \sqsubseteq)$ is defined by

$$S \sqsubseteq_{EM} T \quad \equiv \quad \forall s \in S\ \exists t \in T.\ s \sqsubseteq t \quad \text{and} \quad \forall t \in T\ \exists s \in S.\ s \sqsubseteq t.$$

Furthermore define the equivalence $=_{EM}$ by

$$S =_{EM} T \equiv S \sqsubseteq_{EM} T \text{ and } T \sqsubseteq_{EM} S.$$

It is well known that the Egli-Milner ordering is in general only a preorder on subsets (e.g., 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 4.4.2. 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 nonempty set S is the least (with respect to $\subseteq$) convex set that contains S. We write this set as $\overline{S}$.

It is straightforward to see that for a subset S we always have $S\sqsubseteq_{EM}\overline{S}$ and $\overline{S}\sqsubseteq_{EM}S$. 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 $=_{EM}$.

Definition 4.4.3. For A a partial order, define $P_C(A)$, the partial order of nonempty convex sets of A, by

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

For integers we have the formulae $\mathtt{t}$ and $\mathtt{f}$, and the partial order of irreducibles is $\mathcal{J}(\mathsf{Int}) = \{0 \sqsubseteq 1\}$. This yields the set of convex sets:

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

We can interpret this as properties of nonempty sets of integers as follows. 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 are undefined. The middle point $\downarrow\{\{0,1\}\}$ is the interpretation of the formula $\Box(\mathtt{t}\vee\mathtt{f})\wedge\Diamond\mathtt{t}\wedge\Diamond\mathtt{f} = \Box\mathtt{t}\wedge\Diamond\mathtt{f}$ which states that among the elements at least one is undefined.

The irreducible formulae of type $\mu\alpha. \sum_{k=1}^{n} \sigma_i \times \alpha^{n_k}$ are of the form

$$\Box(\bigvee_{i\in I} \varphi_i) \wedge \bigwedge_{i \in I}(\Diamond\varphi_i) \qquad \varphi_i \in \mathsf{Irr}(L(\sum_{k=1}^{n} \sigma_k)),$$

so our partial order $\mathcal{J}(\mu\alpha. \sum_{k=1}^{n} \sigma_k \times \alpha^{n_k})$ will consist of convex sets from the sum of the partial orders $\mathcal{J}(\sigma_k)$. However, as was noted in Section 2 when discussing the axioms for $\Box$ and $\Diamond$, recursive structures will not contain the undefined sum value—all elements are forcibly tagged. In order to exclude the bottom element $\perp_{\sigma_1+\ldots+\sigma_n}$ from the convex sets we use the disjoint *union* of the $\mathcal{J}(\sigma)$ and take convex sets of this partial order. This still leaves us with the problem of dealing with the completely undefined structure that contains no elements and therefore, following the preceding idea, should be modeled by the empty set. The empty set is not an element in the partial order of convex sets (it is incomparable to all other sets in the $\sqsubseteq_{EM}$ order), but we can adjoin it explicitly to the partial order. To serve our purposes we place it as the least element in the extended order; that is, we *lift* the partial order of convex sets with the element $\emptyset$.[1] This yields the following definition.

Definition 4.4.4.

$$\mathcal{J}(\mu\alpha.\sum_{k=1}^{n}\sigma_k \times \alpha^{n_k}) = (P_C(\bigcup_{k=1}^{n}\mathcal{J}(\sigma_k)))_\emptyset$$

The formulae are interpreted as lower sets of this partial order i.e., as lower sets of convex sets. The interpretation is given by

$$\begin{aligned} [\![\Box\varphi]\!]_{\mu\alpha.T} &= \{C : C \subseteq [\![\varphi]\!]_\sigma\} \\ [\![\Diamond\varphi]\!]_{\mu\alpha.T} &= \{C : C \cap [\![\varphi]\!] \text{ is nonempty}\} \cup \{\emptyset\}. \end{aligned}$$

The interpretation of formula $\mathtt{f}$ follows the standard pattern and is defined to be the singleton set containing the least element $\emptyset$. It is immediate to check that $[\![\Box\mathtt{f}]\!] = [\![\Diamond\mathtt{f}]\!] = \{\emptyset\}$, so the axioms $\Box - \mathtt{f}$ and $\Diamond - \mathtt{f}$ are satisfied under this interpretation.

In the case of integer lists (defined by $\mu\alpha.\mathsf{Int} \times \alpha + 1$) this gives the following partial order of irreducibles:

$$\mathcal{J}(\mu\alpha.\mathsf{Int} \times \alpha + 1) = P_C(\mathcal{J}(\mathsf{Int}) \cup \mathcal{J}(1))_\emptyset = P_C(\{in_1(0), in_1(1), in_2(\bullet)\})_\emptyset$$

The Hasse diagram of this partial order is shown in Figure 6, and Section 5 describes in more detail the properties of lists obtained by this construction. Here we just show the interpretation of the formula $\Box(\mathsf{in}_1(\mathtt{f}^{\mathsf{Int}}) \vee \mathsf{in}_2(\mathtt{t}^1))$. First we find that

$$[\![\mathsf{in}_1(\mathtt{f}^{\mathsf{Int}}) \vee \mathsf{in}_2(\mathtt{t}^1)]\!] = [\![\mathsf{in}_1(\mathtt{f}^{\mathsf{Int}})]\!] \cup [\![\mathsf{in}_2(\mathtt{t}^1)]\!] = \{in_1(0), \bot^{\mathsf{Int}+1}, in_2(\bullet)\}.$$

The set $[\![\Box(\mathsf{in}_1(\mathtt{f}^{\mathsf{Int}}) \vee \mathsf{in}_2(\mathtt{t}^1))]\!]$ consists of all those sets in $P_C(\{in_1(0), in_1(1), in_2(\bullet)\})_\emptyset$ that are subsets of $\{in_1(0), \bot_{\mathsf{Int}+1}, in_2(\bullet)\}$. This gives the following interpretation:

$$[\![\Box(\mathsf{in}_1(\mathtt{f}^{\mathsf{Int}}) \vee \mathsf{in}_2(\mathtt{t}^1))]\!] = \{\emptyset, \{in_1(0)\}, \{in_2(\bullet)\}, \{in_1(0), in_2(\bullet)\}\}$$

The following lemma links convex sets with the irreducible formulae for recursive types.

LEMMA 4.4.5. *Let $c_i \in \mathcal{J}(\sum_{k \in K}\sigma_k)$ satisfy that $[\![\varphi_i]\!] = \downarrow\{c_i\}$ for all $i \in I$. Then*

$$[\![\Box(\bigvee_{i \in I}\varphi_i) \wedge \bigwedge_{i \in I}(\Diamond\varphi_i)]\!] = \downarrow\{\overline{\{c_i : i \in I\}}\}.$$

PROOF. For all $C' \in [\![\Box(\bigvee_{i \in I}\varphi_i) \wedge \bigwedge_{i \in I}(\Diamond\varphi_i)]\!]$ we have

—$C' \subseteq [\![\bigvee_{i \in I}\varphi_i]\!] = \bigcup_{i \in I}\downarrow\{c_i\}$,
—$C' \cap [\![\varphi_i]\!] = C' \cap \downarrow\{c_i\}$ is nonempty for all $i \in I$,

from which it follows that $C' \sqsubseteq_{EM} \{c_i : i \in I\}$ and therefore that $C' \sqsubseteq_{EM} \overline{\{c_i : i \in I\}}$, so

$$[\![\Box(\bigvee_{i \in I}\varphi_i) \wedge \bigwedge_{i \in I}(\Diamond\varphi_i)]\!] \subseteq \downarrow\{\overline{\{c_i : i \in I\}}\}.$$

[1]It should be noted that this is neither the only nor the standard way of dealing with $\emptyset$. Other solutions and uses of "adjoining the empty set" are discussed by Abramsky [1991a].

$$
\begin{array}{rcl}
& & \text{Let } \varphi, \varphi_1, \varphi_2 \in \mathcal{L}(\sigma), \psi \in \mathcal{L}(\tau) \text{ and } T = \sum_{k\in K} \sigma_k \times \alpha^{n_k} \\
& & [\![]\!]_\sigma : L(\sigma) \to [\![\sigma]\!] \\
[\![\mathtt{t}]\!]_\sigma & = & \mathcal{J}(\sigma) \\
[\![\mathtt{f}]\!]_\sigma & = & \{\bot_{\mathcal{J}(\sigma)}\} \\
[\![\varphi_1 \wedge \varphi_2]\!]_\sigma & = & [\![\varphi_1]\!]_\sigma \cap [\![\varphi_2]\!]_\sigma \\
[\![\varphi_1 \vee \varphi_2]\!]_\sigma & = & [\![\varphi_1]\!]_\sigma \cup [\![\varphi_2]\!]_\sigma \\
[\![\varphi \times \psi]\!]_{\sigma\times\tau} & = & \{(a,b) \mid a \in [\![\varphi]\!]_\sigma, b \in [\![\psi]\!]_\tau\} = [\![\varphi]\!]_\sigma \times [\![\psi]\!]_\tau \\
[\![\varphi \to \psi]\!]_{\sigma\to\tau} & = & \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]_\sigma) \subseteq [\![\psi]\!]_\tau\} \\
[\![\mathsf{in}_i(\varphi)]\!]_{\Sigma_{i\in I}\sigma_i} & = & \{in_i(d) : d \in [\![\varphi]\!]_{\sigma_i}\} \cup \{\bot_{\mathcal{J}(\Sigma_{i\in I}\sigma_i)}\} \\
[\![\Box\varphi]\!]_{\mu\alpha.T} & = & \{C : C \subseteq [\![\varphi]\!]_{\Sigma_{k\in K}\sigma_k}\} \\
[\![\Diamond\varphi]\!]_{\mu\alpha.T} & = & \{C : C \cap [\![\varphi]\!]_{\Sigma_{k\in K}\sigma_k} \text{ is nonempty}\} \cup \{\emptyset\}
\end{array}
$$

Fig. 4. Semantics of formulae.

The other inclusion follows from observing that $\{c_i : i \in I\}$ itself satisfies

—$\{c_i : i \in I\} \subseteq [\![\bigvee_{i\in I} \varphi_i]\!] = \bigcup_{i\in I} \downarrow\{c_i\}$,
—$\{c_i : i \in I\} \cap [\![\varphi_i]\!] = \cap\downarrow\{c_i\}$ is nonempty for all $i \in I$.

Now, $\bigcup_{i\in I} \downarrow\{c_i\}$ is downward closed and therefore convex, so the convex closure $\overline{\{c_i : i \in I\}}$ is also contained in that set. Furthermore, since taking the convex closure of a set only adds elements to the set, $\overline{\{c_i : i \in I\}} \cap \downarrow\{c_i\}$ is nonempty for all $i \in I$. This means that

$$\overline{\{c_i : i \in I\}} \in [\![\Box(\bigvee_{i\in I} \varphi_i) \wedge \bigwedge_{i\in I}(\Diamond\varphi_i)]\!];$$

hence we have $\downarrow\{\overline{\{c_i : i \in I\}}\} \subseteq [\![\Box(\bigvee_{i\in I} \varphi_i) \wedge \bigwedge_{i\in I}(\Diamond\varphi_i)]\!]$. □

4.5 Interpretation of Formulae

The logical formulae of $\mathcal{L}(\sigma)$ are interpreted in $[\![\sigma]\!]$ using the function $[\![\ ]\!]_\sigma$ defined in Figure 4. The exact connection between logic and semantics is given in the theorems to follow. First we show that every irreducible formula is interpreted as the downward closure of a single element. The Definability Theorem (4.5.2) states that every downward closure of a single element is denoted by an irreducible formula. From this it follows that every lower set is denoted by some formula, so the interpretation $[\![]\!]$ defines a surjective function from formulae to lower sets.

THEOREM 4.5.1. *For all irreducible $\varphi \in L(\sigma)$ there exists $c \in \mathcal{J}(\sigma)$ such that $[\![\varphi]\!] = \downarrow\{c\}$.*

PROOF. The proof follows by induction over the definition of the predicate Irr. The difficult cases are those of function and recursive types that are consequences of Lemma 4.3.1 and Lemma 4.4.5. □

THEOREM 4.5.2 (DEFINABILITY). *For every $\downarrow\{a\} \in [\![\sigma]\!]$ there exists a $\varphi_a \in \mathcal{L}(\sigma)$ such that*

$$\mathsf{Irr}(\varphi_a) \quad \textit{and} \quad \downarrow\{a\} = [\![\varphi_a]\!]_\sigma.$$

PROOF. See Appendix A.2. □

The Soundness Theorem (4.5.3) states that any entailment that can be proved in the formal system is properly reflected in the semantics as an inclusion between subsets. As a consequence of the Soundness Theorem we have that if two formulae are provably equal, then they denote the same lower set.

THEOREM 4.5.3 (SOUNDNESS). *For all elements $\varphi, \psi \in \mathcal{L}(\sigma)$ we have*

$$\varphi \leq \psi \quad \Rightarrow \quad [\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma.$$

PROOF. See Appendix A.3. □

The Completeness Theorem (4.5.4) states that all inclusions between lower sets that hold in the semantics can be proved in the formal system. Since all lower sets in $[\![\sigma]\!]$ are denotable by a formula in $L(\sigma)$ this can be established by proving that an inclusion between the interpretation of two formulae corresponds to a logical entailment between the two formulae in the formal system.

THEOREM 4.5.4 (COMPLETENESS). *For all elements $\varphi, \psi \in \mathcal{L}(\sigma)$ we have*

$$[\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma \quad \Rightarrow \quad \varphi \leq \psi.$$

PROOF. See Appendix A.4. □

For a corollary of these theorems we can show that irreducible formulae cannot be written as a nontrivial disjunction.

COROLLARY 4.5.5. *Let $\varphi, \varphi_1, \varphi_2 \in L(\sigma)$, and assume that $\mathsf{Irr}(\varphi)$. Then,*

$$\varphi = \varphi_1 \vee \varphi_2 \quad \Rightarrow \quad \varphi = \varphi_1 \text{ or } \varphi = \varphi_2.$$

PROOF. By Theorem 4.5.1 there exists an a such that $[\![\varphi]\!] = \downarrow\{a\}$, so by the Soundness Theorem 4.5.3 $\downarrow\{a\} = [\![\varphi_1 \vee \varphi_2]\!] = [\![\varphi_1]\!] \cup [\![\varphi_2]\!]$. Hence $a \in [\![\varphi_1]\!]$ or $a \in [\![\varphi_2]\!]$, and therefore $[\![\varphi]\!] = [\![\varphi_1]\!]$ or $[\![\varphi]\!] = [\![\varphi_2]\!]$. So, by Completeness, we have $\varphi = \varphi_1$ or $\varphi = \varphi_2$. □

For another corollary we get the order-isomorphism showing that $[\![\sigma]\!]$ is the Lindenbaum algebra $\mathcal{LA}(\sigma)$.

COROLLARY 4.5.6. *For every type σ, the mapping*

$$[\![]\!] : L(\sigma) \rightarrow [\![\sigma]\!]$$

induces an order-isomorphism between $\mathcal{LA}(\sigma)$ and $[\![\sigma]\!]$.

PROOF. The induced mapping $[\![]\!] : \mathcal{LA}(\sigma) \rightarrow [\![\sigma]\!]$ that for any $\varphi \in L(\sigma)$ sends $[\varphi]$ to $[\![\varphi]\!]$ is well defined by the Soundness Theorem (4.5.3), surjective (Definability (4.5.2)), and order-preserving (Soundness and Completeness (4.5.3 and 4.5.4)), hence an order-isomorphism. □

4.6 Abstract Interpretation

We now define an abstract interpretation that assigns to each expression of type σ a disjunctive property represented by an element of the lattice $[\![\sigma]\!]$. More precisely, we define the abstract interpretation as a mapping

$$[\![e]\!] : Env \rightarrow [\![\sigma]\!]$$

$$
\begin{array}{rcll}
[\![x^\sigma]\!]\rho &=& \rho(x) & \\
[\![c^\sigma]\!]\rho &=& \downarrow\{1\} & \sigma = \mathsf{Int}, \mathsf{Bool} \\
[\![(e_1,e_2)]\!]\rho &=& \bigcup \{[\![e_1]\!][x_i \mapsto \downarrow\{v_i\}] \times [\![e_2]\!][x_i \mapsto \downarrow\{v_i\}] \mid v_i \in \rho(x_i)\} & \\
[\![\mathsf{fst}(e)]\!]\rho &=& \pi_1([\![e]\!]\rho) & \\
[\![\mathsf{snd}(e)]\!]\rho &=& \pi_2([\![e]\!]\rho) & \\
[\![\lambda x.e]\!]\rho &=& \downarrow\{\mathsf{D}(\lambda v.[\![e]\!]\rho[x \mapsto \downarrow\{v\}])\} & \\
[\![e_1 e_2]\!]\rho &=& \bigcup \{[\![e_1]\!][x_i \mapsto \downarrow\{v_i\}] \mathbin{@} [\![e_2]\!][x_i \mapsto \downarrow\{v_i\}] \mid v_i \in \rho(x_i)\} & \\
[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\rho &=& \begin{cases} \downarrow\{\bot\} & \text{if } [\![b]\!]\rho = \downarrow\{0\} \\ [\![e_1]\!]\rho \cup [\![e_2]\!]\rho & \text{otherwise} \end{cases} & \\
[\![\mathsf{in}_i(e)]\!]\rho &=& \downarrow\{in_i(d) : d \in [\![e]\!]\rho\} & \\
[\![\mathsf{case}\ e\ \mathsf{of}\ \{\mathsf{in}_i(x_i)\ .\ e_i\}]\!]\rho &=& \begin{cases} \downarrow\{\bot\} & \text{if } [\![e]\!]\rho = \downarrow\{\bot_\Sigma\} \\ \bigcup\{[\![e_i]\!]\rho[x_i \mapsto \downarrow\{d\}] : in_i(d) \in [\![e]\!]\rho\} & \text{otherwise} \end{cases} & \\
[\![bop]\!] &=& \downarrow\{\mathsf{D}(\lambda(a,b).a \sqcap b)\} & \\
[\![\mathsf{fix}]\!] &=& \downarrow\{\mathsf{D}(\lambda f. \bigcup_{n=0}^{\infty} f^n(\{\bot\}))\} & \\
[\![\mathsf{fold}]\!] &=& \downarrow\{\mathsf{D}(fold)\} & \\
[\![\mathsf{unfold}]\!] &=& \downarrow\{\mathsf{D}(unfold)\} &
\end{array}
$$

Fig. 5. Disjunctive abstract interpretation.

which, given an environment associating each free variable x^τ in e with a value from $[\![\tau]\!]$, maps λ-terms e of type σ into an element of $[\![\sigma]\!]$. Our main objective is to prove that the interpretation is a sound and complete model of the disjunctive program logic and in this sense is a *disjunctive* abstract interpretation. The interpretation provides an effective, iteration-based way of calculating the strictness properties provable of an expression in our logic.

We note that, as a first important point, the environment ρ maps variables to lower sets. These lower sets represents disjunctions, so if $\rho(x) = \downarrow\{a_1, a_2\} = \downarrow\{a_1\} \cup \downarrow\{a_2\}$ it means that x either satisfies the property $\downarrow\{a_1\}$ or the property $\downarrow\{a_2\}$. In the previous section we observed that we have to make the same choice of properties for the variables in the environment when analyzing the subexpressions of an expression. In the abstract interpretation this is relevant for the analysis of pairing and application, which have more than one subexpression. We must analyze the subexpressions first with x bound to $\downarrow\{a_1\}$ and then with x bound to $\downarrow\{a_2\}$ and take the combination of the results of these analyses. This is achieved by forming the union of the results obtained by analyzing the expression with all possible choices of "simple" environments $[x \mapsto \downarrow\{v\}]$ for $v \in \rho(x)$.

Expressions of function type are interpreted as lower sets of linear functions. The application of a lower set of functions from $[\![\sigma \to \tau]\!]$ to an argument from $[\![\sigma]\!]$ (i.e., a lower set $L \subseteq \mathcal{J}(\sigma)$ is denoted by @ and defined by

$$\downarrow\{f_1, \ldots, f_n\} \mathbin{@} L = \bigcup_{i=1}^{n} f_i(L).$$

The interpretation of the fixed-point operator is the linear extension of the least fixed-point operator on lattices. It accepts a set of abstract functions, computes the least fixed point of each function, and returns the union of these fixed points:

$$\downarrow\{\lambda f_1, \ldots, f_n. \bigcup_{i=1}^{n} \bigcup_{k=0}^{\infty} f^k(\bot)\}$$

In a similar way, built-in functions such as addition and multiplication are defined by linear extension. The addition function + is interpreted as:

$$[\![+]\!](\bigcup_{i=1}^{n} \downarrow\{(a_i, b_i)\}) = \bigcup_{i=1}^{n} \downarrow\{a_i \sqcap b_i\}$$

which captures that + is bistrict:

$$[\![+]\!]\downarrow\{\{(1,0),(0,1)\}\} = \downarrow\{0\sqcap 1\}\cup\downarrow\{1\sqcap 0\} = \downarrow\{0\}\cup\downarrow\{0\} = \downarrow\{0\}$$

More generally, from a monotone function $f : \mathcal{J}(\sigma) \to [\![\tau]\!]$ we obtain an element $\mathsf{D}(f)$ in $[\![\sigma \to \tau]\!]$ as the downward closure of the linear extension of f.

Definition 4.6.1. Let $f : \mathcal{J}(\sigma) \to [\![\tau]\!]$ be a monotone function. Define

$$\begin{aligned} &\mathsf{D}(f) : [\![\sigma]\!] \multimap [\![\tau]\!] \\ &\mathsf{D}(f) = \lambda L. \textstyle\bigcup_{l \in L} f(l). \end{aligned}$$

This technique is used when defining the semantics of operations that were axiomatized on irreducible formulae only (see discussion in Section 3.2) and extended to arbitrary disjunctions via linearity. This is also the case for fold and unfold for which we define the following two operations.

Definition 4.6.2. Let $C_{\neq j}$ be the set $\{in_i(d) \in C : i \neq j\}$. Recall that $\overline{C}$ is the convex closure of C. Writing $\tilde{C}^{n_j}$ for an n_j-tuple $(\tilde{C}, \ldots, \tilde{C})$, we define

$$\begin{aligned} \mathit{fold} &: in_j(d, C_1, \ldots, C_{n_j}) \longmapsto \downarrow\{\overline{C_1 \cup \ldots \cup C_{n_j} \cup in_j(d)}\} \\ \mathit{unfold} &: C \longmapsto \downarrow\{in_j(d, \tilde{C}^{n_j}) : in_j(d) \in C \text{ and } C_{\neq j} \subseteq \tilde{C} \subseteq C\}. \end{aligned}$$

For *fold*, each of the convex sets $C_1, \ldots, C_{n_j}$ represents the content of the substructures. Folding an element d with tag i onto these structures results in a structure whose content is described by the union of all these sets plus the set $\{in_i(d)\}$. The equivalence class of this set is obtained by taking its convex closure. The definition of *unfold* is motivated by observing that when putting the two axioms unfold $- \Box$ and unfold $- \Diamond$ together the result will be described by the conjunction of the formulae

$$\bigvee_{k \in I} \mathsf{in}_k((\bigvee_{j \in J_k} \varphi_{kj}) \times (\Box \bigvee_{i \in I} \bigvee_{j \in J_i} \mathsf{in}_i(\varphi_{ij}))^{n_k})$$

and

$$\bigvee_{k \in I} \mathsf{in}_k((\bigvee_{j \in J_k} \varphi_{kj}) \times (\bigwedge_{i \in I \setminus \{k\}} \bigwedge_{j \in J_i} \Diamond\, \mathsf{in}_i(\varphi_{ij}))^{n_k}).$$

This conjunction can be reduced to

$$\bigvee_{k \in I} \mathsf{in}_k((\bigvee_{j \in J_k} \varphi_{kj}) \times ((\Box \bigvee_{i \in I} \bigvee_{j \in J_i} \mathsf{in}_i(\varphi_{ij}))^{n_k} \wedge (\bigwedge_{i \in I \setminus \{k\}} \bigwedge_{j \in J_i} \Diamond\, \mathsf{in}_i(\varphi_{ij}))^{n_k}))$$

by the rules for $\times, +$, and distributivity. The description of the substructures can furthermore be put into the form

$$(\Box \bigvee_{i \in I} \bigvee_{j \in J_i} \mathsf{in}_i(\varphi_{ij}) \wedge (\bigwedge_{i \in I \setminus \{k\}} \bigwedge_{j \in J_i} \Diamond\ \mathsf{in}_i(\varphi_{ij})))^{n_k}.$$

The formula describing the components is almost in normal form; we just do not know which set of $\mathsf{in}_k(\varphi_{kj})$ exists in the structure (cf. the discussion at the end of Section 3.2). The normal form is the disjunction over each possible choice:

$$\bigvee_{J \subseteq J_k} \Box[\bigvee_{i \in I \setminus \{k\}} \bigvee_{j \in J_i} \mathsf{in}_i(\varphi_{ij}) \vee \bigvee_{j \in J} \mathsf{in}_k(\varphi_{kj})] \wedge \bigwedge_{i \in I \setminus \{k\}} \bigwedge_{j \in J_i} \Diamond\ \mathsf{in}_i(\varphi_{ij}) \wedge \bigwedge_{j \in J} \Diamond\ \mathsf{in}_k(\varphi_{kj})$$

Semantically, this choice is mirrored in the choice of a convex set $\tilde{C}$ satisfying the inclusions

$$\{in_l(d') \in C : l \neq k\} \subseteq \tilde{C} \subseteq C$$

when defining *unfold* on a convex set C. This is justified formally in the proof of Theorem 4.7.1 in Appendix A.6.

4.7 Soundness and Completeness

The following theorem summarizes the relationship between the logic and the abstract interpretation: the abstract interpretation of an expression is the strongest property that the logic can prove of that expression. In other words, the logic can prove a property about an expression if and only if the abstract interpretation of the expression entails that property.

THEOREM 4.7.1 (SOUNDNESS AND COMPLETENESS). *Let e be an expression of type σ, and let ρ and Γ be environments of values and formulae, respectively, such that $\rho(x) = [\![\Gamma(x)]\!]$ for all free variables x in e. Then for all $\varphi \in L(\sigma)$,*

$$[\![e]\!]\rho \subseteq [\![\varphi]\!] \iff \Gamma \vdash e : \varphi.$$

PROOF. We first prove that all deductions made in the logic are sound (the implication $\Leftarrow$). This is done by induction on the proof tree for the judgment and amounts to showing soundness of each rule in the logic; see Appendix A.5. Then we prove that the logic is complete, i.e., that it can show all semantic properties of a given term (the implication $\Rightarrow$). Completeness is first shown under the restriction that all free variables are mapped to irreducible properties, and it is then shown how this restriction can be lifted. The proof is by structural induction over e and is given in Appendix A.6. □

5. ANALYSIS OF LIST STRUCTURES

In this section we give examples of properties of integer lists that can be expressed in our logic and calculate the partial order of irreducible properties of integer lists. Recall that we defined integer lists by

$$\mathsf{Int\ List} = \mu\alpha.\mathsf{Int} \times \alpha + 1,$$

where 1 is the unit type $\{\bullet\}$. Since the language is lazy it is possible to add an element, 17 say, to an undefined list $\bot_{\mathsf{Intlist}}$ and obtain the partial list $17 :: \bot$.

Likewise we can construct infinite lists such as the infinite list $1 :: 1 :: 1 :: \ldots$. Lists are finite if they contain an element from the second summand $\mathbf{1}$. The empty list *nil* is the value of $\mathsf{fold}(\mathsf{in}_2(\bullet))$.

5.1 Strictness Properties of Lists

Even though there is only one property of the unit type $\mathbf{1}$, the trivial $\mathtt{t}_1$, this property plays an important role in describing the structure of the list, since the list is finite if and only if there exists an element in the structure satisfying $\mathsf{in}_2(\mathtt{t}_1)$. For integer values we have the two properties $\mathtt{f}_{\mathsf{Int}}$, meaning that the value is undefined, and $\mathtt{t}_{\mathsf{Int}}$, meaning that the value might be defined. We can then form properties such as the following for lists of integers.

— $\mathtt{f}^{\mu\alpha.\mathsf{Int}\times\alpha+1}$: This property is the strongest property of lists. It is only satisfied by the undefined list $\perp_{\mu\alpha.\mathsf{Int}\times\alpha+1}$.

— $(\Diamond\mathsf{in}_1(\mathtt{f}^{\mathsf{Int}})) \wedge (\Box\mathsf{in}_1(\mathtt{t}^{\mathsf{Int}}))$: This describes all the lists that contain an undefined element *and* (since there is no element from 1) are either partial or infinite.

— $\Box(\mathsf{in}_1(\mathtt{f}^{\mathsf{Int}}) \vee \mathsf{in}_2(\mathtt{t}^1))$: This property is satisfied by all finite, infinite or partial lists, all the elements of which are undefined.

— $\Box(\mathsf{in}_1(\mathtt{t}^{\mathsf{Int}}) \vee \mathsf{in}_2(\mathtt{t}^1))$: The weakest property that holds for all lists. It says nothing about the content and nothing about how the list ends.

In the following we omit the type subscripts on the formulae where possible. The normal form of the formula $\varphi = \Box(\mathsf{in}_1(\mathtt{f}) \vee \mathsf{in}_2(\mathtt{t}))$ can be found by the method from the proof of Theorem 2.1.3 by using the fact that φ is equal to the formulae

$$\varphi_1 = \Box(\mathsf{in}_1(\mathtt{f})) \vee (\Box(\mathsf{in}_1(\mathtt{f}) \vee \mathsf{in}_2(\mathtt{t})) \wedge \Diamond\mathsf{in}_2(\mathtt{t}))$$

and

$$\varphi_2 = \Box(\mathsf{in}_2(\mathtt{t})) \vee (\Box(\mathsf{in}_1(\mathtt{f}) \vee \mathsf{in}_2(\mathtt{t})) \wedge \Diamond\mathsf{in}_1(\mathtt{f})).$$

We first find that

$$\varphi = \varphi \wedge \Diamond(\mathsf{in}_1(\mathtt{f}) \vee \mathsf{in}_2(\mathtt{t})) = (\varphi \wedge \Diamond\mathsf{in}_1(\mathtt{f})) \vee (\varphi \wedge \Diamond\mathsf{in}_2(\mathtt{t}))$$

by $\Diamond - \vee$ and distributivity. Substituting φ_1 (resp., φ_2) for the first (resp., the second) occurrence of φ we obtain

$$\begin{gathered}(\Box(\mathsf{in}_1(\mathtt{f})) \vee (\Box(\mathsf{in}_1(\mathtt{f}) \vee \mathsf{in}_2(\mathtt{t})) \wedge \Diamond\mathsf{in}_2(\mathtt{t}))) \wedge \Diamond\mathsf{in}_1(\mathtt{f}) \\ \vee \\ (\Box(\mathsf{in}_2(\mathtt{t})) \vee (\Box(\mathsf{in}_1(\mathtt{f}) \vee \mathsf{in}_2(\mathtt{t})) \wedge \Diamond\mathsf{in}_1(\mathtt{f}))) \wedge \Diamond\mathsf{in}_2(\mathtt{t}).\end{gathered}$$

By applying distributivity again this can be shown to be equal to

$$\Box(\mathsf{in}_1(\mathtt{f})) \wedge \Diamond\mathsf{in}_1(\mathtt{f}) \;\vee\; \Box(\mathsf{in}_2(\mathtt{t})) \wedge \Diamond\mathsf{in}_2(\mathtt{t}) \;\vee\; \Box(\mathsf{in}_1(\mathtt{f}) \vee \mathsf{in}_2(\mathtt{t})) \wedge \Diamond\mathsf{in}_2(\mathtt{t}) \wedge \Diamond\mathsf{in}_1(\mathtt{f})$$

which is in normal form. Informally, the normal form can be seen as an enumeration of the possible ways a list can satisfy φ: by being an infinite or partial and only containing undefined integer elements, by being empty, or by being finite and only containing undefined elements.

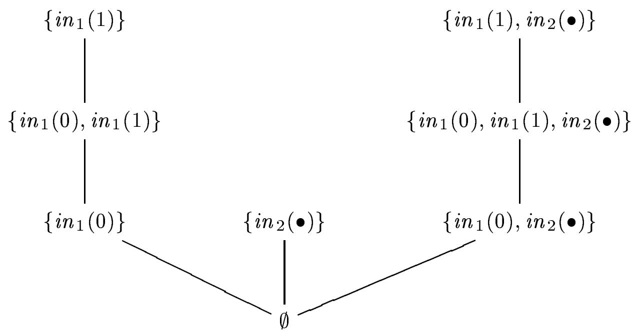

Fig. 6. The partial order $\mathcal{J}(\mu\alpha.\mathsf{Int} \times \alpha + 1) = P_C(\mathbf{2} \cup \{\bullet\})_\emptyset$.

As noticed in Section 4.4 the abstract lattice of integer lists is defined as the lower sets of the partial order

$$\mathcal{J}(\mu\alpha.\mathsf{Int} \times \alpha + 1) = P_C(\mathcal{J}(\mathsf{Int}) \cup \mathcal{J}(1))_\emptyset = P_C(\{in_1(0), in_1(1), in_2(\bullet)\})_\emptyset.$$

The partial order $\mathcal{J}(\mu\alpha.\mathsf{Int} \times \alpha + 1)$ is shown in Figure 6. Each irreducible disjunct in the normal form for φ from above is interpreted as the downward closure of the elements

$$\{in_1(0)\}, \{in_2(\bullet)\}, \text{ and } \{in_1(0), in_2(\bullet)\},$$

respectively. In Section 4.4, the interpretation of the formula φ was found to be the lower set

$$\{\emptyset, \{in_1(0)\}, \{in_2(\bullet)\}, \{in_1(0), in_2(\bullet)\}\}$$

which we recognize as the union of the interpretation of the three disjuncts from above.

We now show as examples of how the logic works some properties of the functions length and map. In our language the function for calculating the length of a list is written

$$\mathsf{length} = \mathsf{fix}\ \lambda L.\lambda l.\mathsf{case\ unfold}\ l\ \mathsf{of\ in}_1(x, xs)\ .\ 1 + L\ xs\ []\ \mathsf{in}_2(y)\ .\ 0,$$

and we prove that taking the length of an infinite list yields an undefined result. The infinite list where nothing is known about the elements is characterized by the formula $\Box(\mathsf{in}_1(\mathtt{t}))$. We want to prove that

$$\mathsf{length} : \Box(\mathsf{in}_1(\mathtt{t})) \to \mathtt{f}.$$

The normal form for $\Box(\mathsf{in}_1(\mathtt{t}))$ is $\Box(\mathsf{in}_1(\mathtt{t})) \wedge \Diamond(\mathsf{in}_1(\mathtt{t}))$, and from the rule $\mathsf{Unfold} - \Box$ we find that unfold satisfies

$$\vdash \mathsf{unfold} : \Box(\mathsf{in}_1(\mathtt{t})) \wedge \Diamond(\mathsf{in}_1(\mathtt{t})) \to \mathsf{in}_1(\mathtt{t} \times \Box(\mathsf{in}_1(\mathtt{t})))$$

which is equivalent to $\Box(\mathsf{in}_1(\mathtt{t})) \to \mathsf{in}_1(\mathtt{t} \times \Box(\mathsf{in}_1(\mathtt{t})))$. With Γ being the environment defined by

$$\Gamma \equiv [L : \Box(\mathsf{in}_1(\mathtt{t})) \to \mathtt{f}, l : \Box(\mathsf{in}_1(\mathtt{t}))]$$

we can prove using the **App** rule that

$$\Gamma \vdash \mathsf{unfold}\ l : \mathsf{in}_1(\mathtt{t} \times \Box(\mathsf{in}_1(\mathtt{t}))).$$

Writing ϕ for $\Box(\mathsf{in}_1(\mathtt{t})) \to \mathtt{f}$, the rest of the proof then looks as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathsf{unfold}\ l : \mathsf{in}_1(\mathtt{t} \times \Box(\mathsf{in}_1(\mathtt{t})))} \qquad \dfrac{\dfrac{\vdots}{\Gamma[x : \mathtt{t}, xs : \Box(\mathsf{in}_1(\mathtt{t}))] \vdash L\ xs : \mathtt{f}}}{\Gamma[x : \mathtt{t}, xs : \Box(\mathsf{in}_1(\mathtt{t}))] \vdash 1 + L\ xs : \mathtt{f}}}{\Gamma \vdash \mathsf{case\ unfold}\ l\ \mathsf{of}\ \mathsf{in}_1(x, xs)\ .\ 1 + L\ xs\ []\ \mathsf{in}_2(y)\ .\ 0 : \mathtt{f}}}{[L : \Box(\mathsf{in}_1(\mathtt{t})) \to \mathtt{f}] \vdash \lambda l.\mathsf{case\ unfold}\ l\ \mathsf{of}\ \mathsf{in}_1(x, xs)\ .\ 1 + L\ xs\ []\ \mathsf{in}_2(y)\ .\ 0 : \phi}}{\vdash \lambda L.\lambda l.\mathsf{case\ unfold}\ l\ \mathsf{of}\ \mathsf{in}_1(x, xs)\ .\ 1 + L\ xs\ []\ \mathsf{in}_2(y)\ .\ 0 : \phi \to \phi}}{\vdash \mathsf{fix}\ \lambda L.\lambda l.\mathsf{case\ unfold}\ l\ \mathsf{of}\ \mathsf{in}_1(x, xs)\ .\ 1 + L\ xs\ []\ \mathsf{in}_2(y)\ .\ 0 : \Box(\mathsf{in}_1(\mathtt{t})) \to \mathtt{f}}$$

where the last four deduction steps were done using the rules **Case** $- \vee$, **Abs**, **Abs**, and **Fix**.

The function map applies another function f to each element of a list l; it is defined as follows:

$$\begin{array}{lll} \mathsf{map} = \mathsf{fix}\ \lambda M.\lambda f.\lambda l.\mathsf{case\ unfold}\ l\ \mathsf{of} & & \mathsf{in}_1(x, xs)\ .\ \mathsf{fold\ in}_1(fx, M\ f\ xs) \\ & [] & \mathsf{in}_2(y)\ .\ \mathsf{fold\ in}_2(\bullet) \end{array}$$

We want to verify that mapping a strict function over a list containing an undefined element gives a list that contains an undefined element. We show that the uniformity of the properties only allows a slightly weaker result to be proved, viz., that the empty list is found to be a possible outcome as well. The set of lists with an undefined element is described by the formula $\Diamond(\mathsf{in}_1(\mathtt{f}))$ and a strict function on the integers by $\mathtt{f} \to \mathtt{f}$. Define the environment Γ by

$$[M : (\mathtt{f} \to \mathtt{f}) \to \Diamond(\mathsf{in}_1(\mathtt{f})) \to \Diamond(\mathsf{in}_1(\mathtt{f})), f : \mathtt{f} \to \mathtt{f}, l : \Diamond(\mathsf{in}_1(\mathtt{f}))].$$

We first determine the normal form of $\Diamond(\mathsf{in}_1(\mathtt{f}))$ to be

$$\begin{array}{c} \Box(\mathsf{in}_1(\mathtt{t}) \vee \mathsf{in}_1(\mathtt{f})) \wedge \Diamond\mathsf{in}_1(\mathtt{t}) \wedge \Diamond\mathsf{in}_1(\mathtt{f}) \\ \vee \\ \Box(\mathsf{in}_1(\mathtt{t}) \vee \mathsf{in}_1(\mathtt{f}) \vee \mathsf{in}_2(\mathtt{t})) \wedge \Diamond\mathsf{in}_1(\mathtt{t}) \wedge \Diamond\mathsf{in}_1(\mathtt{f}) \wedge \Diamond(\mathsf{in}_2(\mathtt{t})). \end{array}$$

Intuitively, two disjuncts arise because we cannot tell whether the list is finite (the second case) or not (first case). Applying the rule **Unfold** $- \Diamond$ to each of the disjuncts we find that unfold l satisfies either

$$\mathsf{in}_1(\mathtt{f} \times \mathtt{t}^{\mathsf{Intlist}}) \vee \mathsf{in}_1(\mathtt{t} \times \Diamond\mathsf{in}_1(\mathtt{f}))$$

or

$$\mathsf{in}_1(\mathtt{f} \times \mathtt{t}^{\mathsf{Intlist}}) \vee \mathsf{in}_1(\mathtt{t} \times \Diamond\mathsf{in}_1(\mathtt{f})) \vee \mathsf{in}_2(\mathtt{t}).$$

We observe that the former implies the latter; hence the result must be described by the weaker $\mathsf{in}_1(\mathtt{f} \times \mathtt{t}^{\mathsf{Intlist}}) \vee \mathsf{in}_1(\mathtt{t} \times \Diamond\mathsf{in}_1(\mathtt{f})) \vee \mathsf{in}_2(\mathtt{t})$. This provides us with three different formulae as input to the case expression; the two first match the first branch; the third matches the second branch. These matchings result in three different extensions to the environment Γ which we denote by Γ_1, Γ_2, and Γ_3.

$$\Gamma_1 = \Gamma[x : \mathtt{f}, xs : \mathtt{t}], \qquad \Gamma_2 = \Gamma[x : \mathtt{t}, xs : \Diamond\mathsf{in}_1(\mathtt{f})], \qquad \Gamma_3 = \Gamma[y : \mathtt{t}].$$

Using the rules **In**, **Pair**, **App**, and **Var** it can be shown that

$$\Gamma_1 \vdash \mathsf{in}_1(fx, M\ f\ xs) : \mathsf{in}_1(\mathtt{f} \times \mathtt{t})$$
$$\Gamma_2 \vdash \mathsf{in}_1(fx, M\ f\ xs) : \mathsf{in}_1(\mathtt{t} \times \Diamond\mathsf{in}_1(\mathtt{f}))$$
$$\Gamma_3 \vdash \mathsf{in}_2(\bullet) : \mathsf{in}_2(\mathtt{t}).$$

Hence, using the rule for fold, we have

$$\Gamma_1 \vdash \mathsf{fold}\ \mathsf{in}_1(fx, M\ f\ xs) : \Diamond\mathsf{in}_1(\mathtt{f})$$
$$\Gamma_2 \vdash \mathsf{fold}\ \mathsf{in}_1(fx, M\ f\ xs) : \Diamond\mathsf{in}_1(\mathtt{f})$$
$$\Gamma_3 \vdash \mathsf{fold}\ \mathsf{in}_2(\bullet) : \Box(\mathsf{in}_2(\mathtt{t})) \wedge \Diamond(\mathsf{in}_2(\mathtt{t})),$$

so by **Case**$-\vee$ the whole case expression satisfies $\Diamond(\mathsf{in}_1(\mathtt{f})) \vee (\Box(\mathsf{in}_2(\mathtt{t})) \wedge \Diamond(\mathsf{in}_2(\mathtt{t})))$ which is weaker than the formula we started out with because of the disjunct $\Box(\mathsf{in}_2(\mathtt{t})) \wedge \Diamond(\mathsf{in}_2(\mathtt{t}))$. This extra disjunct states that the empty list is a possible result which is obviously not the case, since the original list contained at least one element. This imprecision comes from the $\mathsf{Unfold} - \Diamond$ rule that lists the unfolding of the empty list $\mathsf{in}_2(\bullet)$ as one of the possible outcomes. This rule fails to record that we are certain to have obtained an undefined element before we reach the point where we unfold the empty list, but taking this into account would require a rule that distinguishes between the summands that terminate a structure and those that do not. However, as can be seen from the preceding deductions, both of the invariants

(1) an infinite or partial list containing an undefined element is mapped to a list satisfying the same property and

(2) a list that is either empty or contains an undefined element is mapped to a list satisfying the same property

can be deduced using the logic.

5.2 Related Approaches

In his original work on strictness analysis of lists, Wadler [1987] suggested the following four-point domain for the analysis of programs over lists of integers. With each point in the domain we give the set of lists that the point represents.[2]

[2]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
$\perp$	the undefined list

Wadler's $0\in$ can be expressed by the formula $\Box(\mathsf{in}_1(\mathtt{t})) \vee \Diamond(\mathsf{in}_1(\mathtt{f}))$. The $\perp$ property in the four-point domain corresponds to $\mathtt{f}^{\mu\alpha.\mathsf{Int}\times\alpha+1}$ and the $1\in$ property to $\mathtt{t}^{\mu\alpha.\mathsf{Int}\times\alpha+1}$ as noted previously. The property ∞ means that all elements in the structure come from the first summand and can be formalized as $\Box(\mathsf{in}_1(\mathtt{t}_{\mathsf{Int}}))$. Nielson and Nielson [1992; 1994] have suggested a generalization of Wadler's lattices for lists. Their lattice for σ-lists contains, in addition to the two elements $\perp$ and ∞, an element $Y\varepsilon$ for each upward 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."* If all the properties $y \in Y$ are expressible in our logic (i.e., that for each $y \in Y$ there is a formula φ_y such that $[\![\varphi_y]\!] = y$), then the property $Y\epsilon$ is expressible in our system as

$$\Box(\mathsf{in}_1(\mathtt{t})) \vee \bigwedge_{y\in Y} \Diamond\mathsf{in}_1(\varphi_y).$$

More precisely, we obtain the set of properties of Nielson and Nielson as follows.

$$\begin{aligned} \top\varepsilon &\sim \mathtt{t}_{\mu\alpha.\mathsf{Int}\times\alpha+1} \\ Y\varepsilon &\sim \Box(\mathsf{in}_1(\mathtt{t})) \vee \textstyle\bigwedge_{y\in Y} \Diamond\mathsf{in}_1(\varphi_y) \\ 1 &\sim \Box(\mathsf{in}_1(\mathtt{t}_{\mathsf{Int}})) \\ 0 &\sim \mathtt{f}_{\mu\alpha.\mathsf{Int}\times\alpha+1} \end{aligned}$$

The construction of the property $Y\varepsilon$ works for arbitrary set Y, but the logical content of $Y\varepsilon$ for a set Y and for its upward closure is the same. This is clear from the logical description of the properties, since if $\varphi \le \psi$, then $\Diamond\varphi = \Diamond\varphi \wedge \Diamond\psi$; if there exists an element satisfying φ, then there exists also an element satisfying every ψ weaker than φ. Thus our lattice contains the elements of 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 by Nielson and Nielson [1992; 1994].

A drawback of our construction is that it can result in lattices where different elements denote the same property in the concrete semantics. Ernoult and Mycroft [1991] point out that in strictness analysis, where properties are taken to be Scott-closed sets, the two properties $\Box(\mathsf{in}_1(\mathtt{t}))$ and $\Box\mathsf{in}_1(\mathtt{t}) \wedge \Diamond\mathsf{in}_1(\mathtt{f})$ represent the same Scott-closed set of lists. Redundancy among lattice elements is not unique to this construction and does not jeopardize the correctness of the analysis. Cousot and Cousot [1979] discuss this phenomenon in a general context and devise a method for obtaining a lattice without duplicate elements.

5.3 Other List Properties

The $\Box$- and $\Diamond$-properties are parameterized on the set of properties describing the element type and are thus applicable to a range of analyses. The data-flow properties listed in Section 2.2 extend to yield invariance properties such as *"all elements of the list are positive"* expressed using the $\Box$-formulae. These kinds of properties are the ones most frequently encountered, but we remark that the $\Diamond$-properties of lists are useful for analyzing functions that combine all elements of a list using a binary operator,[3] for example they allow us to prove properties such as multiplying the elements of a list containing a zero yields zero. This simple fact is not provable only using $\Box$-properties. Incidentally, expressing the opposite property of containing a zero is also possible using the formula $\Box(\mathsf{in}_1(\mathtt{pos}) \vee \mathsf{in}_1(\mathtt{neg}) \vee \mathsf{in}_2(\mathtt{t}))$, but negation of properties is in general not expressible.

The formulae in $L(\mathsf{Int\ List})$ also have an interpretation as binding-time properties. As mentioned in Section 2.2, the formula $\mathtt{f}$ denotes the undefined list and $\mathtt{t}$ the completely dynamic list. Between these we find properties such as

$$\Box(\mathsf{in}_1(\mathtt{f}) \vee \mathsf{in}_2(\mathtt{t}))$$

which characterize all those lists (empty, finite, infinite, or partial) whose elements are static; that is, this is the "static" predicate for integer lists. Lists whose structure is determined, but where the elements are dynamic, are similarly described by the property

$$\Box(\mathsf{in}_1(\mathtt{t}) \vee \mathsf{in}_2(\mathtt{t}))$$

which we, following Hunt and Sands [1991], call SPINE-D. Satisfying this property is enough for a list to have its length (possibly infinite) determined. A formal proof that the function length defined previously satisfies the property SPINE-D $\to \mathtt{f}$ is constructed by proving the subresults

$$\Gamma \vdash \mathsf{unfold}\ l : \mathsf{in}_1(\mathtt{t}) \times \text{SPINE-D} \vee \mathsf{in}_2(\mathtt{t}) \times \text{SPINE-D}$$
$$\Gamma[x : \mathtt{t}, xs : \text{SPINE-D}] \vdash 1 + L\ xs : \mathtt{f}$$
$$\Gamma[y : \mathtt{t}] \vdash 0 : \mathtt{f}$$

and then proceed as in the proof for length in Section 5.1. A comparison with other sets of properties proposed in the literature appears in Section 6.1.

6. RELATED WORK

Jones and Muchnick [1981] distinguish between two data-flow analyses for type inference of programs over a tuple of variables. One analysis (called independent attribute) describes a state as a tuple of sets of types; the other (the relational) describes it as a set of tuples. They show that the relational analysis is more precise, and they argue that this extra precision is necessary for several data-flow problems. They also prove that the analysis is inherently more complex than the independent attribute method. For programs where the state is an n-tuple of

[3]These functions are usually called fold, foldr, etc. and should not be confused with our fold.

integers, this distinction amounts to using elements from either $\wp(Z) \times \ldots \times \wp(Z)$ or $\wp(Z \times \ldots \times Z)$. Generalizing this to other lattices of properties amounts to constructing a lattice for modeling products from the lattices S_i modelling the i-th component. Nielson [1984; 1985] observed that such an operator should at least satisfy

$$\wp(S_1 \times \ldots \times S_n) \cong \wp(S_1) \star \ldots \star \wp(S_n).$$

and argued that the tensor product of lattices is the right operation for the generalization. The tensor product $A \otimes B$ of two lattices can be constructed in several ways. One possibility is to construct it as a term model of a theory where the elements are terms

$$(a_1 \otimes b_1) \sqcup \ldots \sqcup (a_n \otimes b_n),$$

where $a_i \in A$ and $b_i \in B$. The theory has equations for identifying terms that represent the same information, in particular,

$$(a \otimes b_1) \sqcup (a \otimes b_2) = a \otimes (b_1 \sqcup b_2)$$

and

$$(a_1 \otimes b) \sqcup (a_2 \otimes b) = (a_1 \sqcup a_2) \otimes b$$

that correspond to our axioms $(\vee - \times)$ and $(\times - \vee)$. Our axiomatization thus integrates these "tensor axioms" into a general strictness logic.

Other, more direct constructions of the tensor product of lattices have been considered. A construction based on downward closed sets has been considered by Nielson and Nielson [1992; 1994] and Burn [1992]. Both papers use the following definition of tensor product of finite lattices:

$$L_1 \otimes L_2 = (\{Y \subseteq L_1 \times L_2 \mid Y = LC(Y) = CC_1(Y) = CC_2(Y)\}, \subseteq),$$

where

$$\begin{aligned} LC(Y) &= \downarrow Y(\text{ the downward closure of } Y) \\ CC_1(Y) &= \{(l_1 \sqcup l_2, l) \mid (l_1, l), (l_2, l) \in Y\} \\ CC_2(Y) &= \{(l, l_1 \sqcup l_2) \mid (l, l_1), (l, l_2) \in Y\}. \end{aligned}$$

A term $a_1 \otimes b_1$ is modeled by the lower set $\downarrow\{(a_1, b_1)\}$. The closure conditions CC_1 and CC_2 are there to remove lower sets of the form $\downarrow\{(a, b_1), (a, b_2)\}$, since they represent the same information as $\downarrow\{(a, b_1 \sqcup b_2)\}$. In other words, this model chooses to represent the equivalence class containing $(a \otimes b_1) \sqcup (a \otimes b_2)$ and $a \otimes (b_1 \sqcup b_2)$ by $a \otimes (b_1 \sqcup b_2)$.

In Section 4.2 we defined

$$[\![\sigma \times \tau]\!] = \mathsf{D}(\{(a, b) \mid \downarrow\{a\} \in [\![\sigma]\!], \downarrow\{b\} \in [\![\tau]\!]\})$$

and argued formally that $[\![\sigma \times \tau]\!] \cong [\![\sigma]\!] \otimes [\![\tau]\!]$ (this could also have been shown by appealing to a theorem of Bandelt [1980, Theorem 1.2.]). If $[\![e_1]\!]\rho = \downarrow\{a\}$ and $[\![e_2]\!]\rho = \downarrow\{b_1, b_2\}$, then $[\![(e_1, e_2)]\!]\rho = \downarrow\{(a, b_1), (a, b_2)\}$, which is the least upper bound of $\downarrow\{(a, b_1)\}$ and $\downarrow\{(a, b_2)\}$ in $[\![\sigma \times \tau]\!]$. In terms of tensor products, our interpretation of products can be seen as taking the term $a \otimes b_1 \sqcup a \otimes b_2$ as representative. Our construction has the virtue of being simple; for example, it avoids the additional preceding closure conditions CC_1 and CC_2. On the other hand it is tailored to our approach here and is restricted to lattices constructed as downward closed sets over finite partial orders. The construction used by Nielson and Nielson [1992; 1994] and Burn [1992] works for arbitrary lattices.

Nielson and Nielson [1994] only treat operations on data structures and do not address how this integrates with analyses for other language constructs. Burn [1992] incorporates tensor products into the framework of abstract interpretation of higher-order functions [Burn et al. 1986]. It is not as powerful as the analysis developed here because functions are still interpreted as monotone functions between lattices. This means that certain disjunctive properties modeled by our use of lower sets of monotone functions cannot be modeled. For example consider the two functions

$$f_1 \equiv \lambda x^{\mathsf{Int}}.\lambda y^{\mathsf{Int}}.x$$
$$f_2 \equiv \lambda x^{\mathsf{Int}}.\lambda y^{\mathsf{Int}}.y.$$

Because least upper bounds of functions are computed pointwise when using monotone functions, the abstract interpretation of an expression e that either evaluates to f_1 or f_2 is the function $\lambda x^{\mathsf{Int}}.\lambda y^{\mathsf{Int}}.x \sqcup y$. However, if the result of e is applied to two arguments, at least one of which is undefined, one of the functions will return undefined. Thus with the lower set interpretation we detect the strictness of the function resulting from

$$(\lambda f.\lambda z.(f\ 17\ z) + (f\ z\ 17))\ e,$$

a property we cannot detect if we have to describe e by the function $\lambda x^{\mathsf{Int}}.\lambda y^{\mathsf{Int}}.x \sqcup y$.

Benton [1992a] develops a disjunctive strictness logic and shows it to be correct with respect to the standard semantics of the language. Except for recursive data structures, his logic is similar to ours, but has a different notion of irreducibility that allows an extra rule for function properties, viz.,

$$\varphi \rightarrow (\psi_1 \vee \psi_2) \leq (\varphi \rightarrow \psi_1) \vee (\varphi \rightarrow \psi_2),$$

provided that φ satisfies his notion of irreducibility. This extra rule makes the logic more discerning, but it has as a consequence that it becomes difficult to define a normal form for formulae of function type. Benton is not concerned with the relationship to an abstract interpretation, but the importance of normal forms for relating logics and interpretation suggests that a corresponding abstract interpretation would be difficult to find.

A disjunctive strictness analysis was obtained by Muylaert-Filho [1994] who shows that the analysis presented in this article is equivalent to the analysis obtained by first converting a program into continuation-passing style using (a version of) Reynolds' call-by-value transformation [Reynolds 1974] and then using the abstract interpretation of Burn et al. [1986] on the transformed program. Since this abstract interpretation corresponds to a program logic only involving conjunctive properties [Jensen 1991; 1995] this provides an alternative method for adding disjunctions to an abstract interpretation.

An earlier result of Nielson [1982] concerning data-flow analysis for flow charts states that analyses formulated as continuation-style denotational semantics yields the meet-over-all-paths (MOP) solution [Hecht 1977] whereas a direct-style formulation leads to the maximal fixed-point solution (MFP). In terms of data-flow analysis, the developments presented in this article allow us to find a MOP solution, by enriching the domains of the original analysis, and then calculate an MFP solution in these new domains. Informally, this can be explained by the observation that the disjunctions allow us to keep the intermediate results of subpaths apart when two execution paths join during analysis.

The use of the lower set completion of partial orders in abstract interpretation was proposed by Cousot and Cousot [1979], and variations thereof have been used by, among others, Deutsch [1992] for alias analysis and Filé and Ranzato [1994] for enriching domains used in analysis of logic programs. Filé and Ranzato propose lifting abstract domains to their powerset and then equating those sets of properties where the least upper bound of their concretizations are equal. This means that the actual abstract domain is defined using the concretization function from the Galois connection defining the relation between abstract and concrete domains and must therefore be recalculated for every new instantiation. This is acceptable for analyses that only operate with one abstract domain (as the example from logic programming given by Filé and Ranzato [1994]), but becomes awkward when dealing with a rich type structure where each type has its own abstract domain. A more serious limitation of the construction is that the concrete domain must be a lattice. This is central to the definition of the abstract domain which requires that arbitrary least upper bounds in the concrete domain exist and prohibits its application to most denotational semantics where the semantic domains are complete partial orders rather than lattices.

Cousot and Cousot [1994] use the lower set construction to add disjunctive information to their basic comportment analysis (an analysis combining strictness and termination properties). Apart from the termination properties, their analysis can express more properties of functions than the one developed here, since relations among free variables of a term can be represented. This is obtained by interpreting an expression as a lower set of environment-result pairs whereas we interpret an expression as a mapping from an environment of lower sets to the lower set representing the result.

6.1 Algebraic Types

The analysis of recursive structures (lists, in particular) has been considered by a number of people [Benton 1993; Burn 1991; Ernoult and Mycroft 1991; Ferguson and Hughes 1989; Hunt 1991; Hunt and Sands 1991; Jensen 1992b; Launchbury 1989; Leung and Mishra 1991; Nielson and Nielson 1992; Wadler 1987] The solution proposed here is by no means the only possible, but has been chosen, since it provides a generalization of some of the afore-mentioned analyses. The ones that we cannot deal with are the analyses relying on structural information such as *"the first element of the list is undefined."* This includes in particular Wadler and Hughes' head-strictness [Wadler and Hughes 1987] and Burn's H_B-strictness [Burn 1991].

In Section 5.2 we commented on the relation to the work of Wadler [1987] and Nielson and Nielson [1992; 1994]. Ernoult and Mycroft [1991] introduced uniform ideals of domains to model uniform strictness properties of lists and were the first to use the modal formulae to express them, although they did not provide an axiomatization. The use of $\Diamond$-properties for expressing strictness properties has its origin here. There is a small difference in notation, since their property $\Box\varphi$ means a finite, infinite, or partial list with all elements satisfying φ and hence corresponds to our $\Box(\mathsf{in}_1(\varphi) \vee \mathsf{in}_2(\mathtt{t}))$. To express that the list is partial or infinite they write $\Box_\perp\varphi$ which corresponds to our $\Box\mathsf{in}_1(\varphi)$.

The set of binding-time properties arising from our axiomatization extends earlier domains used for binding-time analysis based on projections and partial equivalence relations [Hunt and Sands 1991; Jones et al. 1993; Launchbury 1989; Mogensen 1988]. Products are modeled as products of binding-time properties and hence do not include our disjunctive properties. Jones et al. [1993] deal with sums by listing one binding-time property for each summand; a more general treatment is given by Launchbury [1989] who employs a list of (tag,property)-pairs that does not necessarily contain all possible tags. The information in such a list corresponds to our disjunction

$$\bigvee_{i \in J} \mathsf{in}_j(\varphi_i),$$

where J is the set of tags present in the list, and φ_i is the property associated with tag in_i in the list. Our framework adds the property *"undefined"* to the set of properties; this is absent in the domains previously cited. Furthermore, they all use a three-point lattice for describing list structures. Using the names from Hunt and Sands [1991], we see that the top element D is the completely dynamic list and thus corresponds to our $\mathtt{t}$. Below is SPINE(D) which is satisfied by all lists whose structure can be determined but whose elements are dynamic; it is equivalent to our $\Box(\mathsf{in}_1(\mathtt{t}) \vee \mathsf{in}_2(\mathtt{t}))$. The smallest element, SPINE(S) denotes lists where all elements are static which is our $\Box(\mathsf{in}_1(\mathtt{f}) \vee \mathsf{in}_2(\mathtt{t}))$. The framework proposed in this article adds properties that allow us to state that a list is empty or that it is infinite or partial, in addition to the property of being the undefined list.

Our technique for constructing the abstract domain in the abstract interpretation in Section 4 included the use of partial orders of convex sets. These sets usually pose a problem in abstract interpretation, since two convex sets do not necessarily have a least upper bound in the Egli-Milner ordering as noted already by Plotkin [1976]. One solution is to embed the convex sets into a lattice via Birkhoff completion as we have done in this article. Another solution is presented by Ferguson and Hughes [1989] who consider a particular collection of convex sets: the cone powerdomain. They define a cone to be a convex set that is closed under least upper bound and show that least upper bounds of cones exist. In their comparison of using cones rather than arbitrary convex sets, Ferguson and Hughes argue that for their application "...we do not really care about the distinction between sets ...differing only by points which are lubs of other points in the set." In general this is only sustainable when there is no loss of information by taking least upper bounds. For example, we lose precision if we have to convert the set $\{(1,0),(0,1)\}$ into the set $\{(1,0),(0,1),(1,1)\}$, since we no longer know that at least one component will be undefined. On the other hand, if $\{(1,0),(0,1)\}$ is converted into $\{(1,0),(0,1),(1,0) \sqcup (0,1)\}$ where the least upper bound operator allows us to retrieve $\{(1,0),(0,1)\}$, then the cone is just another representation of the same information. For a finite partial order P we have that the cones of $\mathsf{D}(P)$ are order-isomorphic to the convex sets of P, so we could have cast our semantics in terms of cones.

A different technique for constructing abstract lattices for recursive data structures has been proposed by de Lavalette [1992] who constructs abstract domains for recursive types by solving recursive domain equations in a category of lattices of abstract properties. The obtained lattices are still infinite, and further abstraction or suitable widening operators [Cousot and Cousot 1992] are therefore necessary to obtain a static analysis. Benton [1993] has taken this idea further and has shown how a large number of uniform and structural properties can be characterized by using the initiality of the solutions to recursive domain equations. Because the semantics of fold and unfold arise directly from initiality, this provides an immediate definition of their behavior on these properties. Even though initiality does not provide conjunctions and disjunctions or suggest an axiomatization of these properties, it seems a very promising approach to linking the abstract properties with properties in the standard semantics and is the only one developed for general algebraic data structures.

The idea of uniform properties has been taken further by Le Métayer [1995] who extends the set of properties of recursive structures with "binary" properties, written ∇R, expressing that a particular relation R holds between an element of the structure and any of the elements in the substructure(s) following that element. For example, an integer list sorted in increasing order satisfies the predicate $\nabla \mathit{leq}$ where leq is the less than or equal predicate on integers. By considering a reduced set of language features (e.g., only first-order, unary functions are allowed), Le Métayer obtains a fully automated procedure for finding preconditions necessary for a given postcondition to hold after execution of a program.

7. CONCLUSIONS

We have presented an axiomatization of disjunctive program properties and a logic for reasoning about such properties in the setting of a higher-order functional language extended with recursive data structures. It has been argued that this general logic can be instantiated to yield binding-time, data-flow, and strictness analysis for this language. We have then constructed an abstract interpretation over domains containing the axiomatized strictness properties and have shown that the abstract interpretation of a program is the strongest property provable of a program in the logic. This equivalence has several consequences. On one hand, the abstract interpretation provides an effective method for calculating the properties provable of a program in the logic. On the other hand, the logic provides a characterization of the use of tensor products in abstract interpretation, substantiating the claim that tensor products represent disjunctive program properties. Furthermore, our axiomatization of recursive data structures based on uniform properties brings out the logical content of the elements in lattices for analyzing lists already present in the literature and demonstrates how these can be integrated into a systematic framework for constructing abstract lattices for algebraic data structures.

We see at least two directions in which our work can be extended. First, the relation to the standard semantics should be defined rigorously through a concretization relation between the lattice of abstract properties and the properties of the concrete semantics. The concretization relation set up by Benton [1992a] covers most of the types and properties considered here, but does not deal with uniform properties. As mentioned above, Benton's work on characterizing properties of recursive data structures via initiality seems likely to be applicable here. Second, the size of the lattices constructed by our method means that it would be very space-consuming to tabulate the functions arising from the abstract interpretation; hence a conventional implementation based on fixed point iteration will often be infeasible. It should be pointed out, though, that there exist methods for finding parts of a fixed point; in particular, one is not obliged to calculate the abstract interpretation of a function in its entirety just to find its value at a particular argument [Jones and Mycroft 1986; Rosendahl 1993]. These methods have given good results in difficult cases [Jensen et al. 1994], so it is conceivable that they could apply here as well. In another direction it would be desirable to extend the work by Monsuez [1995] and Hankin and Le Métayer [1994; 1995] on implementing analyses formulated as type systems to cover disjunctions. The work on union types by Barbanera et al. [1995] and the constraint-based inference algorithms developed by Aiken et al. [1994] for finding union types of functional programs should provide a starting point. Whether an iteration-based or inference-based strategy is adopted, some notion of normal form of properties will be needed in order to compare these, emphasizing the importance of our results about normal forms and lattice models of properties.

APPENDIX

A.1 Normal Forms of Formulae (Theorem 2.1.3)

THEOREM. *For all $\varphi \in L(\sigma)$ there exist $\psi_1, \dots \psi_n \in \mathsf{Irr}(\sigma)$ such that*

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

PROOF. The proof follows by structural induction over the formation rules for formulae. The formulae $\mathtt{f}^\sigma$ are already in this form for all types σ. An induction over σ shows that each of the $\mathtt{t}^\sigma$ can be rewritten to this form—the only complication arises with sums where the axiom $(+ - \mathtt{t})$ gives the desired reduction. Now assume inductively that $\varphi = \bigvee_i \varphi_i$ and $\psi = \bigvee_j \psi_j$. The formula $\varphi \vee \psi$ is already in disjunctive normal form, and the formula $\varphi \wedge \psi$ can be converted into a disjunction of conjunctions of irreducibles by repeated use of the distributive rule. Lemma 2.1.2 then gives that each of these conjunctions is equivalent to a disjunction of irreducibles. For formulae describing products we note that

$$\bigvee_{i \in I} \varphi_i \times \bigvee_{j \in J} \psi_j = \bigvee_{(i,j) \in I \times J} (\varphi_i \times \psi_j),$$

and for formulae of function type we have that

$$(\bigvee_i \varphi_i) \to (\bigvee_j \psi_j)$$

can be rewritten into

$$\bigwedge_i (\varphi_i \to \bigvee_j \psi_j)$$

which satisfies the Irr predicate. For sums we just observe that $\mathsf{in}_j(\bigvee_{i\in I}\varphi_i) = \bigvee_{i\in I}\mathsf{in}_j(\varphi_i)$. The argument for multiset properties follows that of Abramsky [1991b, Proposition 3.4.8], so we only give a sketch of the proof here. By distributivity we can bring any formula from $L(\mu\alpha.\sum_{i\in I}\sigma_i\times\alpha^{n_i})$ into the form

$$\bigvee(\bigwedge \Box(\chi')\wedge\bigwedge\Diamond\chi''),$$

where χ' and χ'' belong to $L(\sum_{i\in I}\sigma_i)$. By $\Box-\wedge$ this can be transformed into

$$\bigvee(\Box(\bigwedge\chi')\wedge\bigwedge\Diamond\chi''),$$

and by the Normal Form Theorem for the underlying types there exist irreducibles φ,ψ such that this is equivalent to

$$\bigvee(\Box(\bigvee\varphi)\wedge\bigwedge\Diamond\bigvee\psi).$$

Using $\Diamond-\vee$ and distributivity we obtain a formula of the form

$$\bigvee(\Box(\bigvee\varphi)\wedge\bigwedge\Diamond\psi),$$

and Lemma 2.1.1 applied to this formula gives a formula of the desired normal form. □

A.2 Definability (Theorem 4.5.2)

THEOREM. *For every* $\downarrow\{a\}\in[\![\sigma]\!]$ *there exists a* $\varphi_a\in\mathcal{L}(\sigma)$ *such that*

$$\mathsf{Irr}(\varphi_a)\quad\textit{and}\quad\downarrow\{a\}=[\![\varphi_a]\!]_\sigma.$$

PROOF. The proof is by induction over the structure of σ. For the base case $\sigma=\mathsf{Int}$ and $\sigma=\mathsf{Bool}$ we have that

$$\{0\}=[\![\mathtt{f}]\!]\qquad\text{and}\qquad\{0,1\}=[\![\mathtt{t}]\!].$$

For product type $\sigma\times\tau$ the downward closures are of the form

$$\downarrow\{(a,b)\}$$

with $a\in\mathcal{J}(\sigma)$ and $b\in\mathcal{J}(\tau)$. By induction hypothesis we then have that there exist irreducibles φ and ψ such that

$$\downarrow\{a\}=[\![\varphi]\!]\qquad\text{and}\qquad\downarrow\{b\}=[\![\psi]\!],$$

and we then have from the definition of the semantics of product formulae that

$$[\![\varphi\times\psi]\!]=\{(a,b)\mid a\in\downarrow\{a\},b\in\downarrow\{b\}\}=\downarrow\{(a,b)\}.$$

For a function type $[\![\sigma\to\tau]\!]$ we consider downward closures of the form $\downarrow\{f\}$ with f a linear map from $[\![\sigma]\!]$ to $[\![\tau]\!]$. A linear function g is smaller than f if and only if it is smaller than f on all downward closures of single elements; hence

$$\downarrow\{f\}=\{g\in[\![\sigma]\!]\multimap[\![\tau]\!]\mid g\sqsubseteq f\}=\bigcap_{\downarrow\{a\}\in[\![\sigma]\!]}\{g\in[\![\sigma]\!]\multimap[\![\tau]\!]\mid g(\downarrow\{a\})\subseteq f(\downarrow\{a\})\}.$$

To every $\downarrow\{a\}$ there exists an irreducible φ_a such that $\downarrow\{a\}=[\![\varphi_a]\!]$. The lower set $f(\downarrow\{a\})$ can be written as a union of downward closures of single elements belonging to $\mathcal{J}(\tau)$, and the induction hypothesis can therefore be applied to obtain a disjunction of irreducibles, whose interpretation is the lower set $f(\downarrow\{a\})$. We denote this disjunction by ψ_{fa}. The preceding equation then yields that

$$\downarrow\{f\} = [\![\bigwedge_{a\in\mathcal{J}(\sigma)} \varphi_a \to \psi_{fa}]\!],$$

and the latter formula is irreducible since the φ_a are. For sums, an element of $\mathcal{J}(\sigma_1+\sigma_2)$ is either $\bot_{\sigma_1+\sigma_2}$ or of the form $\mathsf{in}_i(a)$ with $a \in \mathcal{J}(\sigma_i)$. In the first case we have $\downarrow\{\bot_{\sigma_1+\sigma_2}\} = [\![\mathtt{f}^{\sigma_1+\sigma_2}]\!]$; in the second case we can by hypothesis find a $\varphi_a \in L(\sigma_i)$ such that $\mathsf{Irr}(\varphi_a)$ and $[\![\varphi_a]\!] = \downarrow\{a\}$. Therefore,

$$\downarrow\{\mathsf{in}_i(a)\} = \{\mathsf{in}_i(d) : d \in \downarrow\{a\}\} \cup \{\bot_{\sigma_1+\sigma_2}\} = [\![\mathsf{in}_i(\varphi_a)]\!].$$

Finally, for $\mathcal{J}(\mu\alpha.\sum_{i=1}^n \sigma_i \times \alpha^{n_i}) = P_C(\bigcup_{i=1}^n \mathcal{J}(\sigma_i))_\emptyset$ we have $[\![\mathtt{f}]\!] = \{\emptyset\}$ and for a convex set $C \subseteq \bigcup_{i=1}^n \mathcal{J}(\sigma_i)$ we can find for each $c \in C$ (necessarily of the form $\mathsf{in}_i(a)$ with $a \in \mathcal{J}(\sigma_i)$) a formula φ_c such that $\mathsf{Irr}(\varphi_c)$ and $\downarrow\{c\} = [\![\varphi_c]\!]$. Lemma 4.4.5 then gives that $[\![\Box(\bigvee_{c\in C}\varphi_c) \wedge \bigwedge_{c\in C} \Diamond\, \varphi_c]\!] = \downarrow\{C\}$. □

A.3 Soundness of Axiomatization (Theorem 4.5.3)

THEOREM. *For all elements $\varphi, \psi \in \mathcal{L}(\sigma)$*

$$\varphi \le \psi \quad \Rightarrow \quad [\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma.$$

PROOF. By checking the validity of each axiom and inference rule, we validate a selection of the axioms here.

- $[\![\mathtt{t}^\sigma \to \mathtt{t}^\tau]\!] = \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f(\mathcal{J}(\sigma)) \subseteq \mathcal{J}(\tau)\} = [\![\sigma]\!] \multimap [\![\tau]\!] = [\![\mathtt{t}^{\sigma\to\tau}]\!]$.

- $[\![\mathtt{t}^\sigma \to \mathtt{f}^\tau]\!] = \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f(\mathcal{J}(\sigma)) \subseteq \{\bot_{\mathcal{J}(\tau)}\}\}$
$= \{\lambda x.\{\bot_{\mathcal{J}(\tau)}\}\} = [\![\mathtt{f}^{\sigma\to\tau}]\!]$.

- $[\![(\varphi_1 \times \psi) \vee (\varphi_2 \times \psi)]\!]$
$= \{(a,b) \mid a \in [\![\varphi_1]\!], b \in [\![\psi]\!]\} \cup \{(a,b) \mid a \in [\![\varphi_2]\!], b \in [\![\psi]\!]\}$
$= \{(a,b) \mid a \in [\![\varphi_1]\!] \cup [\![\varphi_2]\!], b \in [\![\psi]\!]\} = \{(a,b) \mid a \in [\![\varphi_1 \vee \varphi_2]\!], b \in [\![\psi]\!]\}$
$= [\![(\varphi_1 \vee \varphi_2) \times \psi]\!]$.

- $[\![(\varphi_1 \times \psi_1) \wedge (\varphi_2 \times \psi_2)]\!]$
$= \{(a,b) \mid a \in [\![\varphi_1]\!], b \in [\![\psi_1]\!]\} \cap \{(a,b) \mid a \in [\![\varphi_2]\!], b \in [\![\psi_2]\!]\}$
$= \{(a,b) \mid a \in [\![\varphi_1]\!] \cap [\![\varphi_2]\!], b \in [\![\psi_1]\!] \cap [\![\psi_2]\!]\} = [\![\varphi_1 \wedge \varphi_2 \times \psi_1 \wedge \psi_2]\!]$.

- $[\![(\varphi \to \psi_1) \wedge (\varphi \to \psi_2)]\!]$
$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]) \subseteq [\![\psi_1]\!]\} \ \cap \ \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]) \subseteq [\![\psi_2]\!]\}$
$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]) \subseteq [\![\psi_1]\!] \cap [\![\psi_2]\!]\}$
$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]) \subseteq [\![\psi_1 \wedge \psi_2]\!]\} = [\![\varphi \to (\psi_1 \wedge \psi_2)]\!]$.

- $[\![(\varphi_1 \to \psi) \wedge (\varphi_2 \to \psi)]\!]$
$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi_1]\!]) \subseteq [\![\psi]\!]\} \ \cap \ \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi_2]\!]) \subseteq [\![\psi]\!]\}$
$= \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi_1]\!] \cup [\![\varphi_2]\!]) = f([\![\varphi_1]\!]) \cup f([\![\varphi_2]\!]) \subseteq [\![\psi]\!]\}$
$= [\![(\varphi_1 \vee \varphi_2) \to \psi]\!]$.

- If a set $C \subseteq S_1 \cup S_2$ then either $C \subseteq S_1$, or there is a $c \in S_2$ hence;
$[\![\Box(\varphi \vee \psi)]\!] = \{C : C \subseteq [\![\varphi]\!] \cup [\![\psi]\!]\}$
$\subseteq \ \{C : C \subseteq [\![\varphi]\!]\} \cup \{C : C \cap [\![\psi]\!] \text{ is nonempty}\} \cup \{\emptyset\}$
$= [\![\Box\varphi \vee \Diamond\psi]\!]$.

- $[\![\Diamond \mathtt{f}]\!] = \{C \in P_C(\bigcup_{i=1}^n [\![\sigma_i]\!]) : C \cap [\![\mathtt{f}_{\sigma_1 + \ldots + \sigma_n}]\!] \text{ is nonempty}\} \cup \{\emptyset\}$
$= \{C \in P_C(\bigcup_{i=1}^n [\![\sigma_i]\!]) : C \cap \{\bot_{\sigma_1 + \ldots + \sigma_n}\} \text{ is nonempty}\} \cup \{\emptyset\} = \{\emptyset\} = [\![\mathtt{f}]\!]$. □

A.4 Completeness of Axiomatization (Theorem 4.5.4)

THEOREM. *For all elements $\varphi, \psi \in \mathcal{L}(\sigma)$,*

$$[\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma \quad \Rightarrow \quad \varphi \leq \psi.$$

PROOF. The strategy of the proof is to proceed by induction over the structure of σ. From the assumption that the theorem holds for all the constituent types of σ we prove that it holds for irreducible formulae of type σ. Given that we have proved completeness for irreducible formulae we can prove completeness for all formulae as follows: we know that for arbitrary φ, ψ there exist φ_i, ψ_j satisfying $\mathsf{Irr}(\varphi_i)$ and $\mathsf{Irr}(\psi_j)$ such that

$$\varphi = \bigvee \varphi_i \qquad \psi = \bigvee \psi_j.$$

For each φ_i and ψ_j there exist a_i, b_j such that

$$[\![\varphi_i]\!]_\sigma = {\downarrow}\{a_i\} \qquad [\![\psi_j]\!]_\sigma = {\downarrow}\{b_j\}.$$

Completeness can then be proved:

$$\begin{aligned}
[\![\varphi]\!]_\sigma \subseteq [\![\psi]\!]_\sigma &\Rightarrow \cup_i [\![\varphi_i]\!]_\sigma \subseteq \cup_j [\![\psi_j]\!]_\sigma \\
&\Rightarrow \cup_i {\downarrow}\{a_i\} \subseteq \cup_j {\downarrow}\{b_j\} \\
&\Rightarrow \forall i\ \exists j. {\downarrow}\{a_i\} \subseteq {\downarrow}\{b_j\} \\
&\Rightarrow \forall i\ \exists j. \varphi_i \leq \psi_j \\
&\Rightarrow \varphi = \bigvee \varphi_i \leq \bigvee \psi_j = \psi
\end{aligned}$$

It remains to prove completeness for irreducibles. For base types Int and Bool every irreducible is a conjunction of $\mathtt{t}$ and $\mathtt{f}$ and therefore provably equal to either $\mathtt{t}$ or $\mathtt{f}$. The only nontrivial inclusion $[\![\mathtt{f}]\!] = \{0\} \subseteq \{0,1\} = [\![\mathtt{t}]\!]$ is proved by the fact $\mathtt{f} \leq \mathtt{t}$. For a product type $\sigma \times \tau$ the proof is simple:

$$\begin{aligned}
[\![\varphi_1 \times \psi_1]\!] \subseteq [\![\varphi_2 \times \psi_2]\!] &\Rightarrow [\![\varphi_1]\!] \subseteq [\![\varphi_2]\!] \quad \text{and} \quad [\![\psi_1]\!] \subseteq [\![\psi_2]\!] \\
&\Rightarrow \varphi_1 \leq \varphi_2 \quad \text{and} \quad \psi_1 \leq \psi_2 \\
&\Rightarrow \varphi_1 \times \psi_1 \leq \varphi_2 \times \psi_2
\end{aligned}$$

For sums, we have

$$[\![\mathsf{in}_i(\varphi_i)]\!] = \{\mathsf{in}_i(d) : d \in [\![\varphi]\!]\} \cup \{\bot_{\sigma_1 + \ldots + \sigma_n}\}]\!];$$

so

$$[\![\mathsf{in}_i(\varphi_i)]\!] \subseteq [\![\mathsf{in}_j(\psi_j)]\!] \ \Rightarrow\ i = j \text{ and } [\![\varphi]\!] \subseteq [\![\psi]\!];$$

hence, by induction hypothesis $\varphi \leq \psi$; so $\mathsf{in}_i(\varphi_i) \leq \mathsf{in}_j(\psi_j)$. For function type $\sigma \to \tau$ we have as assumption that

$$[\![\bigwedge_{i \in I} \varphi_i \to \psi_i]\!] \subseteq [\![\bigwedge_{j \in J} \varphi_j \to \psi_j]\!]$$

with $\mathsf{Irr}(\varphi_i), \mathsf{Irr}(\varphi_j)$ which implies that

$$\forall j \in J\ .\ [\![\bigwedge_{i \in I} \varphi_i \to \psi_i]\!] \subseteq [\![\varphi_j \to \psi_j]\!].$$

If we can prove that for fixed $j \in J$

$$[\![\bigwedge_{i \in I} \varphi_i \to \psi_i]\!] \subseteq [\![\varphi_j \to \psi_j]\!] \quad \Rightarrow \quad \bigwedge_{i \in I} \varphi_i \to \psi_i \leq \varphi_j \to \psi_j$$

the rules for $\wedge$ will give the desired implication. Leaving out the subscript on φ_j, ψ_j, the proposition we have to prove can be stated as follows: if $\mathsf{Irr}(\varphi_i), \mathsf{Irr}(\varphi)$, then

$$[\![\bigwedge_{i \in I} \varphi_i \to \psi_i]\!] \subseteq [\![\varphi \to \psi]\!] \quad \Rightarrow \quad \bigwedge_{i \in I} \varphi_i \to \psi_i \leq \varphi \to \psi.$$

The assumption $\mathsf{Irr}(\varphi), \mathsf{Irr}(\varphi_i)$ means that there exist a, a_i such that $[\![\varphi]\!] = {\downarrow}\{a\}$ and $[\![\varphi_i]\!] = {\downarrow}\{a_i\}$ for all $i \in I$. The definition of the semantics of function type formulae translates the hypothesis to

$$\bigcap_{i \in I} \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f({\downarrow}\{a_i\}) \subseteq [\![\psi_i]\!]\} \subseteq \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f({\downarrow}\{a\}) \subseteq [\![\psi]\!]\}.$$

We now construct a linear function h belonging to the left-hand side as follows. Consider the set $I_0 = \{i \in I \mid \downarrow\{a\} \subseteq \downarrow\{a_i\}\}$, and let G denote the set of pairs

$$G = \{(\downarrow\{a_i\}, [\![\psi_i]\!]) \mid i \in I \setminus I_0\} \;\cup\; \{(\downarrow\{a_j\}, \bigcap_{i\in I_0} [\![\psi_i]\!]) \mid j \in I_0\}.$$

We can view G as a partial graph for a linear function. G induces a monotone map from downward closure of single elements to lower sets as follows:

$$\downarrow\{b\} \longmapsto \begin{cases} \mathcal{J}(\tau) & \text{if } \;\nexists\,(a, ha) \in G \,.\downarrow\{b\} \subseteq a \\ \bigcap\{ha : (a, ha) \in G \text{ and } \downarrow\{b\} \subseteq a\} & \text{otherwise} \end{cases}$$

By construction, the function h obtained by linear extension of this map satisfies

$$h(\downarrow\{a\}) = \bigcap_{i\in I_0} [\![\psi_i]\!]$$

and belongs to the intersection of functions on the left-hand side of the preceding inequality. From this we get that

$$\bigcap_{i\in I_0} [\![\psi_i]\!] \subseteq [\![\psi]\!],$$

and by the induction hypothesis we can prove this inclusion in the formal system; that is,

$$\bigwedge_{i\in I_0} \psi_i \le \psi.$$

Furthermore we have that

$$[\![\varphi]\!] = \downarrow\{a\} \subseteq \bigcap_{i\in I_0} \downarrow\{a_i\} = [\![\bigwedge_{i\in I_0} \varphi_i]\!];$$

so, again by completeness we can prove

$$\varphi \le \bigwedge_{i\in I_0} \varphi_i.$$

Putting these two facts together we can prove that

$$(\bigwedge_{i\in I_0} \varphi_i) \to (\bigwedge_{i\in I_0} \psi_i) \;\le\; \varphi \to \psi.$$

The axioms for $\to$ furthermore give that

$$\bigwedge_{i\in I_0} (\varphi_i \to \psi_i) \le \bigwedge_{i\in I_0} ((\bigwedge_{i\in I_0} \varphi_i) \to \psi_i) \le (\bigwedge_{i\in I_0} \varphi_i) \to (\bigwedge_{i\in I_0} \psi_i)$$

which leads to the desired inequality

$$\bigwedge_{i\in I} (\varphi_i \to \psi_i) \le \bigwedge_{i\in I_0} (\varphi_i \to \psi_i) \le (\bigwedge_{i\in I_0} \varphi_i) \to (\bigwedge_{i\in I_0} \psi_i) \le \varphi \to \psi.$$

For a recursive type $\mu\alpha. \sum_{k\in K} \sigma_k \times \alpha^{n_k}$ we assume

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

with $\varphi_i, \psi_j \in L(\sum_{k\in K} \sigma_k)$. Let $c_i, d_k \in \mathcal{J}(\sum_{k\in K} \sigma_k)$ be the elements given by Theorem 4.5.1 satisfying

$$\downarrow\{c_i\} = [\![\varphi_i]\!] \quad \text{and} \quad \downarrow\{d_j\} = [\![\psi_j]\!],$$

and let $C = \{c_i : i \in I\}$ and $D = \{d_j : j \in J\}$. By Lemma 4.4.5 the preceding inequality is equivalent to $\downarrow\{\overline{C}\} \subseteq \downarrow\{\overline{D}\}$, implying that $C \sqsubseteq_{EM} D$ and hence that

$$\forall i \exists j. c_i \sqsubseteq d_j \quad \text{and} \quad \forall j \exists i. c_i \sqsubseteq d_j.$$

So, by induction hypothesis

$$\forall i \exists j. \varphi_i \sqsubseteq \psi_j \quad \text{and} \quad \forall j \exists i. \varphi_i \sqsubseteq \psi_j$$

from which it folows that $\Box(\bigvee_{i\in I} \varphi_i) \leq \Box(\bigvee_{j\in J} \psi_j)$ and $\bigwedge_{i\in I} \Diamond\varphi_i \leq \bigwedge_{j\in J} \Diamond\psi_j$. $\square$

A.5 Soundness of Strictness Logic (Theorem 4.7.1)

LEMMA A.5.1. *Let e be an expression, ρ an environment defining the free variables in e, and L_1 and L_2 lower sets. Then*

$$[\![e]\!]\rho[x \mapsto L_1 \cup L_2] = [\![e]\!]\rho[x \mapsto L_1] \cup [\![e]\!]\rho[x \mapsto L_2].$$

PROOF. The proof is an induction over the structure of e. The cases where e is a variable, a constant, or an operator are easy. For pairing we have

$$\begin{aligned}
&[\![(e_1, e_2)]\!]\rho[x \mapsto L_1 \cup L_2] \\
=\ & \bigcup\{[\![e_1]\!][x_i \mapsto v_i][x \mapsto l] \times [\![e_2]\!][x_i \mapsto v_i][x \mapsto l] \mid v_i \in \rho(x_i), l \in L_1 \cup L_2\} \\
=\ & \bigcup\{[\![e_1]\!][x_i \mapsto v_i][x \mapsto l] \times [\![e_2]\!][x_i \mapsto v_i][x \mapsto l] \mid v_i \in \rho(x_i), l \in L_1\} \\
\cup\ & \bigcup\{[\![e_1]\!][x_i \mapsto v_i][x \mapsto l] \times [\![e_2]\!][x_i \mapsto v_i][x \mapsto l] \mid v_i \in \rho(x_i), l \in L_2\} \\
=\ & [\![(e_1, e_2)]\!]\rho[x \mapsto L_1] \cup [\![(e_1, e_2)]\!]\rho[x \mapsto L_2].
\end{aligned}$$

The case where e is an application is dealt with similarly. In the case of lambda abstraction we first assume that the variable abstracted over is different from x.

$$\begin{aligned}
&[\![\lambda y.e]\!]\rho[x \mapsto L_1 \cup L_2] \\
=\ & \bigcup \downarrow\{\lambda M. \textstyle\bigcup_{m\in M} [\![e]\!]\rho[x \mapsto \downarrow\{l\}][y \mapsto \downarrow\{m\}] \mid l \in L_1 \cup L_2\} \\
=\ & \bigcup \downarrow\{\lambda M. \textstyle\bigcup_{m\in M} [\![e]\!]\rho[x \mapsto \downarrow\{l\}][y \mapsto \downarrow\{m\}] \mid l \in L_1\} \\
\cup\ & \bigcup \downarrow\{\lambda M. \textstyle\bigcup_{m\in M} [\![e]\!][x \mapsto \downarrow\{l\}][y \mapsto \downarrow\{m\}] \mid l \in L_2\} \\
=\ & [\![\lambda y.e]\!]\rho[x \mapsto L_1] \cup [\![\lambda y.e]\!]\rho[x \mapsto L_2]
\end{aligned}$$

If $y = x$ then the interpretation of the lambda expression is independent of what x is bound to in the environment. The rest of the cases are handled by the same technique. $\square$

THEOREM. *Let e be an expression of type σ, and let ρ and Γ be environments of values and formulae, respectively, such that $\rho(x) = [\![\Gamma(x)]\!]$ for all free variables x in e. Then for all $\varphi \in L(\sigma)$*

$$\Gamma \vdash e : \varphi \;\Rightarrow\; [\![e]\!]\rho \subseteq [\![\varphi]\!].$$

PROOF. The proof is by induction over the structure of the derivation in the program logic, which amounts to checking the validity of each rule in the program logic. The validity of the rules **Conj**, **Var**, and **Taut** are straightforward. The soundness of rule **Disj** is Lemma A.5.1. In the rule for pairing **Pair** we can assume that $[\![e_1]\!]\rho \subseteq [\![\varphi]\!]$ and $[\![e_2]\!]\rho \subseteq [\![\psi]\!]$. This implies

$$[\![(e_1, e_2)]\!]\rho = [\![e_1]\!]\rho \times [\![e_2]\!]\rho \subseteq [\![\varphi]\!] \times [\![\psi]\!] = [\![\varphi \times \psi]\!].$$

Validity of the rules **Fst** and **Snd** follows directly from the definitions. For lambda abstraction **Abs** the assumption is that, with $\psi \in L(\tau)$,

$$[\![e]\!]\rho[x^\sigma \mapsto [\![\varphi]\!]] \subseteq [\![\psi]\!].$$

The abstract interpretation is monotone in the environment, so we get

$$\begin{aligned}
&[\![\lambda x^\sigma.e]\!]\rho = \downarrow\{\lambda L.\textstyle\bigcup_{l \in L}[\![e]\!]\rho[x^\sigma \mapsto \downarrow\{l\}]\} \\
&\subseteq \downarrow\{\lambda L.[\![e]\!]\rho[x^\sigma \mapsto L]\} \\
&\subseteq \{f \in [\![\sigma]\!] \multimap [\![\tau]\!] \mid f([\![\varphi]\!]) \subseteq [\![\psi]\!]\} \qquad \text{(by the preceding assumption)} \\
&= [\![\varphi \rightarrow \psi]\!].
\end{aligned}$$

For function application the assumptions are that

$$[\![e_1]\!]\rho \subseteq \{f \mid f([\![\varphi]\!]) \subseteq [\![\psi]\!]\} \quad \text{and} \quad [\![e_2]\!]\rho \subseteq [\![\varphi]\!].$$

Since application is just interpreted as applying each function in $[\![e_1]\!]\rho$ to the lower set $[\![e_2]\!]\rho$ and taking the union of the results, this gives a set that is a subset of $[\![\psi]\!]$.

To prove the **Fix** axiom sound we must show that the interpretation $[\![\mathsf{fix}]\!]$ defines a set of higher-order linear functions that maps linear functions belonging to $[\![\varphi \rightarrow \varphi]\!]$ into the set $[\![\varphi]\!]$. The interpretation of fix is given by

$$[\![\mathsf{fix}]\!]\rho = \downarrow\{\lambda\!\downarrow\!\{f_1, \dots, f_k\}.\textstyle\bigcup_{i=1}^k \{\bigcup_{n=0}^\infty f_i^n(\{\bot\})\}\},$$

and the function

$$\lambda\!\downarrow\!\{f_1, \dots, f_k\}.\bigcup_{i=1}^k \{\bigcup_{n=0}^\infty f_i^n(\{\bot\})\}$$

is obviously linear. When passed a set of linear functions $\{f_1, \dots, f_k\} \subseteq [\![\varphi \rightarrow \varphi]\!]$ we know that for each f_i

$$\bigcup_{n=0}^\infty f_i^n(\{\bot\}) \subseteq [\![\varphi]\!],$$

since $\{\bot\} \subseteq [\![\varphi]\!]$. Taking the union over all the f_i still gives a set contained in $[\![\varphi]\!]$, so this gives the desired property of $[\![\mathsf{fix}]\!]$. For the rule (**Case** $- \vee$) we assume that

$$[\![e]\!]\rho \subseteq [\![\bigvee_{i \in I} \mathsf{in}_i(\varphi_i)]\!] = \bigcup_{i \in I}\{\mathsf{in}_i(d) : d \in [\![\varphi_i]\!]\}$$

and

$$\forall i \in I.[\![e_i]\!]\rho[x_i \mapsto [\![\varphi_i]\!]] \subseteq [\![\psi_i]\!].$$

Then

$$\begin{aligned}
&[\![\mathsf{case}\ e\ \mathsf{of}\ \{\mathsf{in}_i(x_i)\ .\ e_i\}]\!]\rho \\
&= \bigcup\{[\![e_i]\!]\rho[x_i \mapsto \downarrow\{d\}] : \mathsf{in}_i(d) \in [\![e]\!]\rho\} \\
&\subseteq \bigcup\{[\![e_i]\!]\rho[x_i \mapsto \downarrow\{d\}] : d \in [\![\varphi_i]\!]\} \\
&\subseteq \textstyle\bigcup_{i\in I}[\![e_i]\!]\rho[x_i \mapsto [\![\varphi_i]\!]] \\
&\subseteq \textstyle\bigcup_{i\in I}[\![\psi_i]\!] = [\![\bigvee_{i\in I}\psi_i]\!].
\end{aligned}$$

For the unfolding of recursive structures we prove soundness for the rule **Fold** rather than for the two rules **Fold** $-\Box$ and **Fold** $-\Diamond$. This suffices, since both are derivable from **Fold**. We have to prove

$$\downarrow\{\mathsf{D}(\mathit{fold})\} \subseteq [\![\mathsf{in}_j(\varphi\times(\Box\bigvee_{i\in I}\psi_i\wedge\bigwedge_{i\in I}\Diamond\psi_i)^{n_j}) \to \Box(\mathsf{in}_j(\varphi)\vee\bigvee_{i\in I}\psi_i)\wedge\Diamond\mathsf{in}_j(\varphi)\wedge\bigwedge_{i\in I}\Diamond\psi_i]\!]$$

where $\mathit{fold}(\mathit{in}_j(d, C_1, \ldots, C_{n_j})) = \downarrow\{\overline{C_1 \cup \ldots \cup C_{n_j} \cup \{\mathit{in}_j(d)\}}\}$. For this it suffices to show that $\mathsf{D}(\mathit{fold})$ is in the set on the right-hand side of the inclusion. Since $\mathsf{D}(\mathit{fold})$ is defined by linear extension it is enough to show that

$$\mathit{fold}(\mathit{in}_j(d, C_1, \ldots, C_{n_j})) \subseteq [\![\Box(\mathsf{in}_j(\varphi) \vee \bigvee_{i\in I}\psi_i) \wedge \Diamond\mathsf{in}_j(\varphi) \wedge \bigwedge_{i\in I}\Diamond\psi_i]\!]$$

for all $\mathit{in}_j(d, C_1, \ldots, C_{n_j}) \in [\![\mathsf{in}_j(\varphi \times (\Box\bigvee_{i\in I}\psi_i \wedge \bigwedge_{i\in I}\Diamond\psi_i)^{n_j})]\!]$. So we have to prove the inclusion

$$\overline{C_1 \cup \ldots \cup C_{n_j} \cup \{\mathit{in}_j(d)\}} \subseteq [\![\Box(\mathsf{in}_j(\varphi) \vee \bigvee_{i\in I}\psi_i) \wedge \Diamond\mathsf{in}_j(\varphi) \wedge \bigwedge_{i\in I}\Diamond\psi_i]\!]$$

given that each of the C_i belongs to $[\![\Box\bigvee_{i\in I}\psi_i \wedge \bigwedge_{i\in I}\Diamond\psi_i]\!]$. First we observe that

$$C_1 \cup \ldots \cup C_{n_j} \cup \{\mathit{in}_j(d)\} \subseteq [\![\mathsf{in}_j(\varphi) \vee \bigvee_{i\in I}\psi_i]\!]$$

and that the intersection of the set $C_1 \cup \ldots \cup C_{n_j} \cup \{\mathit{in}_j(d)\}$ with $[\![\mathsf{in}_j(\varphi)]\!]$ as well as with each of the $[\![\psi_i]\!]$ is nonempty. Because $[\![\mathsf{in}_j(\varphi) \vee \bigvee_{i\in I}\psi_i]\!]$ is downward closed, and therefore convex, and because taking convex closure only adds elements to a set, the set $\overline{C_1 \cup \ldots \cup C_{n_j} \cup \{\mathit{in}_j(d)\}}$ satisfies the same properties, so

$$\overline{C_1 \cup \ldots \cup C_{n_j} \cup \{\mathit{in}_j(d)\}} \in [\![\Box(\mathsf{in}_j(\varphi) \vee \bigvee_{i\in I}\psi_i)]\!]\ \cap\ [\![\Diamond\mathsf{in}_j(\varphi)]\!]\ \cap\ \bigcap_{i\in I}[\![\Diamond\psi_i]\!]$$

from which the desired inclusion follows. Likewise, for **unfold** $-\Diamond$ we have to show

$$\mathit{unfold}(C) \subseteq [\![\bigvee_{k\in I}\mathsf{in}_k((\bigvee_{j\in J_k}\varphi_{kj}) \times (\bigwedge_{i\in I\setminus\{k\}}\bigwedge_{j\in J_i}\Diamond\ \mathsf{in}_i(\varphi_{ij}))^{n_i})]\!]$$

for all $C \in [\![\Box\bigvee_{i\in I}\bigvee_{j\in J_i}\mathsf{in}_i(\varphi_{ij}) \wedge \bigwedge_{i\in I}\bigwedge_{j\in J_i}\Diamond\ \mathsf{in}_i(\varphi_{ij})$. We recall that

$$\mathit{unfold}(C) = \downarrow\{\mathsf{in}_k(d, \tilde{C}^{n_k}) : \mathsf{in}_k(d) \in C, C_{\neq k} \subseteq \tilde{C} \subseteq C\}.$$

But for any $k, d, \tilde{C}$ satisfying $\mathsf{in}_k(d) \in C$ and $C_{\neq k} \subseteq \tilde{C} \subseteq C$ we have

$$C \subseteq [\![\bigvee_{i\in I}\bigvee_{j\in J_i}\mathsf{in}_i(\varphi_{ij})]\!] = \bigcup_{i\in I}[\![\mathsf{in}_i(\bigvee_{j\in J_i}\varphi_{ij})]\!],$$

hence $\mathsf{in}_k(d) \in [\![\mathsf{in}_k(\bigvee_{j\in J_k}\varphi_{kj})]\!]$, so $d \in [\![\bigvee_{j\in J_k}\varphi_{kj}]\!]$. Furthermore, $C_{\neq k} \subseteq \tilde{C}$ implies that $\tilde{C} \cap [\![\mathsf{in}_i(\varphi_{ij})]\!]$ is nonempty for all $i \neq k$, so

$$\tilde{C} \in [\![\bigwedge_{i\in I\setminus\{k\}} \bigwedge_{j\in J_i} \Diamond\, \mathsf{in}_i(\varphi_{ij})]\!].$$

A similar argument, now using the inclusion

$$\tilde{C} \subseteq C \subseteq [\![\bigvee_{i\in I}\bigvee_{j\in J_i} \mathsf{in}_i(\varphi_{ij})]\!],$$

shows that the rule **Unfold** $-\,\Box$ is sound. $\square$

A.6 Completeness of the Strictness Logic (Theorem 4.7.1)

THEOREM. *Let e be an expression of type σ, and let ρ and Γ be environments of values and formulae, respectively, such that $\rho(x) = [\![\Gamma(x)]\!]$ for all free variables x in e. Then for all $\varphi \in L(\sigma)$*

$$[\![e]\!]\rho \subseteq [\![\varphi]\!] \;\Rightarrow\; \Gamma \vdash e : \varphi.$$

PROOF. We first prove completeness under the assumption that the environment maps variables to irreducible formulae. Since $\rho(x) = [\![\Gamma(x)]\!]$ this implies that $\rho(x)$ is the downward closure of a single element. The proof proceeds by induction over the structure of the term e.

x^σ. Follows from the assumption on ρ and Γ.

c. We have $[\![c]\!]\rho = \downarrow\{1\} = [\![\mathtt{t}_\sigma]\!]$ and $\Gamma \vdash c : \mathtt{t}_\sigma$.

(e_1, e_2):. Since all variables in ρ are bound to downward closures of single elements, the assumption can be expressed as

$$[\![(e_1,e_2)]\!]\rho = [\![e_1]\!]\rho \times [\![e_2]\!]\rho \subseteq [\![\varphi]\!].$$

Due to the Definability Theorem, we can find formulae φ_1 and φ_2 such that

$$[\![e_1]\!]\rho = [\![\varphi_1]\!] \qquad \text{and} \qquad [\![e_2]\!]\rho = [\![\varphi_2]\!].$$

Hence by the induction hypothesis we can prove $\Gamma \vdash e_1 : \varphi_1$ and $\Gamma \vdash e_2 : \varphi_2$, and therefore $\Gamma \vdash (e_1,e_2) : \varphi_1 \times \varphi_2$. Since $[\![\varphi_1 \times \varphi_2]\!] \subseteq [\![\varphi]\!]$, we have by completeness of the axiomatization of $\leq$ that $\varphi_1 \times \varphi_2 \leq \varphi$; hence by the rule **Weak**

$$\Gamma \vdash (e_1, e_2) : \varphi.$$

$\mathsf{fst}(e)$. If $[\![\mathsf{fst}(e)]\!]\rho = \{a \mid (a,b) \in [\![e]\!]\rho\} \subseteq [\![\varphi]\!]$, then $[\![e]\!]\rho \subseteq [\![\varphi]\!] \times [\![\mathtt{t}]\!] = [\![\varphi \times \mathtt{t}]\!]$; so by the induction hypothesis $\Gamma \vdash e : \varphi \times \mathtt{t}$; hence $\Gamma \vdash \mathsf{fst}(e) : \varphi$. We use a similar proof for snd.

$\lambda x.e$. Assume

$$[\![\lambda x.e]\!]\rho = \downarrow\{\lambda L.\textstyle\bigcup_{l\in L}[\![e]\!]\rho[x \mapsto \downarrow\{l\}]\} \subseteq [\![\displaystyle\bigvee_{i\in I}(\bigwedge_{j\in J_i}(\varphi_{ij} \to \psi_{ij}))]\!]$$

with $\mathsf{Irr}(\varphi_{ij})$. This implies that there exists an $i \in I$ such that

$$\downarrow\{\lambda L.\textstyle\bigcup_{l\in L}[\![e]\!]\rho[x \mapsto \downarrow\{l\}]\} \subseteq [\![\displaystyle\bigwedge_{j\in J_i}\varphi_{ij} \to \psi_{ij}]\!].$$

Since all φ_{ij} are irreducible we have for all $j \in J_i$ that $[\![e]\!]\rho[x \mapsto [\![\varphi_{ij}]\!]] \subseteq [\![\psi_{ij}]\!]$, so it follows by induction hypothesis that $\Gamma[x : \varphi_{ij}] \vdash e : \psi_{ij}$. By using the rule **Abs** it follows that $\Gamma \vdash \lambda x.e : \varphi_{ij} \to \psi_{ij}$ for every $j \in J_i$. By the rule **Conj** and the rule **Weak** we then derive

$$\Gamma \vdash \lambda x.e : \bigvee_{i\in I}(\bigwedge_{j\in J_i}(\varphi_{ij} \to \psi_{ij})).$$

$e_1\, e_2$. Assume that $[\![e_1 e_2]\!]\rho = [\![e_1]\!]\rho @ [\![e_2]\!]\rho \subseteq [\![\psi]\!]$. With $[\![e_1]\!]\rho = \downarrow\{f_1, \ldots, f_k\}$ and φ chosen such that $[\![e_2]\!]\rho = [\![\varphi]\!]$, the assumption tells us that $f_i([\![\varphi]\!]) \subseteq [\![\psi]\!]$ for all $i = 1, \ldots, k$. This implies that $[\![e_1]\!]\rho \subseteq [\![\varphi \to \psi]\!]$; hence by induction hypothesis we can prove $\Gamma \vdash e_1 : \varphi \to \psi$ and $\Gamma \vdash e_2 : \varphi$; so by the rule **App** we have $\Gamma \vdash e_1\, e_2 : \psi$.

fix . The fixed-point operator has type $(\sigma \to \sigma) \to \sigma$; so by using the normal form for formulae in $L((\sigma \to \sigma) \to \sigma)$ we can express the assumption as

$$[\![\mathsf{fix}\ ]\!] = \downarrow\{\mathsf{D}(\lambda f. \textstyle\bigcup_{n=0}^{\infty} f^n(\{\bot\}))\} \subseteq [\![\displaystyle\bigvee_{i\in I}(\bigwedge_{j\in J_i}(\varphi_{ij} \to \psi_{ij}))]\!],$$

where $\mathsf{Irr}(\varphi_{ij})$. This means that there exists an $i \in I$ such that

$$[\![\mathsf{fix}\ ]\!] \subseteq [\![\bigwedge_{j\in J_i}(\varphi_{ij} \to \psi_{ij})]\!],$$

and completeness follows if we can show that for all $j \in J_i$,

$$[\![\mathsf{fix}\ ]\!] \subseteq [\![\varphi_{ij} \to \psi_{ij}]\!] \ \Rightarrow \vdash \mathsf{fix}\ : \varphi_{ij} \to \psi_{ij}.$$

Because $\mathsf{Irr}(\varphi_{ij})$, there exists $f_{ij} \in [\![\sigma]\!] \multimap [\![\sigma]\!]$ such that $\downarrow\{f_{ij}\} = [\![\varphi_{ij}]\!]$. Furthermore, each f_{ij} has a least fixed point, and we can find $\varphi_{f_{ij}}$ corresponding to that fixed point such that

$$[\![\varphi_{f_{ij}}]\!] = \bigcup_{k=0}^{\infty} f_{ij}^k(\{\bot\}) \subseteq [\![\psi_{ij}]\!].$$

If $f \in \downarrow\{f_{ij}\}$, then $f([\![\varphi_{f_{ij}}]\!]) \subseteq [\![\varphi_{f_{ij}}]\!]$; hence $[\![\varphi_{ij}]\!] \subseteq [\![\varphi_{f_{ij}} \to \varphi_{f_{ij}}]\!]$. Completeness of the axiomatization (Theorem 4.5.4) implies that

$$\varphi_{ij} \le \varphi_{f_{ij}} \to \varphi_{f_{ij}} \text{ and } \varphi_{f_{ij}} \le \psi_{ij};$$

so

$$(\varphi_{f_{ij}} \to \varphi_{f_{ij}}) \to \varphi_{f_{ij}} \le \varphi_{ij} \to \psi_{ij}.$$

By the rules **Fix** and **Weak** we can then deduce that

$$\vdash \mathsf{fix}\ : \varphi_{ij} \to \psi_{ij}.$$

$\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. Assume $[\![\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!] \subseteq [\![\varphi]\!]$. If $[\![b]\!]\rho = \{0\}$, then $\Gamma \vdash b : \mathtt{f}^{\mathsf{Bool}}$, and we can prove $\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathtt{f} \le \varphi$. Otherwise we have $[\![e_1]\!]\rho \cup [\![e_2]\!]\rho \subseteq [\![\varphi]\!]$ which implies $[\![e_1]\!]\rho \subseteq [\![\varphi]\!]$ and $[\![e_2]\!]\rho \subseteq [\![\varphi]\!]$, and by induction we have $\Gamma \vdash e_1 : \varphi$ and $\Gamma \vdash e_2 : \varphi$; so $\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \varphi$.

$\mathsf{case}\ e\ \mathsf{of}\ \{in_i(x_i)\ .\ e_i\}$. Assume $[\![e]\!]\rho = [\![\bigvee_{i\in I} \mathsf{in}_i(\bigvee_{j\in J_i}\varphi_{ij})]\!]$ with $\mathsf{Irr}(\varphi_{ij})$. If

$$\bigcup\{[\![e_i]\!]\rho[x_i = \downarrow\{d\}] : in_i(d) \in [\![e]\!]\rho\} \subseteq [\![\psi]\!],$$

then $[\![e_i]\!]\rho[x_i \mapsto [\![\varphi_{ij}]\!]] \subseteq [\![\psi]\!]$. By the induction hypothesis we have

$$\Gamma \vdash e : \bigvee_{i\in I} \mathsf{in}_i(\bigvee_{j\in J_i} \varphi_{ij}) \quad \text{and} \quad \Gamma[x_i : \varphi_{ij}] \vdash e_i : \psi$$

and therefore, by the rule **Disj**, $\Gamma[x_i : \bigvee_{j\in J_i} \varphi_{ij}] \vdash e_i : \psi$. Applying the rule **Case** $- \vee$ we get

$$\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{in_i(x_i)\ .\ e_i\} : \psi.$$

fold. The interpretation of fold is defined as the linear extension of the function

$$in_j(d, C_1, \ldots, C_{n_j}) \longmapsto \downarrow\{\overline{C_1 \cup \ldots \cup C_{n_j} \cup in_j(d)}\}.$$

For given $d, C_1, \ldots, C_{n_j}$, we can find φ, ψ_{ki} such that $\downarrow\{d\} = [\![\varphi]\!]$ and such that for each element $c_{ki} \in C_k$ there is a ψ_{ki} satisfying $\downarrow\{c_{ki}\} = [\![\varphi_{ki}]\!]$. By Lemma 4.4.5 this means that $\downarrow\{C_k\} = [\![\Box \bigvee_{i\in I_k} \psi_{ki} \wedge \bigwedge_{i\in I_k} \Diamond\psi_{ki}]\!]$ for $k = 1, \ldots, n_j$. Completeness for fold restricted to downward closures of a single element follows from the **Fold** rule by showing that

$$\downarrow\{\overline{C_1 \cup \ldots \cup C_{n_j} \cup \{in_j(d)\}}\} = [\![\Box(\mathsf{in}_j(\varphi) \vee \bigvee_{k=1}^{n_j} \bigvee_{i\in I_k} \psi_{ki}) \wedge \Diamond\mathsf{in}_j(\varphi) \wedge \bigwedge_{k=1}^{n_j} \bigwedge_{i\in I_k} \Diamond\psi_{ki}]\!].$$

This is a direct consequence of Lemma 4.4.5. To argue completeness for $[\![\mathsf{fold}]\!] = \mathsf{D}(\mathit{fold})$ for arbitrary arguments we make the following observation. If for a function f and formulae φ_i, ψ_i such that $[\![\varphi_i]\!] = \downarrow\{d_i\}$ and $[\![\psi_i]\!] = \downarrow\{f(\downarrow\{d_i\})\}$ we can prove $f : \varphi_i \to \psi_i$ for all $i \in I$, then we can also prove $f : \varphi_i \to \bigvee_{i\in I} \varphi_i$ by weakening of the result, and from this follows that $f : \bigvee_{i\in I} \varphi_i \to \bigvee_{i\in I} \psi_i$.

unfold. We defined $[\![\mathsf{unfold}]\!]$ as the linear extension of the function

$$C \longmapsto \downarrow\{in_j(d, \tilde{C}^{n_j}) : in_j(d) \in C \textit{ and } C_{\neq j} \subseteq \tilde{C} \subseteq C\}.$$

For given $C = \bigcup_{i\in I} \bigcup_{j\in J_i} \{\mathsf{in}_i(d_{ij})\}$, let φ_{ij} be chosen such that $\downarrow\{in_i(d_{ij})\} = [\![\mathsf{in}_i(\varphi_{ij})]\!]$. From the discussion following Definition 4.6.2 we have that proving completeness amounts to showing that the set

$$[\![\bigvee_{J\subseteq J_k} \Box[\bigvee_{i\in I\setminus\{k\}, j\in J_i} \mathsf{in}_i(\varphi_{ij}) \vee \bigvee_{j\in J} \mathsf{in}_k(\varphi_{kj})] \wedge \bigwedge_{i\in I\setminus\{k\}, j\in J_i} \Diamond\ \mathsf{in}_i(\varphi_{ij}) \wedge \bigwedge_{j\in J} \Diamond\ \mathsf{in}_k(\varphi_{kj})]\!]$$

is equal to the set

$$\downarrow\{\tilde{C} : C_{\neq k} \subseteq \tilde{C} \subseteq C\}.$$

Let $\tilde{C}$ satisfy $C_{\neq k} \subseteq \tilde{C} \subseteq C$, and define the set $J \subseteq J_k$ such that $\{\mathsf{in}_k(d_{kj}) : j \in J\} = \tilde{C} \setminus C_{\neq k}$. Then we have

$$\{\mathsf{in}_k(d_{kj}) : j \in J\} \subseteq \bigcup_{j\in J} [\![\varphi_{kj}]\!],$$

and likewise

$$\{\mathsf{in}_k(d_{kj}) : j \in J\} \cap [\![\varphi_{kj}]\!]$$

is nonempty for all $j \in J$ from which it follows that

$$\tilde{C} \subseteq [\![\bigvee_{i\in I\setminus\{k\}, j\in J_i} \mathsf{in}_i(\varphi_{ij}) \vee \bigvee_{j\in J} \mathsf{in}_k(\varphi_{kj})]\!]$$

as well as

$$\tilde{C} \cap [\![\mathsf{in}_i(\varphi_{ij}]\!]$$

are nonempty for all $(i,j) \in \{(i,j) : i \in I \setminus \{k\}, j \in J_i\} \cup \{(k,j) : j \in J\}$. This proves one of the inclusions ($\supseteq$).

For the other inclusion let $J \subseteq J_k$, and let C' be an element of

$$[\![\Box[\bigvee_{i \in I\setminus\{k\}, j \in J_i} \mathsf{in}_i(\varphi_{ij}) \vee \bigvee_{j \in J} \mathsf{in}_k(\varphi_{kj})] \wedge \bigwedge_{i \in I\setminus\{k\}, j \in J_i} \Diamond\, \mathsf{in}_i(\varphi_{ij}) \wedge \bigwedge_{j \in J} \Diamond\, \mathsf{in}_k(\varphi_{kj}).]\!]$$

With c_{kj} and φ_{kj} as in the preceding, define $\tilde{C} = \overline{C_{\neq k} \cup \{c_{kj} : j \in J\}}$. Since

$$C' \subseteq \bigcup_{i \in I\setminus\{k\}, j \in J_i} [\![\mathsf{in}_i(\varphi_{ij})]\!] \cup \bigcup_{j \in J} [\![\mathsf{in}_i(\varphi_{kj})]\!]$$

then for each $c' \in C'$ there exists $\tilde{c} \in \tilde{C}$ such that $c' \sqsubseteq \tilde{c}$. Similarly, since $C' \cap [\![\varphi_{ij}]\!]$ is nonempty for $(i,j) \in \{(i,j) : i \in I \setminus \{k\}, j \in J_i\} \cup \{(k,j) : j \in J\}$ there exists for every $\tilde{c} \in \tilde{C}$ a $c' \in C'$ such that $c' \sqsubseteq \tilde{c}$. Hence $C' \sqsubseteq_{EM} \tilde{C}$, and the other inclusion follows. Completeness for the linear extension $\mathsf{D}(\mathit{unfold})$ now follows by exactly the same argument as for fold.

It remains to show how the rule **Disj** enables us to extend the preceding results to arbitrary environments. Assume $\Gamma = [x_1 : \bigvee_{j \in J_1} \varphi_{1j}, \ldots, x_n : \bigvee_{j \in J_n} \varphi_{nj}]$. Then for all possible choices of environments $[x_i \mapsto [\![\varphi_{ij}]\!]]$ with $j \in J_i$ we have

$$[\![e]\!][x_i \mapsto [\![\varphi_{ij}]\!]] \subseteq [\![e]\!]\rho \subseteq [\![\varphi]\!]$$

and hence by the previous theorem

$$[x_i : \varphi_{ij}] \vdash e : \varphi.$$

Now fix the choice of φ_{ij} for $i = 1, \ldots, n-1$. Then

$$\forall j \in J_n\ .\ [x_1 : \varphi_1, \ldots, x_{n-1} : \varphi_{n-1}, x_n : \varphi_{nj}] \vdash e : \varphi;$$

hence by the rule **Disj**

$$[x_1 : \varphi_1, \ldots, x_{n-1} : \varphi_{n-1}, x_n : \bigvee_{j \in J_n} \varphi_{nj}] \vdash e : \varphi.$$

By repeating this process for each x_i we finally obtain

$$[x_1 : \bigvee_{j \in J_1} \varphi_{1j}, \ldots, x_n : \bigvee_{j \in J_n} \varphi_{nj}] \vdash e : \varphi.$$

as desired. □

ACKNOWLEDGMENT

This article combines and improves results and corrects some errors that have appeared in earlier papers [Jensen 1992a; 1992b; 1992c; 1994]. Many people have contributed with comments and discussions pertaining to the subject of this article; I wish in particular to thank N. Benton, P. Burton, C. Colby, C. Hankin, and P. Wadler. Special thanks to A. Kennedy, F. Vedrine, and the referees for very valuable feedback on an earlier version.

REFERENCES

Abramsky, S. 1991a. A domain equation for bisimulation. *Inf. Comput. 92,* 2, 161–218.

Abramsky, S. 1991b. Domain theory in logical form. *Ann. Pure Appl. Logic 51*, 1–77.

Aiken, A., Wimmers, E. L., and Lakshman, T. K. 1994. Soft typing with conditional types. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages*. ACM Press, New York, 163–173.

Bandelt, H.-J. 1980. The tensor product of continuous lattices. *Mathematische Zeitschrift 172*, 89–96.

Barbanera, F., Dezani-Ciancaglini, M., and De'Liguoro, U. 1995. Intersection and union types: Syntax and semantics. *Inf. Comput. 119*, 202–230.

Benton, P. N. 1992a. Strictness analysis and optimising transformations for lazy functional programs. Ph.D. thesis, Univ. of Cambridge, Cambridge, U.K.

Benton, P. N. 1992b. Strictness logic and polymorphic invariance. In *Proceedings of the 2nd International Symposium on Logical Foundations of Computer Science*, A. Nerode and M. Taitslin, Eds. Lecture Notes in Computer Science, vol. 620. Springer Verlag, Berlin.

Benton, P. N. 1993. Strictness properties of lazy algebraic datatypes. In *Static Analysis*, P. C. et. al., Ed. Lecture Notes in Computer Science, vol. 724. Springer Verlag, Berlin.

Burn, G. L. 1991. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing. Pitman, London, U.K.

Burn, G. L. 1992. Properties of abstract interpretation techniques. Tech. Rep. Doc 92/19, Imperial College, Univ. of London, London, U.K.

Burn, G. L., Hankin, C., and Abramsky, S. 1986. The theory and practice of strictness analysis for higher order functions. *Sci. Comput. Program. 7*, 249–278.

Cousot, P. and Cousot, R. 1976. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming.* Dunod, Paris, France, 106–130.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages.* ACM Press, New York, 269–282.

Cousot, P. and Cousot, R. 1992. Comparing the Galois connection and widening/narrowing approach to abstract interpretation. Tech. Rep. LIX/RR/92/09, Ecole Polytechnique, Palaiseau, France.

Cousot, P. and Cousot, R. 1994. Higher-order abstract interpretation (and application to comportment analysis generalising strictness, termination, projection and PER analysis of functional languages). In *Proceedings of the 5th IEEE International Conference on Computer Languages*, H. Bal, Ed. IEEE Press, New York, 95–112.

Davey, B. and Priestley, H. 1990. *Introduction to Lattices and Order.* Cambridge University Press, Cambridge, U.K.

de Lavalette, G. R. R. 1992. Strictness analysis via abstract interpretation for recursively defined types. *Inf. Comput. 99*, 154–177.

Deutsch, A. 1992. Operational models of programming languages and representations of relations on regular languages with application to the static determination of dynamic aliasing properties of data. Ph.D. thesis, Univ. Paris VI, Paris, France.

Ernoult, C. and Mycroft, A. 1991. Uniform ideals and strictness analysis. In *Proceedings of the 18th International Colloqium on Automata, Languages and Programming.* Lecture Notes in Computer Science, vol. 510. Springer Verlag, Berlin, 47–59.

Ferguson, A. and Hughes, J. 1989. An iterative powerdomain construction. In *Functional Programming. Proceedings of the 1989 Glasgow workshop*, K. Davis and J. Hughes, Eds. Springer Verlag, Berlin.

Filé, G. and Ranzato, F. 1994. Improving abstract interpretations by systematic lifting to the powerset. In *Logic Programming—Proceedings of the 1994 International Symposium*, M. Bruynooghe, Ed. The MIT Press, Cambridge, Mass., 655–669.

GUNTER, C. AND SCOTT, D. 1990. Semantic domains. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B, *Formal Models and Semantics*. North-Holland, Amsterdam.

HANKIN, C. AND LE MÉTAYER, D. 1994. Deriving algorithms from type inference systems: Applications to strictness analysis. In *Proceedings of the 21st Symposium on Principles of Programming Languages*. ACM Press, New York, 202–212.

HANKIN, C. AND LE MÉTAYER, D. 1995. Lazy type inference and program analysis. *Sci. Comput. Program. 25*, 219–249.

HECHT, M. S. 1977. *Flow Analysis of Computer Programs*. Programming Languages Series. North-Holland, New York.

HUNT, S. 1991. Abstract interpretation of functional languages: From theory to practice. Ph.D. thesis, Imperial College, Univ. of London, London, U.K.

HUNT, S. AND SANDS, D. 1991. Binding time analysis: A new PERspective. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics–Based Program Manipulation*. ACM Press, New York.

JENSEN, K. D., HJÆRESEN, P., AND ROSENDAHL, M. 1994. Efficient strictness analysis of Haskell. In *Proceedings of the 1st International Static Analysis Symposium (SAS'94)*, B. Le Charlier, Ed. Lecture Notes in Computer Science, vol. 864. Springer-Verlag, Berlin, 346–362.

JENSEN, T. 1991. Strictness analysis in logical form. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, J. Hughes, Ed. Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Berlin, 352–366.

JENSEN, T. 1992a. Abstract interpretation in logical form. Ph.D. thesis, Imperial College, Univ. of London, London, U.K.

JENSEN, T. 1992b. Axiomatising uniform properties of recursive data structures. In *Proceedings of the 2nd Bordeaux Workshop on Static Analysis*, M. Billaud, P. Castéran, M. Corsini, K. Musumbu, and A. Rauzy, Eds. Bigre no. 81–82. IRISA, Rennes, France.

JENSEN, T. 1992c. Disjunctive strictness analysis. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif.

JENSEN, T. 1994. Abstract interpretation over algebraic data types. In *Proceedings of the 5th IEEE International Conference on Computer Languages*, H. Bal, Ed. IEEE Press, New York.

JENSEN, T. 1995. Conjunctive type systems and abstract interpretation of higher-order functional programs. *J. Logic Comput. 5,* 4, 397–421.

JONES, N., GOMARD, C., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Series in Computer Science. Prentice-Hall International, Chichester, U.K.

JONES, N. AND MUCHNICK, S. 1981. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In *Program Flow Analysis: Theory and Applications*, N. Jones and S. Muchnick, Eds. Prentice-Hall, Englewood Cliffs, N.J.

JONES, N., SESTOFT, P., AND SØNDERGAARD, H. 1985. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications*, J. P. Jouannaud, Ed. Lecture Notes in Computer Science, vol. 202. Springer-Verlag, Berlin, 124–140.

JONES, N. D. AND MYCROFT, A. 1986. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*. ACM Press, New York, 296–306.

LAUNCHBURY, J. 1989. Projection factorisations in partial evaluation. Ph.D. thesis, Dept. of Computing, Univ. of Glasgow, Glasgow, Scotland.

LE MÉTAYER, D. 1995. Proving properties of programs defined over recursive data structures. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, W. Scherlis, Ed. ACM Press, New York, 88–99.

LEUNG, A. AND MISHRA, P. 1991. Reasoning about simple and exhaustive demand in higher–order lazy languages. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, J. Hughes, Ed. Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Berlin.

MOGENSEN, T. 1988. Partially static structures in a self-applicable partial evaluator. In *Partial Evaluation and Mixed Computation*, D. Bjørner, A. P. Ershov, and N. D. Jones, Eds. North-Holland, Amsterdam, 325–347.

Monsuez, B. 1995. Using abstract interpretation to define a strictness type inference system. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, W. Scherlis, Ed. ACM Press, New York, 122–133.

Muylaert-Filho, J. 1994. Abstract interpretation in continuation passing style. Ph.D. thesis, Imperial College, Univ. of London, London, U.K.

Nielson, F. 1982. A denotational framework for data flow analysis. *Acta Informatica 18*, 265–287.

Nielson, F. 1984. Abstract interpretation using domain theory. Ph.D. thesis, Univ. of Edinburgh, Edinburgh, Scotland.

Nielson, F. 1985. Tensor products generalize the relational data flow analysis method. In *Proceedings of the 4th Hungarian Computer Science Conference.* 211–225.

Nielson, F. and Nielson, H. R. 1994. The tensor product in Wadler's analysis of lists. *Sci. Comput. Program. 22*, 327–354.

Nielson, H. R. and Nielson, F. 1992. The tensor product in Wadler's analysis of lists. In *Proceedings of the 4th European Symposium on Programming*, B. Krieg–Brückner, Ed. Lecture Notes in Computer Science, vol. 582. Springer-Verlag, Berlin.

Plotkin, G. D. 1976. A powerdomain construction. *SIAM J. Comput. 5,* 3, 452–487.

Reynolds, J. C. 1974. On the relation between direct and continuation semantics. In *Proceedings of the 2nd International Conference on Algorithms, Languages and Programming.* Lecture Notes in Computer Science, vol. 14. Springer-Verlag, Berlin, 141–156.

Rosendahl, M. 1993. Higher-order chaotic iteration sequences. In *Proceedings of Programming Language Implementation and Logic Programming (PLILP'93)*, M. Bruynooghe and J. Penjam, Eds. Lecture Notes in Computer Science, vol. 714. Springer-Verlag, Berlin, 332–345.

Sintzoff, M. 1972. Calculating properties of programs by valuations on specific models. *Proceedings of the ACM Conference on Proving Assertions about Programs. SIGPLAN Notices 7,* 1, 203–207.

Wadler, P. 1987. Strictness analysis in non–flat domains (by abstract interpretation over finite domains). In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, Eds. Ellis Horwood, Chichester, U.K.

Wadler, P. and Hughes, R. J. M. 1987. Projections for strictness analysis. In *Proceedings of the Conference on Functional Programming and Computer Architecture.* Lecture Notes in Computer Science, vol. 274. Springer-Verlag, Berlin.

Winskel, G. 1983. Powerdomains and modality. In *Foundations of Computation Theory*, M. Karpinski, Ed. Lecture Notes in Computer Science, vol. 158. Springer-Verlag, Berlin.

Received October 1996; revised June 1997; accepted July 1997