ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 51 (2004) 153–196

Science of Computer Programming

www.elsevier.com/locate/scico

Safety of abstract interpretations for free, via logical relations and Galois connections

Kevin Backhouse[a,*,1], Roland Backhouse[b]

[a] *ARM Ltd., Cambridge CB1 9NJ, UK*

[b] *School of Computer Science and Information Technology, University of Nottingham, Nottingham NG8 1BB, UK*

Received 28 January 2003; received in revised form 30 May 2003; accepted 5 June 2003

Abstract

Algebraic properties of logical relations on partially ordered sets are studied. It is shown how to construct a logical relation that extends a collection of base Galois connections to a Galois connection of arbitrary higher-order type. "Theorems-for-free" is used to show that the construction ensures safe abstract interpretation of parametrically polymorphic functions. The properties are used to show how abstract interpretations of program libraries can be constructed.

Keywords: Abstract interpretation; Logical relations; Parametricity; Theorems for free; Galois connections

1. Introduction

Logical relations were introduced by Plotkin [22] and Reynolds [23] as a basis for reasoning about possible implementations of the polymorphic lambda calculus and its models. Later, Wadler [24] showed that Reynolds' "abstraction theorem" can be used to derive many useful properties of parametrically polymorphic functions from their types. This paper is about applying the algebraic properties of logical relations to constructing Galois connections of higher-order type.

The primary application of this work is to the construction of abstract interpretations [9,10]. Abstract interpretation is a technique for approximating the execution behaviour of a computer program. It is used, for example, for strictness analysis of functional

* Corresponding author.

E-mail addresses: kevin.backhouse@arm.com (K. Backhouse), roland.backhouse@nottingham.ac.uk (R. Backhouse).

[1] This work was done whilst at the Computing Laboratory, University of Oxford.

0167-6423/$ - see front matter
doi:10.1016/j.scico.2003.06.002

programs; the results of such an analysis are approximate in the sense that non-strict programs will always be correctly identified as such but strict programs may be incorrectly classified. Consequently, optimisations based on identifying strictness will only be applied to strict programs; in this sense, the approximation is "safe".

The paper begins, in Section 3, with a review of the basic algebraic properties of Reynolds' arrow operator on relations. Proofs are omitted in this section because the results are known. Section 4 contains the main results of the paper. The essential ideas have already been observed by Abramsky [1]; our contribution is to specialise his "uniformisation theorem" to Galois-connected functions. This enables us to give a concise calculational formulation of the construction of a Galois connection of any given type, given a collection of Galois connections on the base types.

In the literature on abstract interpretations, it would appear that there is incomplete understanding of the relevance of logical relations—Cousot and Cousot [11] claim that their guidelines for designing abstract interpretations exhibit "a definite advantage of the Galois connection approach to abstract interpretations over its variant formalisation using logical relations". We show, however, that the construction of Galois connections of higher-order type is entirely equivalent to the construction of a logical relation.

The most important aspect of our results is that they suggest a design principle for the development of programs involving a separation into generic and non-generic components. We provide two examples. Section 6 shows how the semantics of CSP traces can be parameterised to allow two interpretations: the standard semantics, and an abstract interpretation that simply evaluates the alphabet of a process. Section 7 considers a classic example of the use of abstract interpretation, so-called "strictness analysis". Further examples are discussed in [3].

2. Running example

We start by introducing a simple, running example. Consider the *fold* function which, given a starting value m of some type a, a binary operator $\oslash$ of type $a \leftarrow a \times a$ and a list $[m_o, \ldots, m_{n-1}]$ of a's, evaluates $((m \oslash m_o) \oslash \ldots) \oslash m_{n-1}$. So, $fold.0.(+).ms$ sums a list ms of integer values, $fold.1.(*).ms$ evaluates the product of the list of values,[2] $fold.true.(\equiv).bs$ evaluates the associative (as opposed to conjunctional) equivalence of the list of booleans bs, and $fold.false.(\vee).bs$ evaluates whether one of the values in the list bs is true.

Now, suppose we want to determine whether some *fold* on a list of integer values evaluates to a number that is even without actually evaluating the *fold* computation proper. For example, we may want to determine whether

$$18975 * 3 * 2 * 21$$

is even, without performing the integer multiplication. "Theorems-for-free" [24] predicts that this can be done by evaluating

$$even.18975 \vee even.3 \vee even.2 \vee even.21.$$

[2] The symbol "$*$" is used to denote multiplication in order to distinguish it from "$\times$", which is used to mean cartesian product.

Formally, the "free theorem" for *fold* predicts that

$$even.(fold.1.(*).ms) = fold.false.(\vee).(List.even.ms)$$

where $List.even.ms$ maps the function *even* over the list of numbers ms.[3] Similarly, the "free theorem" also predicts that[4]

$$even.(18975 + 3 + 2 + 21) = (even.18975 \equiv even.3 \equiv even.2 \equiv even.21).$$

Formally,

$$even.(fold.0.(+).ms) = fold.true.(\equiv).(List.even.ms).$$

That is, a sum of numbers is even if the (associative) equivalence of their individual parities is even.

That these properties of *fold* are applications of theorems-for-free is, of course, well-known. (For *fold* the general theorem is often called the *fusion* theorem.)

Suppose now that we want to determine whether a product of numbers is divisible by k, for some given positive number k. For brevity, define the function dk mapping integers to booleans by

$$dk.m = m/k$$

where "m/k" is read as "m is divisible by k". We show that theorems-for-free predicts that

$$dk.(fold.1.(*).ms) \Leftarrow fold.(k = 1).(\otimes).(List.dk.ms)$$

where

$$b \otimes c \;=\; (k = 1) \vee b \vee c.$$

That is, the product of a list of numbers is divisible by k if $k = 1$ or one of the numbers is divisible by k.

Note that the "free theorem" here is an "if" ($\Leftarrow$) rather than an equality. We say that $fold.(k = 1).(\otimes).(List.dk.ms)$ is an *abstract interpretation* of $dk.(fold.1.(*).ms)$ and its result is *safe* in the sense that, if the abstract interpretation evaluates to *true*, it is the case that the given product is divisible by k; however, if the abstract interpretation evaluates to *false*, it is not known whether or not the given product is divisible by k. For example, taking k to be 4 (the smallest non-prime positive number), the above abstract interpretation of $2 * 2$ evaluates to *false* whereas $2 * 2$ is of course divisible by 4.

Our running example is *deliberately* simple. The reader will have no difficulty in seeing how to evaluate divisibility by k, for arbitrary k, but that is not the point of the

[3] *List* is normally denoted by *map* in functional languages. As we want to generalise *List* to an arbitrary datatype F, it is useful to adopt the categorical convention of using the same symbol to denote the type constructor and its associated map operation.

[4] The right side of this equation should be read associatively and not conjunctionally. See [12] for an excellent discussion of the associativity of equivalence.

example. Abstract interpretations are typically concerned with computations that are so complex, or perhaps even impossible, that a safe result is all that one can expect to achieve. Moreover, we are concerned here with a very general result; the example we have chosen—the combination of the function *fold* with the test for divisibility by k—combines the two basic ingredients of this general result, namely "theorems-for-free" and Galois connections. Let us now introduce the basics needed to understand these concepts.

3. Preliminaries

This section introduces the basic notions for future reference.

Types: For the purposes of this paper, we assume that types are partially ordered sets (abbreviated poset). Supposing that $(A, \sqsubseteq)$ and $(B, \preccurlyeq)$ are posets, $(A, \sqsubseteq) \leftarrow (B, \preccurlyeq)$ denotes the set of *monotonic* functions with range A and domain B, ordered pointwise.

Relation algebra: A *binary relation of type* $A \sim B$ is a subset of $A \times B$. Given binary relations $R \in A \sim B$ and $S \in B \sim C$, $R \bullet S$ denotes their composition, the relation of type $A \sim C$ defined by

$$(x,z) \in R \bullet S \quad \equiv \quad \langle \exists y :: (x,y) \in R \wedge (y,z) \in S \rangle. \tag{1}$$

We use $^\cup$ (pronounced "wok") as a postfix operator to denote the converse operation on relations. So, if $R \in A \sim B$, $R^\cup$ is the relation of type $B \sim A$ defined by

$$(x,y) \in R^\cup \quad \equiv \quad (y,x) \in R. \tag{2}$$

Shunting of functions: Functions are very special sorts of relations because they enjoy a number of properties not generally enjoyed by relations proper. In point-free relation algebra, functions are characterized by a so-called *shunting* rule. The rule is that, relation f is a (total) *function* if and only if for all relations R and S,

$$f \bullet R \subseteq S \quad \equiv \quad R \subseteq f^\cup \bullet S.$$

(Strictly we should specify the types of f, R and S. These details are omitted.) Note that, essentially by taking the converse of both sides of this equivalence, we also have, for all relations R and S (of appropriate type),

$$R \bullet f^\cup \subseteq S \quad \equiv \quad R \subseteq S \bullet f.$$

An alternative, equivalent, characterisation of total functions is the following. The relation f of type $A \leftarrow B$ is a total function if and only if it is *simple*:

$$f \bullet f^\cup \subseteq \mathsf{id}_A,$$

and *entire*:

$$\mathsf{id}_B \subseteq f^\cup \bullet f,$$

where id_A and id_B denote the identity relations on the sets A and B, respectively. (It is easy to see that simplicity and entirety follow from the shunting rules. That the shunting

rules follow from simplicity and entirety is an easy application of the monotonicity of relational composition.)

If f is a function, we denote by $f.y$ the unique value x such that $(x, y) \in f$. That is, for function f of type $A \sim B$ we have, for all x in A and all y in B,

$$(x, y) \in f \quad \equiv \quad x = f.y.$$

(Given $y \in B$ the existence of $x \in A$ such that $(x, y) \in f$ is guaranteed by the entirety of f and its uniqueness is guaranteed by the simplicity of f.)

Galois connections and pair algebras: Galois connections are most often defined in terms of a pair of functions. A better starting point for a discussion of Galois connections is, arguably, relations—or so-called "pair algebras", as proposed by Hartmanis and Stearns [13,14].

A binary relation R on the posets $(A, \sqsubseteq)$ and $(B, \preccurlyeq)$ is called a *pair algebra* if there are functions $f \in A \leftarrow B$ and $g \in B \leftarrow A$ such that, for all x in B and all y in A,

$$(x, y) \in R \quad \equiv \quad f.x \sqsubseteq y$$

and

$$(x, y) \in R \quad \equiv \quad x \preccurlyeq g.y.$$

We say that the pair of functions (f, g) is a *Galois connection between the posets* $(A, \sqsubseteq)$ *and* $(B, \preccurlyeq)$ if $f \in A \leftarrow B$ and $g \in B \leftarrow A$ and, for all x in B and all y in A,

$$f.x \sqsubseteq y \quad \equiv \quad x \preccurlyeq g.y.$$

The two functions f and g are said to be the *lower* and *upper adjoints* (respectively).

Clearly a Galois connection defines a pair algebra, and a pair algebra defines a Galois connection. A major element of the proofs in this paper is that we instantiate "theorems-for-free" with the pair algebra defined by a Galois connection.

The Galois connection in our running example is the pair (dk, kd) between $(\mathsf{Bool}, \Leftarrow)$ and $(\mathsf{PosInt}, /)$ where $dk \in \mathsf{Bool} \leftarrow \mathsf{PosInt}$, $kd \in \mathsf{PosInt} \leftarrow \mathsf{Bool}$ and $/$ is the is-divisible-by ordering on positive integers. Specifically, kd is defined by, for all booleans b,

$$kd.b = \textbf{if } b \textbf{ then } k \textbf{ else } 1$$

and satisfies, for all positive integers m and all booleans b,

$$dk.m \Leftarrow b \quad \equiv \quad m/kd.b.$$

(Read "(m is divisible by k if b) is (m is divisible by $kd.b$)".) The pair algebra corresponding to this Galois connection relates all positive integers to the boolean *false* and the positive integers divisible by k to the boolean *true*.

For effective calculation, it helps to express all definitions and properties in "point-free" form. A point-free definition of a pair algebra eliminates reference to the "points" x and y in the definition above. Specifically, a binary relation R on the posets $(A, \sqsubseteq)$

and $(B, \preccurlyeq)$ is called a *pair algebra* if there are functions $f \in A \leftarrow B$ and $g \in B \leftarrow A$ such that

$$f^{\cup} \bullet \sqsubseteq \quad = \quad R \quad = \quad \preccurlyeq \bullet g.$$

It is this definition of a pair algebra that we use in the sequel. (For those unfamiliar with point-free relation calculus, it helps to remember the rule

$$(x, y) \in h^{\cup} \bullet R \bullet k \quad \equiv \quad (h.x,\ k.y) \in R$$

for all x, y, functions h and k, and relations R. We sometimes write the right side of this equivalence as

$$h.x \ \ R \ \ k.y,$$

particularly when R is an ordering relation. In this way, we extract the relation $f^{\cup} \bullet \sqsubseteq$ from the expression $f.x \sqsubseteq y$, and the relation $\preccurlyeq \bullet g$ from the expression $x \preccurlyeq g.y$.)

An alternative characterization of Galois connections is in terms of *cancellation* properties. The pair of functions (f, g) between $(A, \sqsubseteq)$ and $(B, \preccurlyeq)$ is Galois connected if both functions are monotonic and, for all x in B and all y in A, $x \preccurlyeq g.(f.x)$ and $f.(g.y) \sqsubseteq y$. Expressed in point-free relation algebra, these are the rules:

$$g \bullet \sqsubseteq \quad \subseteq \quad \preccurlyeq \bullet g,$$

$$f \bullet \preccurlyeq \quad \subseteq \quad \sqsubseteq \bullet f,$$

$$\preccurlyeq \quad \subseteq \quad \preccurlyeq \bullet g \bullet f$$

and

$$f \bullet g \bullet \sqsubseteq \quad \subseteq \quad \sqsubseteq.$$

The first two properties are the monotonicity properties, and the second two are the cancellation properties.

Theorems for free: Based on Reynolds' abstraction theorem [23], Wadler [24] showed how to derive a theorem about a polymorphic function from its type. The key to such "free theorems" is, in Wadler's words, "that types may be read as relations". Briefly, the type A is read as the identity relation id_A on A and the function space constructor "$\leftarrow$" is read as a mapping from a pair of relations R and S of types $A \sim B$ and $C \sim D$, respectively, to a binary relation $R \leftarrow S$ on functions f and g of type $A \leftarrow C$ and $B \leftarrow D$, respectively. Formally, suppose R and S are binary relations of type $A \sim B$ and $C \sim D$, respectively. Then $R \leftarrow S$ is the binary relation of type $(A \leftarrow C) \sim (B \leftarrow D)$ defined by, for all functions $f \in A \leftarrow C$ and $g \in B \leftarrow D$,

$$(f, g) \in R \leftarrow S \quad \equiv \quad \langle \forall x, y :: (f.x, g.y) \in R \ \Leftarrow \ (x, y) \in S \rangle. \tag{3}$$

In words, f and g construct R-related values from S-related values.

As an example, suppose $(A, \sqsubseteq)$ and $(B, \preccurlyeq)$ are posets. Then we can instantiate R to $\sqsubseteq$ and S to $\preccurlyeq$ getting a relation $\sqsubseteq \leftarrow \preccurlyeq$ between functions f and g of type $A \leftarrow B$. In

particular, switching to the usual infix notation for membership of an ordering relation,

$$(f,f) \in \sqsubseteq \leftarrow \preccurlyeq \quad \equiv \quad \langle \forall x, y :: f.x \sqsubseteq f.y \Leftarrow x \preccurlyeq y \rangle.$$

So, $(f,f) \in \sqsubseteq \leftarrow \preccurlyeq$ is the statement that f is a monotonic function. In Wadler's words, f maps $\preccurlyeq$-related values to $\sqsubseteq$-related values.

Particularly relevant to this paper is that, if f and g are both *monotonic* functions of type $A \leftarrow B$ where $(A, \sqsubseteq)$ and $(B, \preccurlyeq)$ are posets,

$$(f,g) \in \sqsubseteq \leftarrow \preccurlyeq \quad \equiv \quad \langle \forall x :: f.x \sqsubseteq g.x \rangle.$$

(The easy proof is left to the reader.) We thus recognize $\sqsubseteq \leftarrow \preccurlyeq$ as the usual pointwise ordering on monotonic functions of type $A \leftarrow B$. Equivalently, in point-free relation algebra, if the domain of $\leftarrow$ is restricted to monotonic functions,

$$\sqsubseteq \leftarrow \preccurlyeq \quad = \quad \sqsubseteq \leftarrow \mathsf{id}_B, \tag{4}$$

where id_B denotes the identity relation on B. See Lemma 45 for the generalisation of this property to higher order types.

"Theorems-for-free" is the property that, if θ is a parametrically polymorphic function of type t, where t is a type expression parameterised by type variables $a, \ldots, c$, and, for each type variable a, R_a is a relation, then $(\theta, \theta) \in t[a, \ldots, c := R_a, \ldots, R_c]$.

An example of a "free theorem" is obtained by considering the composition operator on functions. Composition has parametric type

$$(a \leftarrow b) \leftarrow (a \leftarrow c) \leftarrow (c \leftarrow b),$$

for all types a, b and c. The free theorem is that, for all relations R, S and T,

$$(\bullet, \bullet) \in (R \leftarrow S) \leftarrow (R \leftarrow T) \leftarrow (T \leftarrow S).$$

In particular, substituting orderings $\sqsubseteq$, $\preccurlyeq$ and $\leqslant$ for R, S and T, respectively,

$$(\bullet, \bullet) \in (\sqsubseteq \leftarrow \preccurlyeq) \leftarrow (\sqsubseteq \leftarrow \leqslant) \leftarrow (\leqslant \leftarrow \preccurlyeq).$$

Spelling out the definition of the arrow operator, this has the corollary that

$$(f \bullet g, f \bullet g) \in \sqsubseteq \leftarrow \preccurlyeq \quad \Leftarrow \quad (f,f) \in \sqsubseteq \leftarrow \leqslant \ \wedge \ (g,g) \in \leqslant \leftarrow \preccurlyeq .$$

In words, $f \bullet g$ is a monotonic function if both f and g are monotonic. Note that this property is derived solely from the type of function composition. So, "theorems-for-free" says that any parametrically polymorphic function that has the same type as function composition will preserve the monotonicity property. For numerous additional examples of "free" theorems, see [24].

Note that we should distinguish between the different instances of θ. The free theorem is actually that the pair $(\theta(A), \theta(B))$ is an element of relation $t[a, \ldots, c := R_a, \ldots, R_c]$, where A and B indicate how the particular instances of θ are determined (depending on the types of the relations R). We omit these details for the moment but include them later.

An instance of $R \leftarrow S$, that is used extensively in this paper, is when R is a function and S is the converse of a function. Suppose $(A, \sqsubseteq)$, $(B, \preccurlyeq)$, $(C, \trianglelefteq)$ and $(D, \leqslant)$ are posets. Then, for all monotonic functions f of type $(A, \sqsubseteq) \leftarrow (B, \preccurlyeq)$ and g of type $(C, \trianglelefteq) \leftarrow (D, \leqslant)$, $f \leftarrow g^{\cup}$ is the monotonic function of type $(A \leftarrow D) \leftarrow (B \leftarrow C)$ defined by

$$(f \leftarrow g^{\cup}).h = f \bullet h \bullet g. \tag{5}$$

(The easy proof is left to the reader.)

Properties: Throughout this paper, it is much more effective to use a point-free definition of the arrow operator. Also, as mentioned earlier, types will be assumed to be posets. Specifically, suppose R and S are binary relations of type $A \sim B$ and $C \sim D$, respectively. Suppose, also, that $(A, \sqsubseteq)$, $(B, \preccurlyeq)$, $(C, \trianglelefteq)$ and $(D, \leqslant)$ are posets. Then $R \leftarrow S$ is the binary relation of type

$$((A, \sqsubseteq) \leftarrow (C, \trianglelefteq)) \sim ((B, \preccurlyeq) \leftarrow (D, \leqslant))$$

defined by, for all (monotonic) functions $f \in A \leftarrow C$ and $g \in B \leftarrow D$,

$$(f, g) \in R \leftarrow S \quad \equiv \quad f \bullet S \subseteq R \bullet g. \tag{6}$$

(The advantage of this formulation over the pointwise definition (3) is that it eliminates the universal quantification as well as the bound variables, x and y.)

The following properties of the arrow operator are easily derived from its definition [4]. For all relations R and S,

$$(R \leftarrow S)^{\cup} \;=\; R^{\cup} \leftarrow S^{\cup}, \tag{7}$$

$$(R \leftarrow T) \bullet (S \leftarrow U) \;\subseteq\; (R \bullet S) \leftarrow (T \bullet U). \tag{8}$$

The arrow operator is also monotonic in its left argument and anti-monotonic in its right argument. That is

$$R \leftarrow S \subseteq T \leftarrow U \quad \Leftarrow \quad R \subseteq T \wedge S \supseteq U. \tag{9}$$

Note carefully that (8) expresses an inclusion. To see that the inclusion cannot be strengthened to an equality, let $\amalg_{X,Y}$ denote the empty relation on sets X and Y (that is the empty subset of $X \times Y$). Also, let $\top\!\!\top_{X,Y}$ denote the universal relation on sets X and Y (that is the set $X \times Y$). Then, it suffices to note that

$$\amalg_{A,B} \leftarrow \amalg_{C,D} = \top\!\!\top_{(A \leftarrow C),(B \leftarrow D)}$$

whereas, for any *non-empty* relation X on the sets C and D,

$$\amalg_{A,B} \leftarrow X = \amalg_{(A \leftarrow C),(B \leftarrow D)}.$$

So, for all T and U, of types $D \sim E$ and $E \sim F$, respectively,

$$
\begin{array}{ll}
 & (\amalg_{A,B} \leftarrow T) \bullet (\amalg_{B,C} \leftarrow U) \\
= & \quad \{\text{choose } \textit{non-empty } T \text{ and } U\} \\
 & \amalg_{(A \leftarrow D),(B \leftarrow E)} \bullet \amalg_{(B \leftarrow E),(C \leftarrow F)} \\
= & \quad \{\text{composition of empty relations is empty}\} \\
 & \amalg_{(A \leftarrow D),(C \leftarrow F)} \\
\subsetneq & \quad \{\text{choose types } A \text{ and } C \text{ to be non-empty}\} \\
 & \top\!\!\top_{(A \leftarrow D),(C \leftarrow F)} \\
= & \quad \{\text{choose } T \text{ and } U \text{ so that } T \bullet U \text{ is } \textit{empty}\} \\
 & (\amalg_{A,B} \bullet \amalg_{B,C}) \leftarrow (T \bullet U).
\end{array}
$$

In words, a counterexample is obtained by choosing T and U to be both non-empty whilst their composition $T \bullet U$ is empty.

Property (8) can be strengthened to an equality by restricting particular instances of the relations R, S, T and U to be monotonic functions. Suppose $(A, \sqsubseteq)$, $(B, \preccurlyeq)$, $(C, \trianglelefteq)$ and $(D, \leqslant)$ are posets, and $f \in A \leftarrow C$ and $g \in B \leftarrow D$ are monotonic functions. Then, for all relations R and S of appropriate type,

$$(R \bullet f) \leftarrow (S \bullet g^{\cup}) \;=\; (R \leftarrow S) \bullet (f \leftarrow g^{\cup}). \tag{10}$$

Dually,

$$(f^{\cup} \bullet R) \leftarrow (g \bullet S) \;=\; (f^{\cup} \leftarrow g) \bullet (R \leftarrow S). \tag{11}$$

We refer to (7), (10) and (11) as the *distributivity* properties of $\leftarrow$. Note carefully that (10) and (11) require f and g to be functions.

Using id_X to denote the identity relation on set X, we also assume the so-called *identity axiom*:

$$\mathsf{id}_A \leftarrow \mathsf{id}_B = \mathsf{id}_{A \leftarrow B}. \tag{12}$$

From now on, we assume that composition has precedence over the arrow operator (so that the parentheses can be omitted in the lhs of (10) but not in the rhs).

Relators: Another element of "theorems-for-free" is that type constructors, like disjoint sum $+$, cartesian product $\times$ and *List* have to be extended to map relations to relations. This led the second author to propose the notion of a *relator* as the basis for a relational theory of datatypes [5,6]. Briefly, a relator is a monotonic functor that commutes with converse.[5] Formally, a relator, F, is a pair of mappings from a source allegory to a target allegory. The first element of the pair is from objects (types) of the source allegory to objects of the target allegory and the second element is from morphisms (relations) of the source allegory to morphisms of the target allegory. The mappings are required to have the properties that they form a functor from the

[5] Readers unfamiliar with category or allegory theory can safely assume that a relator is a monotonic function from relations to relations that preserves identities, composition and converse. The formal definition is more general.

underlying source category to the underlying target category. This means that a relator, F, preserves composition: for all (composable) morphisms R and S,

$$F.(R \bullet S) = F.R \bullet F.S,$$

and preserve identities: for all objects A,

$$F.\mathsf{id}_A = \mathsf{id}_{F.A}.$$

Relators are, in addition, monotonic: for all morphisms R and S,

$$F.R \subseteq F.S \;\Leftarrow\; R \subseteq S,$$

and preserve converse:

$$(F.R)^{\cup} = F.(R^{\cup}).$$

An example of a unary relator is *List*. Suppose R relates values of type A to values of type B. Then *List*.R relates values of type *List*.A to values of type *List*.B. Specifically, the list *as* is related to the list *bs* by *List*.R if *as* and *bs* have the same length and corresponding elements are related by R. An example of a binary relator is cartesian product. Suppose R relates values of type A to values of type B, and S relates values of type C to values of type D. Then $R \times S$ relates values of type $A \times C$ to values of type $B \times D$. Specifically, the pair (a, c) is related to the pair (b, d) by $R \times S$ if a is related by R to b, and c is related by S to d. An example of a zero-ary relator is $\mathbb{N}$; this relates two natural numbers exactly when they are equal. (Any type defines a zero-ary relator in this way.)

By design, it is easy to show that relators preserve functions. That is, if f is a function then $F.f$ is also a function.

Note that the formal definition of a relator encompasses the possibility of relators with different source and target allegories. We will be somewhat informal in our treatment of relators. Typically, we assume that the source allegory is the n-fold product of the target allegory with itself, for some unspecified number n ($n \geqslant 0$) and will write $F.(u \ldots v)$ for the application of the relator to arguments $u, \ldots, v$.

In this paper, we also require that relators preserve partial orderings (so that if R is reflexive, antisymmetric and transitive then so too is $F.R$). For this we need the additional property that relators distribute through binary intersections. That is, we assume that, for all relations R and S, $F.(R \cap S) = F.R \cap F.S$. This assumption is satisfied by all "regular" datatypes [16]. Indeed, it is easily verified for disjoint sum and cartesian product and then a straightforward proof shows that the property is preserved by the usual fixpoint construction of datatypes like *List*. So, the assumption is a reasonable one in the context of modern programming languages. Formally, we have:

Lemma 13. *Let $(A, \leqslant)$ be a poset. Suppose F is a relator that preserves binary intersections (i.e. for all relations R and S, $F.(R \cap S) = F.R \cap F.S$). Then $(F.A, F.\leqslant)$ is a poset.*

Proof. It is easy to verify that transitivity and reflexivity of $F.\leqslant$ follows from transitivity and reflexivity of $\leqslant$. (For transitivity, use that relators distribute through composition; for reflexivity, that relators preserve identities.) Antisymmetry of $F.\leqslant$ is the property that

$$F.\leqslant \cap (F.\leqslant)^{\cup} \subseteq \mathsf{id}_{F.A}.$$

This follows from the antisymmetry of $\leqslant$ ($\leqslant \cap \leqslant^{\cup} \subseteq \mathsf{id}_A$) and that F preserves identities, commutes with converse and is monotonic, and the assumption that F preserves binary intersections. □

Our use of relators generalises and unifies a number of techniques discussed in the literature on abstract interpretation. For example, what is known as the "independent attribute method" [19, p. 247] is a special case of our theorems in which the relator F is the product relator.

Running example: Returning to our running example, the function *fold* has type

$$a \leftarrow List.a \leftarrow (a \leftarrow a \times a) \leftarrow a$$

for all instances of the type variable a. (Actually, it has a more general type, but this is all we need for the purposes of the example.) Theorems-for-free predicts that, for arbitrary relation P

$$(fold, fold) \in P \leftarrow List.P \leftarrow (P \leftarrow P \times P) \leftarrow P.$$

In particular, we can take for P the pair algebra of the Galois connection (dk, kd) mentioned above.

Theorem 40 predicts, first, that

$$dk \leftarrow List.kd^{\cup} \leftarrow (kd^{\cup} \leftarrow dk \times dk) \leftarrow kd^{\cup}$$

is the lower adjoint in a Galois connection between the poset of monotonic functions of type

$$\mathsf{Bool} \leftarrow List.\mathsf{Bool} \leftarrow (\mathsf{Bool} \leftarrow \mathsf{Bool} \times \mathsf{Bool}) \leftarrow \mathsf{Bool}$$

(ordered pointwise by $\Leftarrow$) and monotonic functions of type

$$\mathsf{PosInt} \leftarrow List.\mathsf{PosInt} \leftarrow (\mathsf{PosInt} \leftarrow \mathsf{PosInt} \times \mathsf{PosInt}) \leftarrow \mathsf{PosInt}$$

(ordered pointwise by divisibility). The upper adjoint is

$$kd \leftarrow List.dk^{\cup} \leftarrow (dk^{\cup} \leftarrow kd \times kd) \leftarrow dk^{\cup}.$$

Our second theorem (Theorem 41) says that the free theorem for *fold* instantiated with the pair algebra for the (dk, kd) Galois connection has the corollary (Corollary 47) that, for all binary operators $\otimes \in \mathsf{PosInt} \leftarrow \mathsf{PosInt} \times \mathsf{PosInt}$ that are monotonic with respect to divisibility, all $m \in \mathsf{PosInt}$ and $ms \in List.\mathsf{PosInt}$,

$$dk.ConcreteValue \Leftarrow AbstractValue$$

where

$$ConcreteValue = fold.m.(\otimes).ms$$

and

$$AbstractValue = fold.(dk.m).((dk \leftarrow kd^{\cup} \times kd^{\cup}).(\otimes)).(List.dk.ms).$$

Substituting integer multiplication for $\otimes$ we calculate that, for all boolean values b and c,

$$\begin{array}{ll} & (dk \leftarrow kd^{\cup} \times kd^{\cup}).(*).(b,c) \\ = & \quad \{\text{definitions}\} \\ & dk.(kd.b{*}kd.c) \\ = & \quad \{kd.true = k, kd.false = 1, \\ & \quad\ dk.(1*1) = (k=1), \\ & \quad\ dk.(1*k) = dk.(k*1) = dk.(k*k) = true\} \\ & (k=1) \vee b \vee c. \end{array}$$

In this way, we have calculated[6] a safe abstract interpretation determining whether a sequence of integer multiplications results in a value divisible by k.

Let us now present the theorems formally.

4. Extending Galois connections

In this section, we define an operator that extends mappings on type variables to mappings on type expressions. (See definitions (16), (17) and (18).) The operator is a key ingredient in the construction of Galois connections of higher-order type. (See Lemma 25 and Theorem 40.) Logical relations, as introduced by Plotkin and Reynolds, are a special case.

4.1. Types and assignments to variables

We consider the following (extended BNF) grammar of type expressions:

$$Exp \quad ::= \quad (Relator.Exp^*) \mid (Exp \leftarrow Exp) \mid Variable,$$

where *Variable* and *Relator* are (disjoint) finite sets. Variables act as place-holders for types and relations, as explained in detail later. (A common name for them is "polymorphic type variable".) Each element of *Relator* denotes a relator of a certain arity.

According to the syntax above, the application of a binary relator, say $+$, to types t and u would be denoted by the prefix notation $+tu$. We will, however, deviate from the formal syntax in our examples and write $t + u$ instead. Also, the formal syntax

[6] Note how the disjunct "$k = 1$" emerges from the calculation. Trying to cut corners, by omitting the calculation, is likely to lead to this disjunct being forgotten.

stipulates that subtypes are parenthesised (in order to avoid specifying precedence and associativity rules); again we will ignore this requirement when presenting examples. Instead we assume that $\leftarrow$ has the lowest precedence and associates to the left.

An example of a type expression is

$$List.a \leftarrow (Bool \leftarrow a) \times List.a.$$

This uses the unary relator *List*, binary product relator, $\times$, and the zero-ary relator *Bool*. It describes the type of the (uncurried) *filter* function on lists, which is parametrically polymorphic in the variable a.

Polymorphism is defined in terms of *substitutions* [17], which map type variables to types. Theorems-for-free [23,24] is similarly defined by mapping type variables to relations. In order to encompass both in one definition, we introduce the general notion of a "variable assignment" and an operator which performs the substitution.

A *variable assignment* is a function with domain *Variable*, the set of type variables. We consider two kinds of variable assignments, one with range posets and the other with range (binary) relations on posets. A variable assignment will be denoted by an assignment statement as in, for example,

$$a, b := integer, boolean.$$

Given a variable assignment V, its application to type variable a is denoted by V_a.

In the case of a variable assignment V such that V_a is a relation, we need to define the converse and composition operators. The definitions are:

$$(V^{\cup})_a = (V_a)^{\cup} \tag{14}$$

and

$$(V \bullet W)_a = V_a \bullet W_a \cdot \tag{15}$$

We extend variable assignments to type expressions. The extension uses two variable assignments, one for the positive and the other for the negative occurrences of type variables. Specifically, suppose V and W are two variable assignments of the same kind (that is, both to posets, or both to relations). Then the *assignment* $[V, W]$ is defined inductively as follows:

$$[V, W]_a = V_a, \tag{16}$$

$$[V, W]_{u \leftarrow v} = [V, W]_u \leftarrow [W, V]_v, \tag{17}$$

$$[V, W]_{F.(u \ldots v)} = F.([V, W]_u \ldots [V, W]_v). \tag{18}$$

For example,

$$\begin{array}{ll} & [(a := R), (a := S)]_{a \leftarrow List.a \leftarrow (a \leftarrow a \times a) \leftarrow a} \\ = & R \leftarrow List.S \leftarrow (S \leftarrow R \times R) \leftarrow S. \end{array}$$

Note that these definitions make sense whenever V and W are both assignments of posets, or assignments of relations, to the type variables. In the case that V and W

assign posets to the type variables, we define the arrow operator to map posets A and B into the poset of *monotonic* functions mapping values of poset B into values of poset A, ordered pointwise. Correspondingly, in the case that V and W assign relations to the type variables, we restrict the arrow operator to relate *monotonic* functions of the appropriate type. (Formally, we need Lemma 24—see below—to make precise which posets and orderings are meant in this statement.)

The requirement that the arrow operator relates monotonic functions is essential to the main results in this paper although many of its properties do not depend on this requirement.

A special case is when the variable assignments V and W are the same. For brevity, we use the notation V_t in this case. So, by definition,

$$V_t = [V, V]_t. \tag{19}$$

Care needs to be taken with this shorthand because it is not the case that $(V \bullet W)_t$ equals $V_t \bullet W_t$ for all type expressions t. Property (15) is only valid for type variables a.

It is easy to establish that

$$([V, W]_t)^{\cup} = [V^{\cup}, W^{\cup}]_t. \tag{20}$$

In particular,

$$(V_t)^{\cup} = (V^{\cup})_t. \tag{21}$$

Because of (21), we write $V_t^{\cup}$, omitting the parentheses. We also write $[V, W]_t^{\cup}$; here the justification is that we define $[V, W]^{\cup}$ to be $[V^{\cup}, W^{\cup}]$.

Two other obvious properties are: for all type expressions u, v

$$V_{u \leftarrow v} = V_u \leftarrow V_v \tag{22}$$

and, for all type expressions $u, \ldots, v$,

$$V_{F.(u \ldots v)} = F.(V_u \ldots V_v). \tag{23}$$

In the case that V assigns relations to the type variables, the function mapping type expression t to V_t is called a *logical relation*.[7]

4.2. Basic constructions

This section catalogues a number of properties of assignments. Many of the results extend properties discussed in earlier sections to arbitrary type expressions. For example, Lemma 24 shows how to construct a poset for arbitrary types. Similarly, Lemma 25 extends (5) and Lemma 27 extends (9).

[7] The standard definition of a logical relation is a bit more general: a logical relation is a family of relations indexed by types such that properties (22) and (23) hold for all types $u, \ldots, v$. Clearly, a logical relation in our sense uniquely defines a logical relation according to the standard definition.

Lemma 24. *Let t be a type expression and let $\sqsubseteq$ and $\preccurlyeq$ assign partial ordering relations to the type variables in t. Let A and B be the corresponding poset assignments, respectively. (So, for each type variable a, $\sqsubseteq_a$ is a partial ordering relation on A_a and $\preccurlyeq_a$ is a partial ordering relation on B_a.) Then $[\sqsubseteq, \preccurlyeq]_t$ is a partial ordering on $[A, B]_t$. As a corollary, $\sqsubseteq_t$ is a partial ordering relation on A_t.*

Proof. Straightforward induction on the structure of type expressions using Lemma 13 for the case that t is a relator application. (Reflexivity in the case of the arrow operator is by definition: it is the requirement mentioned above that if A and B are posets then $A \leftarrow B$ is the set of *monotonic* functions with target A and source B.) Property (19) is then used to derive the special case. □

Applied to our running example, Lemma 24 states that the monotonic functions of type

$$\mathsf{Bool} \leftarrow \mathit{List}.\mathsf{Bool} \leftarrow (\mathsf{Bool} \leftarrow \mathsf{Bool} \times \mathsf{Bool}) \leftarrow \mathsf{Bool}$$

are partially ordered by the relation

$$(\Leftarrow) \leftarrow \mathit{List}.(\Leftarrow) \leftarrow ((\Leftarrow) \leftarrow (\Leftarrow) \times (\Leftarrow)) \leftarrow (\Leftarrow)$$

and monotonic functions of type

$$\mathsf{PosInt} \leftarrow \mathit{List}.\mathsf{PosInt} \leftarrow (\mathsf{PosInt} \leftarrow \mathsf{PosInt} \times \mathsf{PosInt}) \leftarrow \mathsf{PosInt}$$

are partially ordered by

$$(/) \leftarrow \mathit{List}.(/) \leftarrow ((/) \leftarrow (/) \times (/)) \leftarrow (/).$$

Later (Lemma 45), we see that these are just the pointwise orderings on the two function spaces.

We remarked earlier that, if f is a total function of type $A \leftarrow B$ and g is a total function of type $C \leftarrow D$, then $f \leftarrow g^{\cup}$ is also a total function of type $(A \leftarrow D) \leftarrow (B \leftarrow C)$. The following lemma generalises this property to arbitrary type expressions.

Lemma 25. *Suppose f and g are assignments such that, for all type variables a, f_a and g_a are total functions of type $A_a \leftarrow B_a$ and $C_a \leftarrow D_a$, respectively. Then, for all type expressions t, the assignment $[f, g^{\cup}]_t$ is a total function of type $[A, D]_t \leftarrow [B, C]_t$.*

Proof. Straightforward induction on the structure of type expressions using (5) and (20), together with the fact that relators preserve total functions. □

Several lemmas are needed, expressing the properties of the function $[f, g^{\cup}]_t$. In particular, distributivity properties with respect to composition are crucial.

Lemma 26. *For all assignments f, g, h and k of functions to type variables and for all type expressions t*

$$[f, g^{\cup}]_t \bullet [h, k^{\cup}]_t = [f \bullet h, (k \bullet g)^{\cup}]_t.$$

(*Note the contravariance in the second argument.*)

Proof. The induction hypothesis is the equality above together with the symmetric equality

$$[k, h^{\cup}]_t \bullet [g, f^{\cup}]_t = [k \bullet g, (f \bullet h)^{\cup}]_t.$$

The basis follows immediately from (16).

For the case $t = u \leftarrow v$, we have:

$$\begin{array}{ll} & [f, g^{\cup}]_t \bullet [h, k^{\cup}]_t \\ = & \quad \{t = u \leftarrow v, (17) \text{ and } (20)\} \\ & ([f, g^{\cup}]_u \leftarrow [g, f^{\cup}]_v^{\cup}) \bullet ([h, k^{\cup}]_u \leftarrow [k, h^{\cup}]_v^{\cup}) \\ = & \quad \{(10)\text{—which is applicable because} \\ & \quad\ [h, k^{\cup}]_u \text{ and } [k, h^{\cup}]_v \text{ are functions}\} \\ & [f, g^{\cup}]_u \bullet [h, k^{\cup}]_u \leftarrow [g, f^{\cup}]_v^{\cup} \bullet [k, h^{\cup}]_v^{\cup} \\ = & \quad \{\text{converse}\} \\ & [f, g^{\cup}]_u \bullet [h, k^{\cup}]_u \leftarrow ([k, h^{\cup}]_v \bullet [g, f^{\cup}]_v)^{\cup} \\ = & \quad \{\text{induction hypothesis}\} \\ & [f \bullet h, (k \bullet g)^{\cup}]_u \leftarrow [k \bullet g, (f \bullet h)^{\cup}]_v^{\cup} \\ = & \quad \{(20) \text{ and } t = u \leftarrow v\} \\ & [f \bullet h, (k \bullet g)^{\cup}]_t. \end{array}$$

The proof of the second equality is completely symmetrical.

In the case that t is a relator application, the claimed equalities are immediate consequences of the fact that relators distribute through composition, together with (18). □

Lemma 27 (Monotonicity). *Suppose R, S, T and U are assignments such that, for all type variables a,*

$$R_a \subseteq T_a \wedge S_a \supseteq U_a.$$

Then, for all type expressions t,

$$[R, S]_t \subseteq [T, U]_t.$$

Proof. Straightforward induction on the structure of type expressions using (16) for the base case, (17) and (9) for the case that $t = u \leftarrow v$, and the monotonicity of relators and (18) for the remaining case. □

Lemma 28 (Factorisation of functions). *Suppose R, S, f and g are assignments such that, for all type variables a, f_a and g_a are total functions. Then, for all type*

expressions t,

$$[R \bullet f, S \bullet g^{\cup}]_t \subseteq [R,S]_t \bullet [f,g^{\cup}]_t, \quad \textit{and} \tag{29}$$

$$[S,R]_t \bullet [g^{\cup}, f]_t \subseteq [S \bullet g^{\cup}, R \bullet f]_t. \tag{30}$$

Proof. The proof is by induction on the structure of type expressions. First, consider the case that t is a type variable a. By applying property (16), we have

$$[R,S]_a \bullet [f,g^{\cup}]_a = R_a \bullet f_a = [R \bullet f, S \bullet g^{\cup}]_a, \quad \text{and}$$

$$[S,R]_a \bullet [g^{\cup}, f]_a = S_a \bullet g_a^{\cup} = [S \bullet g^{\cup}, R \bullet f]_a.$$

Second, consider the case where $t = u \leftarrow v$. For property (29) we reason:

$$\begin{array}{ll} & [R,S]_{u \leftarrow v} \bullet [f,g^{\cup}]_{u \leftarrow v} \\ = & \quad \{\text{definition: (17)}\} \\ & ([R,S]_u \leftarrow [S,R]_v) \bullet ([f,g^{\cup}]_u \leftarrow [g^{\cup},f]_v) \\ = & \quad \{\text{distributivity: (10), Lemma 25 and (20)}\} \\ & ([R,S]_u \bullet [f,g^{\cup}]_u) \leftarrow ([S,R]_v \bullet [g^{\cup},f]_v) \\ \supseteq & \quad \{\text{monotonicity : (9),} \\ & \quad\ \ \text{induction hypotheses: (29) and (30)}\} \\ & [R \bullet f, S \bullet g^{\cup}]_u \leftarrow [S \bullet g^{\cup}, R \bullet f]_v \\ = & \quad \{\text{definition: (17)}\} \\ & [R \bullet f, S \bullet g^{\cup}]_{u \leftarrow v}. \end{array}$$

For property (30) we reason:

$$\begin{array}{ll} & [S,R]_{u \leftarrow v} \bullet [g^{\cup}, f]_{u \leftarrow v} \\ = & \quad \{\text{definition: (17)}\} \\ & ([S,R]_u \leftarrow [R,S]_v) \bullet ([g^{\cup},f]_u \leftarrow [f,g^{\cup}]_v) \\ \subseteq & \quad \{\text{distributivity: (8)}\} \\ & ([S,R]_u \bullet [g^{\cup},f]_u) \leftarrow ([R,S]_v \bullet [f,g^{\cup}]_v) \\ \subseteq & \quad \{\text{monotonicity: (9), induction hypotheses}\} \\ & [S \bullet g^{\cup}, R \bullet f]_u \leftarrow [R \bullet f, S \bullet g^{\cup}]_v \\ = & \quad \{\text{definition: (17)}\} \\ & [S \bullet g^{\cup}, R \bullet f]_{u \leftarrow v}. \end{array}$$

The case where t is a relator application is straightforward and thus omitted. □

The function that maps function assignments f and g and type expression t to $[f,g^{\cup}]_t$ is parametrically polymorphic. The "free theorem" that this observation yields is the following [7]. (Compare Theorem 31 with Lemma 25 noting how types are replaced by relations. We prove it independently, rather than invoke "theorems-for-free", because it is just as easy to do so.)

Theorem 31 (Naturality of higher-order assignments). *Suppose* f, g, h *and* k *assign total functions to type variables, and* R *and* S *assign relations to type variables.*

Suppose further that, for all type variables a, $(f_a, h_a) \in R_a \leftarrow S_a$ *and* $(g_a, k_a) \in T_a \leftarrow U_a$. *Then, for all type expressions* t,

$$([f, g^{\cup}]_t, [h, k^{\cup}]_t) \in [R, U]_t \leftarrow [S, T]_t.$$

Proof.

$$\begin{array}{ll} & ([f, g^{\cup}]_t, [h, k^{\cup}]_t) \in [R, U]_t \leftarrow [S, T]_t \\ = & \quad \{\text{definition of } \leftarrow (6)\} \\ & [f, g^{\cup}]_t \bullet [S, T]_t \subseteq [R, U]_t \bullet [h, k^{\cup}]_t \\ \Leftarrow & \quad \{(29), (30) \text{ together with the fact that} \\ & \quad\quad \text{converse is a poset isomorphism,} \\ & \quad\quad \text{transitivity of } \subseteq\} \\ & [f \bullet S, g^{\cup} \bullet T]_t \subseteq [R \bullet h, U \bullet k^{\cup}]_t \\ \Leftarrow & \quad \{\text{monotonicity: Lemma 27}\} \\ & \langle \forall a :: (f \bullet S)_a \subseteq (R \bullet h)_a \wedge (g^{\cup} \bullet T)_a \supseteq (U \bullet k^{\cup})_a \rangle \\ = & \quad \{\text{definition: (15)}\} \\ & \langle \forall a :: f_a \bullet S_a \subseteq R_a \bullet h_a \wedge g_a^{\cup} \bullet T_a \supseteq U_a \bullet k_a^{\cup} \rangle \\ = & \quad \{\text{shunting of functions and definition of } \leftarrow : (16)\} \\ & \langle \forall a :: (f_a, h_a) \in R_a \leftarrow S_a \wedge (g_a, k_a) \in T_a \leftarrow U_a \rangle. \quad \square \end{array}$$

Theorem 31 has a number of corollaries. For the purposes of this paper, the most important is the following monotonicity property.

Corollary 32. *Suppose that* f *and* g *assign total functions to type variables, and* R *and* S *assign relations to type variables. Suppose further that, for all type variables* a, $f_a \in R_a \leftarrow S_a$ *and* $g_a \in S_a \leftarrow R_a$. *Then, for all type expressions* t,

$$[f, g^{\cup}]_t \in R_t \leftarrow S_t, \quad \textit{and}$$

$$[g, f^{\cup}]_t \in S_t \leftarrow R_t.$$

Thus, if, for each a, R_a *and* S_a *are ordering relations, and* f_a *and* g_a *are monotonic functions, then* $[f, g^{\cup}]_t$ *and* $[g, f^{\cup}]_t$ *are monotonic functions.*

Proof. Make the instantiation $h, k, T, U := f, g, S, R$ in Theorem 31 and use (19). $\square$

The remainder of this section discusses some other interesting corollaries of Theorem 31. These are not used elsewhere in the paper.

The following is the lemma proved by Backhouse in [4].

Corollary 33. *Suppose* f *and* g *assign total functions to type variables, where* f_a *and* g_a *have type* $A_a \leftarrow B_a$ *and* $C_a \leftarrow D_a$, *respectively. Then, for all type expressions* t,

$$[f, \mathsf{id}_C]_t \bullet [\mathsf{id}_B, g^{\cup}]_t^{\cup} \subseteq [f, g]_t \subseteq [\mathsf{id}_A, g^{\cup}]_t^{\cup} \bullet [f, \mathsf{id}_D]_t.$$

Proof. Using (6), we have $(f_a, \mathsf{id}_{B_a}) \in f_a \leftarrow \mathsf{id}_{B_a}$ and $(\mathsf{id}_{C_a}, g_a) \in \mathsf{id}_{C_a} \leftarrow g_a$. This allows us to instantiate Theorem 31. Specifically,

$$
\begin{array}{ll}
 & \mathit{true} \\
= & \quad \{\text{Theorem 31 with } f,g,h,k := f, \mathsf{id}_C, \mathsf{id}_B, g \\
 & \qquad \text{and } R,S,T,U := f, \mathsf{id}_B, \mathsf{id}_C, g\} \\
 & ([f, \mathsf{id}_C^{\cup}]_t, [\mathsf{id}_B, g^{\cup}]_t) \in [f,g]_t \leftarrow [\mathsf{id}_B, \mathsf{id}_C]_t \\
= & \quad \{\text{definition of arrow (6)}\} \\
 & [f, \mathsf{id}_C^{\cup}]_t \bullet [\mathsf{id}_B, \mathsf{id}_C]_t \subseteq [f,g]_t \bullet [\mathsf{id}_B, g^{\cup}]_t \\
= & \quad \{[\mathsf{id}_B, \mathsf{id}_C]_t \text{ is the identity function}, \\
 & \qquad \text{shunting of functions}, \\
 & \qquad \mathsf{id}_C = \mathsf{id}_C^{\cup}\} \\
 & [f, \mathsf{id}_C]_t \bullet [\mathsf{id}_B, g^{\cup}]_t^{\cup} \subseteq [f,g]_t.
\end{array}
$$

The second inclusion is obtained similarly by observing that $(\mathsf{id}_{A_a}, f_a) \in \mathsf{id}_{A_a} \leftarrow f_a$ and that $(g_a, \mathsf{id}_{D_a}) \in g_a \leftarrow \mathsf{id}_{D_a}$. □

De Bruin [7] shows how Theorem 31 is used to derive a so-called "dinaturality" property from the type of a parametrically polymorphic function.

Corollary 34 (Dinaturality). *Suppose θ is a parametrically polymorphic function of type $t = u \leftarrow v$ for all instances of the type variables in t. Then θ is "dinatural". Specifically, for all assignments f of total functions to the type variables, where f_a has type $A_a \leftarrow B_a$,*

$$[\mathsf{id}_A, f^{\cup}]_u \bullet \theta(A) \bullet [f, \mathsf{id}_A]_v = [f, \mathsf{id}_B]_u \bullet \theta(B) \bullet [\mathsf{id}_B, f^{\cup}]_v.$$

Proof. We combine the free theorem with Corollary 33.

$$
\begin{array}{ll}
 & \mathit{true} \\
= & \quad \{\text{free theorem}\} \\
 & (\theta(A), \theta(B)) \in f_t \\
\Rightarrow & \quad \{f_t = [f,f]_t, \text{ Corollary 33}\} \\
 & (\theta(A), \theta(B)) \in [\mathsf{id}_A, f^{\cup}]_t^{\cup} \bullet [f, \mathsf{id}_B]_t \\
= & \quad \{[\mathsf{id}_A, f^{\cup}]_t \text{ and } [f, \mathsf{id}_B]_t \text{ are functions}\} \\
 & [\mathsf{id}_A, f^{\cup}]_t.\theta(A) = [f, \mathsf{id}_B]_t.\theta(B) \\
= & \quad \{t = u \leftarrow v, \text{ definition (17)}\} \\
 & ([\mathsf{id}_A, f^{\cup}]_u \leftarrow [f^{\cup}, \mathsf{id}_A]_v).\theta(A) = ([f, \mathsf{id}_B]_u \leftarrow [\mathsf{id}_B, f]_v).\theta(B) \\
= & \quad \{\text{converse (20) and definition (5)}\} \\
 & [\mathsf{id}_A, f^{\cup}]_u \bullet \theta(A) \bullet [f, \mathsf{id}_A]_v = [f, \mathsf{id}_B]_u \bullet \theta(B) \bullet [\mathsf{id}_B, f^{\cup}]_v. \qquad \square
\end{array}
$$

A special (and well-known) case of dinaturality is "naturality". A parametrically polymorphic function θ of type $\langle \forall a :: F.a \leftarrow G.a \rangle$, for some relators F and G, is a so-called "natural transformation" to F from G. That is, for all assignments f of functions to the type variables a,

$$\theta(A) \bullet G.f_a = F.f_a \bullet \theta(B).$$

There are many examples: the cons function has type $List.a \leftarrow a \times List.a$, for all a, and so has the property that

$$\mathsf{cons}(A) \bullet f \times List.f = List.f \bullet \mathsf{cons}(B),$$

for all functions f of type $A \leftarrow B$. (Recall that $List.f$ is the function that maps the function f over all elements of a list. In functional programming languages, the type parameters A and B, are not normally made explicit.) Also, the flatten function on lists has type $List.a \leftarrow List.(List.a)$, for all a, and so has the property that

$$\mathsf{flatten}(A) \bullet List.(List.f) = List.f \bullet \mathsf{flatten}(B),$$

for all functions f of type $A \leftarrow B$. An elementary example involving a non-unary relator is the function swap that swaps the values in a pair. This has type $a \times b \leftarrow b \times a$, for all a and b, and so has the property that

$$\mathsf{swap}(A,B) \bullet f \times g = g \times f \bullet \mathsf{swap}(C,D),$$

for all functions f and g, of types $A \leftarrow C$ and $B \leftarrow D$, respectively.

A more illustrative example of the dinaturality property is provided by function composition. Composition has type [8]

$$(a \leftarrow b) \leftarrow (a \leftarrow c) \times (c \leftarrow b)$$

for all instances of the type variables a, b and c. Suppose a function denoted by the infix operator "$\circ$" has the same type as composition. Then, if we expand the dinaturality property of this function, we get that, for all total functions f_a, f_b and f_c (specifying the assignment f to the type variables in the theorem), and all functions g and h,

$$((f_a \bullet g) \circ (f_c \bullet h)) \bullet f_b = f_a \bullet ((g \bullet f_c) \circ (h \bullet f_b)).$$

(See the appendix for the full derivation. The different instances of "$\circ$" have not been distinguished for the purposes of readability.) In particular, by instantiating g and h with identity functions, we get that, for all total functions f_a, f_b and f_c,

$$(f_a \circ f_c) \bullet f_b = f_a \bullet (f_c \circ f_b).$$

In words, any function that has the same type as function composition must "associate" with function composition. Moreover, by now taking f_b to be the identity function (denoted here by id) we get

$$f_a \circ f_c = f_a \bullet f_c \Leftarrow f_c \circ \mathsf{id} = f_c.$$

[8] Astute readers will note that the type of composition given here is different to that given earlier, the difference being whether composition is viewed as being "curried" or "uncurried". Earlier, we used the "curried" type for the technical reason that we had not yet introduced the product relator. Here we use the "uncurried" type because it is easier to apply the dinaturality property.

We conclude that function composition is uniquely defined by the fact that it is parametrically polymorphic and has the identity function as unit.

4.3. Main theorems

In this section, we establish the central theorems on extending pair algebras and Galois connections to higher-order types. First, some simple lemmas are needed.

Lemma 35. *For all partial orderings* $\sqsubseteq$ *and* $\preccurlyeq$ *and all functions* g,

$$\sqsubseteq\leftarrow\preccurlyeq \bullet g \quad = \quad \sqsubseteq\leftarrow g \quad \textit{and}$$

$$\sqsubseteq\leftarrow g^{\cup}\bullet \preccurlyeq \quad = \quad \sqsubseteq\leftarrow g^{\cup}.$$

Proof. We have, for all monotonic functions h and k,

$$\begin{array}{ll} & (h,k) \in \sqsubseteq \leftarrow\preccurlyeq \bullet g \\ = & \quad \{\text{definitions: } (3),(1)\} \\ & \langle \forall u,v :: h.u \sqsubseteq k.v \Leftarrow u \preccurlyeq g.v\rangle \\ = & \quad \{(\Rightarrow) \text{ reflexivity of } \preccurlyeq \\ & \quad (\Leftarrow)\ h \text{ is monotonic, transitivity of } \sqsubseteq\} \\ & \langle \forall v :: h.(g.v) \sqsubseteq k.v\rangle \\ = & \quad \{\text{definitions: } (3),(1)\} \\ & (h,k) \in \sqsubseteq\leftarrow g. \end{array}$$

The second claim is proved similarly. (This is where monotonicity of k is used.) □

Lemma 35 is often used in combination with the distributivity properties (10) and (11). Its most immediate application is property (4).

The next lemma can be seen as a special case of Abramsky's "uniformisation theorem" [1, Proposition 6.4]. (Abramsky does not assume that the function g has an upper adjoint.)

Lemma 36 (Uniformisation). *If* f *is a monotonic function with range* $(A,\sqsubseteq)$, *and the pair* $(g,g^{\sharp})$ *is a Galois connection between the posets* $(B,\preccurlyeq)$ *and* $(C,\leqslant)$,

$$f^{\cup}\bullet \sqsubseteq\leftarrow g^{\cup}\bullet \preccurlyeq \quad = \quad (f^{\cup} \leftarrow g^{\sharp}) \bullet (\sqsubseteq\leftarrow\preccurlyeq).$$

Also,

$$\sqsubseteq \bullet f \leftarrow\preccurlyeq \bullet g^{\sharp} \quad = \quad (\sqsubseteq\leftarrow\leqslant) \bullet (f \leftarrow g^{\cup}).$$

Proof.

$$
\begin{array}{ll}
 & f^{\cup}\bullet \sqsubseteq \leftarrow g^{\cup}\bullet \preccurlyeq \\
= & \quad \{(g, g^{\sharp}) \text{ is a Galois connection.} \\
 & \quad\quad \text{So, } g^{\cup}\bullet \preccurlyeq \;=\; \leqslant \bullet g^{\sharp}.\} \\
 & f^{\cup}\bullet \sqsubseteq \leftarrow \leqslant \bullet g^{\sharp} \\
= & \quad \{\text{distributivity: (11), with } g := \mathsf{id}_C\} \\
 & (f^{\cup} \leftarrow \mathsf{id}_C)\bullet(\sqsubseteq\leftarrow \leqslant \bullet g^{\sharp}) \\
= & \quad \{\text{Lemma 35, with } g := g^{\sharp}\} \\
 & (f^{\cup} \leftarrow \mathsf{id}_C)\bullet(\sqsubseteq\leftarrow g^{\sharp}) \\
= & \quad \{\text{distributivity: (11), with } g := \mathsf{id}_C\} \\
 & f^{\cup}\bullet \sqsubseteq \leftarrow g^{\sharp} \\
= & \quad \{\text{distributivity: (11), with } g := g^{\sharp}\} \\
 & (f^{\cup} \leftarrow g^{\sharp})\bullet(\sqsubseteq\leftarrow \mathsf{id}_B) \\
= & \quad \{\text{Lemma 35, with } g := \mathsf{id}_B\} \\
 & (f^{\cup} \leftarrow g^{\sharp})\bullet(\sqsubseteq\leftarrow\preccurlyeq).
\end{array}
$$

The second property is proved similarly. □

We now come to one of the most important lemmas of the paper. The lemma details how to lift Galois connections to arbitrary types.

Lemma 37 (Higher-order Galois connections). *Suppose that, for each type variable a, we are given a Galois connection (f_a, g_a), between the posets $(A_a, \sqsubseteq_a)$ and $(B_a, \preccurlyeq_a)$, and a Galois connection (h_a, k_a), between the posets $(C_a, \trianglelefteq_a)$ and $(D_a, \leqslant_a)$. Then, for all type expressions t,*

$$[f, k^{\cup}]_t^{\cup} \bullet [\sqsubseteq, \trianglelefteq]_t = [\preccurlyeq, \leqslant]_t \bullet [g, h^{\cup}]_t,$$

and

$$[h, g^{\cup}]_t^{\cup} \bullet [\trianglelefteq, \sqsubseteq]_t = [\leqslant, \preccurlyeq]_t \bullet [k, f^{\cup}]_t.$$

In particular, the pair of functions $([f, k^{\cup}]_t, [g, h^{\cup}]_t)$ forms a Galois connection between the posets $([A, C]_t, [\sqsubseteq, \trianglelefteq]_t)$ and $([B, D]_t, [\preccurlyeq, \leqslant]_t)$.

Proof. Lemma 25 establishes that $[f, k^{\cup}]_t$ and $[g, h^{\cup}]_t$ are functions (of the right type) and Lemma 24 that $[\sqsubseteq, \trianglelefteq]_t$ and $[\preccurlyeq, \leqslant]_t$ are partial ordering relations. So, it suffices only to prove the equality. This we prove by induction on the structure of type expressions.

We are given the equalities: for all type variables a,

$$f_a^{\cup}\bullet \sqsubseteq_a \quad = \quad \preccurlyeq_a \bullet g_a,$$

and

$$h_a^{\cup} \bullet \trianglelefteq_a \quad = \quad \leqslant_a \bullet k_a.$$

Combined with (16), these establish the basis of the proof.

For the case $t = u \leftarrow v$ we have:

$$
\begin{array}{ll}
 & [f, k^{\cup}]_t^{\cup} \bullet [\sqsubseteq, \trianglelefteq]_t \\
= & \quad \{t = u \leftarrow v, \text{ definition (17)}\} \\
 & ([f, k^{\cup}]_u \leftarrow [k^{\cup}, f]_v)^{\cup} \bullet ([\sqsubseteq, \trianglelefteq]_u \leftarrow [\trianglelefteq, \sqsubseteq]_v) \\
= & \quad \{\text{converse: (7) and (20)}\} \\
 & ([f, k^{\cup}]_u^{\cup} \leftarrow [k, f^{\cup}]_v) \bullet ([\sqsubseteq, \trianglelefteq]_u \leftarrow [\trianglelefteq, \sqsubseteq]_v) \\
= & \quad \{\text{uniformization (Lemma 36)} \\
 & \qquad \text{and induction hypothesis} \\
 & \qquad ([k, f^{\cup}]_v \text{ is the upper adjoint of } [h, g^{\cup}]_v)\} \\
 & [f, k^{\cup}]_u^{\cup} \bullet [\sqsubseteq, \trianglelefteq]_u \leftarrow [h, g^{\cup}]_v^{\cup} \bullet [\trianglelefteq, \sqsubseteq]_v \\
= & \quad \{\text{induction hypothesis}\} \\
 & [\preccurlyeq, \leqslant]_u \bullet [g, h^{\cup}]_u \leftarrow [\leqslant, \preccurlyeq]_v \bullet [k, f^{\cup}]_v \\
= & \quad \{\text{uniformization (Lemma 36) and induction hypothesis} \\
 & \qquad ([k, f^{\cup}]_v \text{ is the upper adjoint of } [h, g^{\cup}]_v)\} \\
 & ([\preccurlyeq, \leqslant]_u \leftarrow [\leqslant, \preccurlyeq]_v) \bullet ([g, h^{\cup}]_u \leftarrow [h, g^{\cup}]_v^{\cup}) \\
= & \quad \{\text{converse (20), } t = u \leftarrow v, \text{ definition (17)}\} \\
 & [\preccurlyeq, \leqslant]_t \bullet [g, h^{\cup}]_t.
\end{array}
$$

The proof of the second equality is completely symmetric. Finally, the case $t = F.(u \ldots v)$ is a straightforward application of the fact that relators distribute through composition and commute with converse. □

The pair $(g, g^{\sharp})$ is a *perfect* Galois connection between the posets $(B, \preccurlyeq)$ and $(C, \leqslant)$ if $g \bullet g^{\sharp}$ is the identity relation on B.

Lemma 38. *If the base Galois connections defined in Lemma* 37 *are all perfect, so are the constructed Galois connections.*

Proof. That the base connections are perfect means that $f_a \bullet g_a = \mathsf{id}_{A_a}$ and that $h_a \bullet k_a = \mathsf{id}_{C_a}$. We have to prove that, for all type expressions t,

$$[f, k^{\cup}]_t \bullet [g, h^{\cup}]_t = [\mathsf{id}_A, \mathsf{id}_C]_t.$$

The proof is an application of Lemma 26:

$$
\begin{array}{ll}
 & [f, k^{\cup}]_t \bullet [g, h^{\cup}]_t = [\mathsf{id}_A, \mathsf{id}_C]_t \\
= & \quad \{\text{Lemma 26}\} \\
 & [f \bullet g, (h \bullet k)^{\cup}]_t = [\mathsf{id}_A, \mathsf{id}_C]_t \\
\Leftarrow & \quad \{\text{Lemma 27 (and equality and} \\
 & \qquad \text{inclusion are the same for functions)}\} \\
 & \langle \forall a :: (f \bullet g)_a = \mathsf{id}_{A_a} \wedge (h \bullet k)_a^{\cup} = \mathsf{id}_{C_a} \rangle \\
= & \quad \{(15), \text{ assumption, } (\mathsf{id}_{C_a})^{\cup} = \mathsf{id}_{C_a}\} \\
 & \mathit{true}. \qquad \square
\end{array}
$$

From now to the end of the paper, we suppose we are given, for each type variable a, a Galois connection (abs_a, con_a), between the posets $(A_a, \sqsubseteq_a)$ and $(B_a, \preccurlyeq_a)$. For

each variable a, we define P_a to be the pair algebra corresponding to the given Galois connection. That is,

$$abs_a^{\cup} \bullet \sqsubseteq_a \quad = \quad P_a \quad = \quad \preccurlyeq_a \bullet con_a. \tag{39}$$

Theorem 40 (Logical pair algebras). *For all type expressions t,*

$$[abs, con^{\cup}]_t^{\cup} \bullet \sqsubseteq_t \quad = \quad P_t \quad = \quad \preccurlyeq_t \bullet [con, abs^{\cup}]_t.$$

In words, the logical relation P defines a Galois connection for all type expressions t, namely the pair of functions $([abs, con^{\cup}]_t, [con, abs^{\cup}]_t)$ connecting the posets $(A_t, \sqsubseteq_t)$ and $(B_t, \preccurlyeq_t)$. Moreover, P_t is a pair algebra for all type expressions t.

Proof. Lemma 37 establishes the equality between the outer terms. (Take $f = h = abs$ and $g = k = con$. Also, take $\trianglelefteq$ to be equal to $\sqsubseteq$, and $\leqslant$ to be equal to $\preccurlyeq$.) We, therefore, only have to establish the equality between the first two terms.

The proof is by induction on the structure of type expressions. The basis is assumption (39) combined with (16). For $t = u \leftarrow v$ we have:

$$\begin{array}{ll} & [abs, con^{\cup}]_t^{\cup} \bullet \sqsubseteq_t \\ = & \quad \{t = u \leftarrow v,\ (22) \text{ and } (17)\} \\ & ([abs, con^{\cup}]_u \leftarrow [con^{\cup}, abs]_v)^{\cup} \bullet (\sqsubseteq_u \leftarrow \sqsubseteq_v) \\ = & \quad \{\text{converse: } (7) \text{ and } (20)\} \\ & ([abs, con^{\cup}]_u^{\cup} \leftarrow [con, abs^{\cup}]_v) \bullet (\sqsubseteq_u \leftarrow \sqsubseteq_v) \\ = & \quad \{\text{uniformisation (Lemma 36), induction hypothesis}\} \\ & [abs, con^{\cup}]_u^{\cup} \bullet \sqsubseteq_u \leftarrow [abs, con^{\cup}]_v^{\cup} \bullet \sqsubseteq_v \\ = & \quad \{\text{induction hypothesis, } t = u \leftarrow v, \text{ definition } (22)\} \\ & P_t. \end{array}$$

Finally, the case $t = F.(u \ldots v)$ is a straightforward application of the distributivity of relators through composition and their preservation of converse. □

Returning to our running example, an instance of theorem 40 is that the pair of functions

$$dk \leftarrow List.kd^{\cup} \leftarrow (kd^{\cup} \leftarrow dk \times dk) \leftarrow kd^{\cup}$$

and

$$kd \leftarrow List.dk^{\cup} \leftarrow (dk^{\cup} \leftarrow kd \times kd) \leftarrow dk^{\cup}$$

are the lower and upper adjoints in a Galois connection. The posets were given earlier—see Lemma 24.

Theorem 40 expresses formally that the extension of a family of Galois connections from a collection of base types to arbitrary higher-order types is precisely defined by a logical relation (the pair algebra P in the theorem). An important corollary, already observed by Abramsky [1], is that this leads to "safety-for-free" in applications to abstract interpretation.

Theorem 41 (Safety for free). *If θ is a parametrically polymorphic function of type t,*

$$[abs, con^{\cup}]_t.\theta(B) \sqsubseteq_t \theta(A),$$

and

$$\theta(B) \preccurlyeq_t [con, abs^{\cup}]_t.\theta(A).$$

($\theta(X)$ denotes the instance of θ of type X_t.)

Proof. The assumption that θ has type t and is parametrically polymorphic in all the type variables in t means that

$$(\theta(B), \theta(A)) \in P_t,$$

where P is the logical relation defined by (39). The theorem follows immediately from Theorem 40, which states that P_t is a pair algebra with lower and upper adjoints $[abs, con^{\cup}]_t$ and $[con, abs^{\cup}]_t$, respectively. □

5. Pointwise orderings

In this section we specialise the theorems of the previous section in order to demonstrate more clearly their relevance to practical application. For example, we show that the logical relation $\sqsubseteq_t$ in the statement of Theorem 41 is just a pointwise ordering relation. We first introduce a novel way of assigning to type variables.

Suppose V and W are two assignments. Then, the assignment $[\![V, W]\!]$ is defined inductively as follows:

$$[\![V, W]\!]_a = V_a, \tag{42}$$

$$[\![V, W]\!]_{u \leftarrow v} = [\![V, W]\!]_u \leftarrow W_v, \tag{43}$$

$$[\![V, W]\!]_{F.(u \ldots v)} = F.([\![V, W]\!]_u \ldots [\![V, W]\!]_v). \tag{44}$$

The assignment $[\![V, W]\!]$ applies the assignment V to the highest positive occurrences of the variables, and applies assignment W to all others. For example,

$$\begin{array}{ll} & [\![(a, b := \sqsubseteq, \trianglelefteq), (a, b := I, J)]\!]_{List.((a \leftarrow b) \times b) \leftarrow a \leftarrow (b \leftarrow a)} \\ = & List.((\sqsubseteq \leftarrow J) \times \trianglelefteq) \leftarrow I \leftarrow (J \leftarrow I), \end{array}$$

whereas

$$\begin{array}{ll} & [(a, b := \sqsubseteq, \trianglelefteq), (a, b := I, J)]_{List.((a \leftarrow b) \times b) \leftarrow a \leftarrow (b \leftarrow a)} \\ = & List.((\sqsubseteq \leftarrow J) \times \trianglelefteq) \leftarrow I \leftarrow (J \leftarrow \sqsubseteq). \end{array}$$

(Note the difference in the treatment of the rightmost occurrence of the type variable a.) If I and J are instantiated to the identity relations on posets A and B, respectively, the relation

$$List.((\sqsubseteq \leftarrow \mathsf{id}_B) \times \trianglelefteq) \leftarrow \mathsf{id}_A \leftarrow (\mathsf{id}_B \leftarrow \mathsf{id}_A)$$

is the pointwise ordering relation on functions of type

$$List.((A \leftarrow B) \times B) \leftarrow A \leftarrow (B \leftarrow A).$$

As the next lemma states, it is equal to the relation

$$List.((\sqsubseteq \leftarrow \trianglelefteq) \times \trianglelefteq) \leftarrow \sqsubseteq \leftarrow (\trianglelefteq \leftarrow \sqsubseteq).$$

Lemma 45 (Higher-order pointwise orderings). *If, for each variable a, $\sqsubseteq_a$ is a partial ordering relation on the set A_a and id_a is the identity relation on A_a then, for all type expressions t,*

$$[\![\sqsubseteq, \mathsf{id}]\!]_t = \sqsubseteq_t .$$

Proof. By induction on type expressions. The basis is a trivial use of (42). For $t = u \leftarrow v$ we apply property (4):

$$\begin{array}{ll} & \sqsubseteq_{u \leftarrow v} \\ = & \quad \{\text{definition}\} \\ & \sqsubseteq_u \leftarrow \sqsubseteq_v \\ = & \quad \{\text{Lemma 35, with } g := \mathsf{id}\} \\ & \sqsubseteq_u \leftarrow \mathsf{id}_v \\ = & \quad \{\text{induction hypothesis}\} \\ & [\![\sqsubseteq, \mathsf{id}]\!]_u \leftarrow \mathsf{id}_v \\ = & \quad \{t = u \leftarrow v, \text{ definition: (43)}\} \\ & [\![\sqsubseteq, \mathsf{id}]\!]_t. \end{array}$$

The case $t = F.(u \ldots v)$ is straightforward:

$$\begin{array}{ll} & \sqsubseteq_{F.(u \ldots v)} \\ = & \quad \{\text{definition}\} \\ & F.(\sqsubseteq_u \ldots \sqsubseteq_v) \\ = & \quad \{\text{induction hypothesis}\} \\ & F.([\![\sqsubseteq, \mathsf{id}]\!]_u \ldots [\![\sqsubseteq, \mathsf{id}]\!]_v) \\ = & \quad \{t = F.(u \ldots v), \text{ definition: (44)}\} \\ & [\![\sqsubseteq, \mathsf{id}]\!]_t. \qquad \square \end{array}$$

In words, Lemma 45 states that the pointwise ordering of functions, $[\![\sqsubseteq, \mathsf{id}]\!]_t$, is the logical relation $\sqsubseteq_t$ generated by the given base orderings. For example, the ordering relation

$$(\Leftarrow) \leftarrow List.(\Leftarrow) \leftarrow ((\Leftarrow) \leftarrow (\Leftarrow) \times (\Leftarrow)) \leftarrow (\Leftarrow)$$

on

$$\mathsf{Bool} \leftarrow List.\mathsf{Bool} \leftarrow (\mathsf{Bool} \leftarrow \mathsf{Bool} \times \mathsf{Bool}) \leftarrow \mathsf{Bool}$$

is the "pointwise" ordering,

$$(\Leftarrow) \leftarrow List.\mathsf{id}_{\mathsf{Bool}} \leftarrow (\mathsf{id}_{\mathsf{Bool}} \leftarrow \mathsf{id}_{\mathsf{Bool}} \times \mathsf{id}_{\mathsf{Bool}}) \leftarrow \mathsf{id}_{\mathsf{Bool}}.$$

Spelt out, this is just the relation $\dot{\sqsubseteq}$ on boolean-valued functions f and g defined by

$$f \dot{\sqsubseteq} g \quad \equiv \quad \langle \forall b, bop, bs :: f.b.bop.bs \Leftarrow g.b.bop.bs \rangle.$$

Similarly, the ordering relation

$$(/) \leftarrow List.(/) \leftarrow ((/) \leftarrow (/) \times (/)) \leftarrow (/)$$

on

$$\mathsf{PosInt} \leftarrow List.\mathsf{PosInt} \leftarrow (\mathsf{PosInt} \leftarrow \mathsf{PosInt} \times \mathsf{PosInt}) \leftarrow \mathsf{PosInt}$$

is the pointwise ordering of positive-integer-valued functions f and g,

$$f \dot{\sqsubseteq} g \quad \equiv \quad \langle \forall m, pop, ms :: f.m.pop.ms \,/\, g.m.pop.ms \rangle.$$

Lemma 46.

$$P_t = [\![abs^{\cup} \bullet \sqsubseteq, [abs, con^{\cup}]^{\cup}]\!]_t.$$

Proof. The proof is by induction on the structure of type expressions. The basis is trivial. For $t = u \leftarrow v$ we have:

$$\begin{array}{ll} & P_t \\ = & \quad \{t = u \leftarrow v, \text{ definition (22), Theorem 40}\} \\ & P_u \leftarrow [abs, con^{\cup}]_v^{\cup} \bullet \sqsubseteq_v \\ = & \quad \{\text{distributivity: (11), used twice with } g := \mathsf{id}, \\ & \qquad \text{and Lemma 35} \\ & \qquad \text{(noting that } P_u = [abs, con^{\cup}]_u^{\cup} \bullet \sqsubseteq_u)\} \\ & P_u \leftarrow [abs, con^{\cup}]_v^{\cup} \\ = & \quad \{\text{induction hypothesis}\} \\ & [\![abs^{\cup} \bullet \sqsubseteq, [abs, con^{\cup}]^{\cup}]\!]_u \leftarrow [abs, con^{\cup}]_v^{\cup} \\ = & \quad \{\text{definition: (43), } t = u \leftarrow v\} \\ & [\![abs^{\cup} \bullet \sqsubseteq, [abs, con^{\cup}]^{\cup}]\!]_t. \end{array}$$

The case $t = F.(u \ldots v)$ is a straightforward consequence of (44). □

Corollary 47 (Safety for free, special case). *If θ has type $t = a \leftarrow v$, where a is a type variable and v is a type expression, and is parametrically polymorphic in all the type variables in t then*

$$abs_a \bullet \theta(B) \quad \dot{\sqsubseteq} \quad \theta(A) \bullet [abs, con^{\cup}]_v$$

where $\dot{\sqsubseteq}$ denotes the pointwise ordering $[\![\sqsubseteq, \mathsf{id}]\!]$ on functions with range A_a.

Proof. The assumption that θ has type t and is parametrically polymorphic in all the type variables in $a \leftarrow v$ means that

$$(\theta(B), \theta(A)) \in P_{a \leftarrow v}.$$

Now,

$$
\begin{array}{ll}
 & P_{a \leftarrow v} \\
= & \quad \{\text{Lemma 46}\} \\
 & [\![abs^{\cup} \bullet \sqsubseteq, [abs, con^{\cup}]^{\cup}]\!]_{a \leftarrow v} \\
= & \quad \{\text{definitions: (43) and (15)}\} \\
 & abs_a^{\cup} \bullet \sqsubseteq_a \leftarrow [abs, con^{\cup}]_v^{\cup} \\
= & \quad \{\text{distributivity: (10) and (11)}\} \\
 & (abs_a^{\cup} \leftarrow \mathsf{id}_v) \bullet (\sqsubseteq_a \leftarrow \mathsf{id}_v) \bullet (\mathsf{id}_a \leftarrow [abs, con^{\cup}]_v^{\cup}) \\
= & \quad \{\dot{\sqsubseteq} = [\![\sqsubseteq, \mathsf{id}]\!]_{a \leftarrow v} = \sqsubseteq_a \leftarrow \mathsf{id}_v, \\
 & \qquad \text{converse: (7)}\} \\
 & (abs_a \leftarrow \mathsf{id}_v)^{\cup} \bullet \dot{\sqsubseteq} \bullet (\mathsf{id}_a \leftarrow [abs, con^{\cup}]_v^{\cup}).
\end{array}
$$

Hence, using (5) and the definitions of composition and converse,

$$abs_a \bullet \theta(B) \quad \dot{\sqsubseteq} \quad \theta(A) \bullet [abs, con^{\cup}]_v. \qquad \square$$

Let us now apply our theorems to the running example. We have already observed that

$$dk \leftarrow List.kd^{\cup} \leftarrow (kd^{\cup} \leftarrow dk \times dk) \leftarrow kd^{\cup}$$

is the lower adjoint in a Galois connection between the poset of monotonic functions of type

$$\mathsf{Bool} \leftarrow List.\mathsf{Bool} \leftarrow (\mathsf{Bool} \leftarrow \mathsf{Bool} \times \mathsf{Bool}) \leftarrow \mathsf{Bool}$$

ordered pointwise by $\Leftarrow$, and monotonic functions of type

$$\mathsf{PosInt} \leftarrow List.\mathsf{PosInt} \leftarrow (\mathsf{PosInt} \leftarrow \mathsf{PosInt} \times \mathsf{PosInt}) \leftarrow \mathsf{PosInt}$$

ordered pointwise by divisibility. We have also observed that the upper adjoint is the function

$$kd \leftarrow List.dk^{\cup} \leftarrow (dk^{\cup} \leftarrow kd \times kd) \leftarrow dk^{\cup}.$$

Theorem 41 says that the Galois connection defines a safe abstract interpretation of the evaluation of the *fold* function. This is made explicit by "uncurrying" the function and applying Corollary 47. To be precise, assuming *fold* has type

$$a \leftarrow a \times (a \leftarrow a \times a) \times List.a,$$

the corollary predicts that, for all positive numbers m and list of positive numbers ms,

$$dk.ConcreteValue \Leftarrow AbstractValue$$

where

$$ConcreteValue = fold.(m, (*), ms)$$

and

$$AbstractValue = fold.(dk.m, (dk \leftarrow kd^{\cup} \times kd^{\cup}).(*), List.dk.ms).$$

We have also calculated that, for booleans b and c,

$$(dk \leftarrow kd^{\cup} \times kd^{\cup}).(*).(b, c) = dk.(kd.b * kd.c) = ((k = 1) \vee b \vee c).$$

So, the theorem predicts the (obvious) property that it is safe to evaluate whether the product of a list of numbers is divisible by k by interpreting each number as the boolean value "the number is divisible by k" and interpreting multiplication as logical disjunction. "Safe" means that a *true* answer can be relied upon.

We conclude our theorems with a property crucial to the abstract interpretation of functions that exploit general recursion.

When applying "theorems-for-free", care must be taken when the polymorphic functions in question are defined as fixed points (i.e. when general recursion is used). The proof of "theorems-for-free" is by induction on the construction of polymorphic functions; the "free theorem" is, thus, a "healthiness" property of the mechanisms for constructing functions. But, fixed-point computation is not completely "healthy" in this sense [24]. Fortunately, it is sufficiently healthy for our purposes. Specifically, the inductive step in the proof of "theorems-for-free" is valid in the case of fixed-point computations if the relation assignments in the statement of the theorem are not arbitrary, but restricted to pair algebras. This is stated formally in the next theorem. The theorem is needed, for example, to justify the safety of strictness analysis—see Section 7—since the construction of an interpreter for a language inevitably involves a fixed-point computation.

Theorem 48 (Fixed point fusion). *Suppose that, for each type variable a, $(A_a, \sqsubseteq_a)$ is a poset. Let μ denote the polymorphic function that maps (monotonic) function f of type $(A_t, \sqsubseteq_t) \leftarrow (A_t, \sqsubseteq_t)$ to its least fixed point. Then μ is parametric in all pair algebras. That is, given an assignment of Galois connections (abs_a, con_a) between the posets $(A_a, \sqsubseteq_a)$ and $(B_a, \preccurlyeq_a)$ for each type variable a, and letting P be the corresponding logical pair algebra, we have, for all type expressions t,*

$$(\mu, \mu) \in P_{t \leftarrow (t \leftarrow t)}.$$

Also, let ν denote the polymorphic function that maps (monotonic) function f of type $(A_t, \sqsubseteq_t) \leftarrow (A_t, \sqsubseteq_t)$ to its greatest fixed point. Then ν is parametric in all pair algebras. That is, for all type expressions t,

$$(\nu, \nu) \in P_{t \leftarrow (t \leftarrow t)}.$$

Proof.

$$
\begin{array}{ll}
 & (\mu,\mu) \in P_{t\leftarrow(t\leftarrow t)} \\
= & \quad \{\text{definition: (3)}\} \\
 & \langle \forall f,g :: (\mu f,\mu g) \in P_t \Leftarrow (f,g) \in P_{t\leftarrow t} \rangle \\
= & \quad \{\text{Theorem 40 and Lemma 46}\} \\
 & \langle \forall f,g :: \\
 & \qquad [abs, con^{\cup}]_t.\mu f \sqsubseteq_t \mu g \\
 & \qquad \Leftarrow (f,g) \in [\![abs^{\cup} \bullet \sqsubseteq, [abs, con^{\cup}]^{\cup}]\!]_{t\leftarrow t} \\
 & \rangle \\
= & \quad \{\text{fixed-point fusion (see below for explanation)}\} \\
 & true.
\end{array}
$$

To see that the last step of the above proof is an instance of fixed-point fusion, i.e. that

$$h.\mu f \sqsubseteq \mu g \Leftarrow h \bullet f \mathrel{\dot{\sqsubseteq}} g \bullet h,$$

whenever h is a lower adjoint in a Galois connection, we need to rephrase the premise in the above implication:

$$
\begin{array}{ll}
 & (f,g) \in [\![abs^{\cup} \bullet \sqsubseteq, [abs, con^{\cup}]^{\cup}]\!]_{t\leftarrow t} \\
= & \quad \{\text{definition (43), Lemma 46 and Theorem 40}\} \\
 & (f,g) \in [abs, con^{\cup}]_t^{\cup} \bullet \sqsubseteq_t \leftarrow [abs, con^{\cup}]_t^{\cup} \\
= & \quad \{\text{distributivity: (10) and (11)}\} \\
 & (f,g) \in ([abs, con^{\cup}]_t^{\cup} \leftarrow \mathsf{id}_{A_t}) \bullet (\sqsubseteq_t \leftarrow \mathsf{id}_{A_t}) \bullet (\mathsf{id}_{B_t} \leftarrow [abs, con^{\cup}]_t^{\cup}) \\
= & \quad \{(5) \text{ and relation calculus} \\
 & \qquad \text{(specifically, } (x,y) \in h^{\cup} \bullet R \bullet k \quad \equiv \quad h.x\, R\, k.y \\
 & \qquad \text{for all } x,\ y, \text{ functions } h \text{ and } k, \text{ and relations } R)\} \\
 & ([abs, con^{\cup}]_t \bullet f) \sqsubseteq_t \leftarrow \mathsf{id}_{A_t}\ (g \bullet [abs, con^{\cup}]_t) \\
= & \quad \{\text{definition (43) and Lemma 45}\} \\
 & ([abs, con^{\cup}]_t^{\cup} \bullet f)\ [\![\sqsubseteq, \mathsf{id}_A]\!]_{t\leftarrow t}\ (g \bullet [abs, con^{\cup}]_t).
\end{array}
$$

The second claim is dual to the first. □

The next two sections consider more substantial examples. For another substantial example, involving a definedness test on attribute grammars, see [2].

6. CSP traces

This section illustrates how our theorems might be applied to programming language semantics. The example is based on a version of Hoare's CSP language [15], restricted to just choice and concurrency. Here are two simple processes, written in Hoare's notation:

$$
\begin{array}{l}
Customer = coin \rightarrow choc \rightarrow Customer, \\
Vendor = coin \rightarrow (choc \rightarrow Vendor \mid toffee \rightarrow Vendor).
\end{array}
$$

Identifiers beginning with a capital letter denote *processes*; the other identifiers denote *events*. If P is a process and x an event, then $x \to P$ is a process that first engages in the event x and then behaves like P. The process $x \to P \mid y \to Q$ offers a choice. Either it engages in x and then continues as P, or it engages in y and then continues as Q. Hoare generalises these constructions to the general *choice* operator $(x : B \to P(x))$. Here B is a finite set of events. The process first engages in one of the events in B and then continues as $P(x)$.

A second method of constructing processes is *concurrency*. The process $P \| Q$ is the synchronisation of the processes P and Q. That is, an event can happen only if both P and Q can simultaneously engage in that event. For example, we might want to run the *Customer* and *Vendor* processes concurrently. This would lead to a process that is equivalent to the *Customer* process, because the *Vendor* process is prepared to vend toffee as well as chocolate, but the customer only ever requests chocolate.

6.1. The interface to the library

We first define an interface for a simple CSP library. The interface contains two methods, the general choice operator and the concurrency operator. These are parameterised by the type of processes, τ, and the type of events, α, in order that they can be given different interpretations at a later stage.

$$\textbf{type}\ \mathit{CSP}.\tau.\alpha = \begin{bmatrix} \mathit{choice} & : \tau \leftarrow \mathit{List}.(\alpha \times \tau), \\ (\|) & : \tau \leftarrow \tau \times \tau. \end{bmatrix}$$

The definition of $\to$ is a special case of *choice*:

$$\begin{aligned} &(\to) \quad :: \langle \forall \tau, \alpha :: \tau \leftarrow \tau \leftarrow \alpha \leftarrow \mathit{CSP}.\tau.\alpha \rangle, \\ &x \to_C P = \mathit{choice}_C.[(x, P)]. \end{aligned}$$

(The use of a subscript C is overloaded. In $x \to_C P$, it denotes application of $\to$, with first argument C, second argument x, and third argument P. As a subscript of *choice*, it means field selection from a record.)

Let us now define the *Customer* and *Vendor* processes with our library. First, we need to define the concrete type of events:

$$\textbf{type}\ E = \mathit{coin} \mid \mathit{choc} \mid \mathit{toffee}.$$

Then the definitions of the processes are:

$$\begin{aligned} &\mathit{Customer} \quad :: \langle \forall \tau :: \tau \leftarrow \mathit{CSP}.\tau.E \rangle, \\ &\mathit{Customer}_C = \mathit{coin} \to_C \mathit{choc} \to_C \mathit{Customer}_C. \end{aligned}$$

$$\begin{aligned} &\mathit{Vendor} \quad :: \langle \forall \tau :: \tau \leftarrow \mathit{CSP}.\tau.E \rangle, \\ &\mathit{Vendor}_C = \mathit{coin} \to_C (\mathit{choice}_C\ [(\mathit{choc}, \mathit{Vendor}_C), (\mathit{toffee}, \mathit{Vendor}_C)]). \end{aligned}$$

Note that these processes are not polymorphic in the type of events, because they explicitly refer to the events *coin*, *choc* and *toffee*. However, they are still polymorphic in τ, because they are still independent of the implementation of the library.

6.2. Traces

Hoare defines the semantics of CSP using *traces*. A trace t of the process P is a finite sequence of events that P might engage in. We use lists to represent traces. Traces are partially ordered:

$$s \leqslant t \quad \equiv \quad \langle \exists u :: s +\!\!+ u = t \rangle. \tag{49}$$

The semantics of a process P is the set of all traces of P. This leads us to define the type of processes as:

$$\textbf{type } \mathit{Proc}.\alpha \quad = \quad \mathit{Set}.(\mathit{List}.\alpha).$$

That is, if events have type α, processes are sets of traces of α. For a given set E of events, $\mathit{Proc}.E$ is a complete lattice under the set inclusion ($\subseteq$) ordering. The bottom element of the lattice is the empty set $\emptyset$ and the top element is $\mathit{all}.E$, where all is defined as follows:

$$\begin{array}{l} \mathit{all} \quad :: \langle \forall \alpha :: \mathit{Proc}.\alpha \leftarrow \mathit{Set}.\alpha \rangle, \\ \mathit{all}.X \; = \; \{s \mid \mathit{elems}.s \subseteq X\}. \end{array}$$

(In the notation of regular expressions $\mathit{all}.X$ is X^*.) The function elems calculates the set of events that appear in a trace. Its definition is:

$$\begin{array}{l} \mathit{elems} \qquad :: \langle \forall \alpha :: \mathit{Set}.\alpha \leftarrow \mathit{List}.\alpha \rangle, \\ \mathit{elems}.[] \qquad = \emptyset, \\ \mathit{elems}.(x : xs) \; = \; \{x\} \cup \mathit{elems}.xs. \end{array}$$

Using this definition of a process, we can implement the *CSP* interface:

$$Tr :: \langle \forall \alpha :: \mathit{CSP}.(\mathit{Proc}.\alpha).\alpha \rangle,$$

$$Tr = \begin{bmatrix} \mathit{choice}.[] & = \{[]\}, \\ \mathit{choice}.((x,P) : cs) & = \{x : s \mid s \in P\} \cup \mathit{choice}.cs, \\ P \| Q & = P \cap Q. \end{bmatrix}$$

We can use Tr to evaluate some of the processes that we defined earlier. For example, if we evaluate the *Customer* process, we find that:

$$\begin{array}{ll} \mathit{Customer}_{Tr} & = \{[]\} \cup \{\mathit{coin} : s \mid s \in X\}, \\ X & = \{[]\} \cup \{\mathit{choc} : s \mid s \in \mathit{Customer}_{Tr}\}. \end{array}$$

6.3. Analysing the alphabet

The *Customer* and *Vendor* processes are very simple, so it is easy for us to see which events they might engage in. *Customer* can only engage in *coin* and *choc*, whereas *Vendor* is also capable of engaging in *toffee*. However, in a larger system it may be useful to have a tool that can do this analysis automatically.

If we are given the set of traces of a process, then we can use the function *events* to find the set of events that the process might engage in:

$$\begin{aligned} &events &&:: \langle \forall \alpha :: Set.\alpha \leftarrow Proc.\alpha \rangle, \\ &events.P &&= \left\langle \bigcup s : s \in P : elems.s \right\rangle. \end{aligned}$$

Notice that *events* forms a Galois connection with *all*:

$$events.P \subseteq X \quad \equiv \quad P \subseteq all.X.$$

This means that we can derive a new library *Ev* that calculates the set of events that a process engages in. The library is derived with the goal of constructing safe abstract interpretations of *Customer* and *Vendor*, which have types

$$\langle \forall \tau :: \tau \leftarrow CSP.\tau.E \rangle.$$

Therefore, we apply Corollary 47, with t instantiated to the type expression $a \leftarrow CSP.a.E$. The variable assignments A and B are defined to be $(a := Set.E)$ and $(a := Proc.E)$, respectively. The variable assignments *abs* and *con* are defined to be $(a := events)$ and $(a := all)$, respectively.

The type expression t has the special form discussed in Corollary 47, so it is this corollary we apply, rather than Theorem 41 itself.

First, taking θ to be *Customer*, we get that

$$\begin{aligned} &events \bullet Customer(a := Proc.E) \\ \dot{\sqsubseteq} \quad &Customer(a := Set.E) \bullet [abs, con^{\cup}]_{CSP.a.E}. \end{aligned}$$

The ordering $\dot{\sqsubseteq}$ is the pointwise subset ordering on functions with type

$$Set.E \leftarrow CSP.(Proc.E).E.$$

So, we now apply both sides of the ordering to the trace implementation of the *CSP* interface. On the left, we get

$$events.Customer_{Tr},$$

where $Customer_{Tr}$ is as defined above. The right side we call $Customer_{Ev}$, and the pointwise ordering becomes the subset ordering. That is, we have

$$events.Customer_{Tr} \subseteq Customer_{Ev} \tag{50}$$

provided that the implementations of choice and concurrency in the new library *Ev* are given by

$$\begin{aligned} &choice_{Ev} &&= [abs, con^{\cup}]_{a \leftarrow List.(E \times a)}.choice_{Tr}, \\ &(\|_{Ev}) &&= [abs, con^{\cup}]_{a \leftarrow a \times a}.(\|_{Tr}). \end{aligned}$$

Using the properties of the arrow operator (in particular (5)), this reduces to:

$$\begin{aligned} &choice_{Ev} &&= events \bullet choice_{Tr} \bullet List.(id_E \times all), \\ &(\|_{Ev}) &&= events \bullet (\|_{Tr}) \bullet (all \times all). \end{aligned}$$

Hence, the definition of Ev is:

$$Ev :: \langle \forall \alpha :: CSP.(Set.\alpha).\alpha \rangle,$$
$$Ev = \begin{bmatrix} choice.[] & = \emptyset, \\ choice.((x,P) : cs) & = \{x\} \cup P \cup choice.cs. \\ P \| Q & = P \cap Q. \end{bmatrix}$$

Using this library, we evaluate the *Customer* process:

$$Customer_{Ev} = \{coin\} \cup \{choc\} \cup Customer_{Ev}.$$

In other words, $Customer_{Ev} \supseteq \{coin, choc\}$, as expected.

Corollary 47 can also be applied with θ instantiated to *Vendor*. Because *Vendor* and *Customer* have the same types, the library Ev used to analyse *Customer* is also used to analyse *Vendor*. Applying it, we find that $Vendor_{Ev} \supseteq \{coin, choc, toffee\}$, which is also as expected. The safety of these evaluations—formally expressed by (50) in the case of *Customer*—is guaranteed by Corollary 47 and the parametric polymorphism of *Tr*.

7. Strictness analysis

Possibly one of the most successful applications of abstract interpretation has been the strictness analysis of lazy functional programs. This application was discovered by Mycroft [18]. Peyton Jones [21, pp. 380–395] gives an excellent introduction to strictness analysis and a discussion of its practical use in compilers. For a more formal treatment, see Burn et al. [8]. The goal of strictness analysis is to discover functions that do not require lazy evaluation techniques. Such *strict* functions can be implemented more efficiently. A simple example of a lazy function is the Boolean *and* operator: $\wedge$. If the first argument is *false*, there is no need to evaluate the second argument, because $(false \wedge X) = false$, for all X. In a lazy language, $false \wedge X$ should therefore terminate, even if X is a non-terminating expression. Implementations achieve this effect by wrapping expressions in *thunks*, which are evaluated on demand. Thunks are costly, both in terms of efficiency and memory usage, so it is profitable to minimise their use. In particular, a thunk is redundant for an expression that is guaranteed to be evaluated. The first argument of $\wedge$ is always evaluated, so it does not need a thunk. A 3-argument function f is strict in (say) its second argument if $f.x.Bot.z = Bot$, for all x and z. *Bot* is the semantic representation of failure and non-termination. So, this equation states that, if evaluation of the second argument fails, the function is guaranteed to fail. The goal of strictness analysis is to discover such properties, so that an implementation can exploit them.

In this section, safety-for-free is illustrated by developing a strictness analyser for a simple first-order functional language. This is done by following the three standard steps of abstract interpretation:

1. An interpreter for the language is defined.
2. The analysis is specified by defining a Galois connection.
3. The Galois connection is used to derive an *abstract* interpreter for the language.

Step 3 of this process is substantially simplified by exploiting safety-for-free, if the interpreter is defined as a polymorphic function. The abstract interpreter is then just a different instantiation of the same function.

7.1. The first-order language

The language considered in this section is first-order, which means that it does not have lambda abstractions or currying. Instead, function declarations appear at the top level, as follows:

$$\begin{array}{ll} f(x,\ldots,y) & = \ldots \\ & \vdots \\ g(x,\ldots,y) & = \ldots \\ main(x,\ldots,y) & = \ldots \end{array}$$

A program consists of a list of (first-order) function definitions, one of which should define a function called *main*; this is the entry point of the program. The function definitions (including *main*) are allowed to be recursive or mutually recursive.

Below is an example of a program that computes factorials:

$$\begin{array}{ll} fac(n) & = \textbf{if } n \geqslant 1 \quad \textbf{then } n * fac(n-1) \\ & \qquad\qquad\quad \textbf{else } 1 \\ main(n) & = fac(n) \end{array}$$

A second example, which is used later to illustrate strictness analysis, is an incorrect version of the factorial program in which the test for zero has been omitted; this program fails to terminate:

$$\begin{array}{ll} fac(n) & = n * fac(n-1) \\ main(n) & = fac(n) \end{array}$$

Finally, here is a simple example of a non-strict program:

$$\begin{array}{ll} main(n,m) & = \textbf{if } n \geqslant 0 \quad \textbf{then } n * m \\ & \qquad\qquad\quad \textbf{else } 0 \end{array}$$

This program only evaluates its second argument if its first argument is at least zero. Hence, it is strict in its first argument, but not in its second.

7.2. Abstract syntax

A program is a list of function definitions, so its abstract syntax is:

$$\textbf{type } Prog = List.FunDef\,.$$

A function definition consists of a name, a list of parameter names and a function body:

$$\textbf{type } FunDef = String \times List.String \times Exp.$$

An expression is either a function application, a constant application or a variable.

$$\begin{aligned}\textbf{data}\ Exp = {} & Call\ String\ List.Exp \\ \mid {} & Const\ String\ List.Exp \\ \mid {} & Var\ String\end{aligned}$$

This language has been kept intentionally simple by not providing many built-in language features; instead they are imported as constants. For example, *if−then−else* is a constant function that takes three parameters: the condition and two expressions. Simple constants, such as the integer constant 10, are constant functions that take zero arguments.

7.3. The interpreter

A simple interpreter for the language is a function of two parameters: the abstract syntax of the program, and the list of arguments to *main*:

$$evalProg \quad :: \quad Value \leftarrow List.Value \leftarrow Prog.$$

To maximise the benefit obtained from theorems-for-free, the interpreter is designed to be parametrically polymorphic instead. This is achieved by abstracting from the *Value* type, and adding an extra parameter:

$$evalProg \quad :: \quad \langle \forall v :: v \leftarrow List.v \leftarrow Prog \leftarrow ConstEnv.v \rangle.$$

The additional parameter of type $ConstEnv.v$ is an environment defining the constants. The arguments to *main* are passed as a list of type $List.v$ and the result is a value of type v. The additional parameterisation means that the interpreter can be used to evaluate programs over different semantic domains by substituting different value types and different constant environments.

7.4. Environments

An environment is a function or table that defines the values of variables and constants. In the interpreter, three different environments are used: one for the constants, one for the functions and one for variables.

The constants environment has the following type:

$$\textbf{type}\ ConstEnv.v = v \leftarrow List.v \leftarrow String.$$

Its inputs are the name of the constant and a (possibly empty) list of parameter values. Its output is the value computed by the constant. As discussed above, the interpreter is polymorphic in the value type v. Different constant environments can therefore use different types. To evaluate trivial examples, like factorial, the only values that are needed are integers and Booleans, so the following datatype suffices:

$$\textbf{data}\ Value = Bot \mid IVal\ Int \mid BVal\ Bool \mid Top.$$

As well as integers and Booleans, the values *Bot* and *Top* are included. This ensures that *Value* is a complete lattice, so that it can serve as a semantic domain. The partial order is:

$$x \preccurlyeq y \quad \equiv \quad x = Bot \vee y = Top \vee x = y.$$

Bot is the semantic representation of non-termination and failure. *Top*, on the other hand, has no further meaning; its only purpose is to make *Value* a complete lattice.

The function below defines a simple environment of constants, based on the datatype *Value*:

$$\begin{array}{lll}
consts & & :: ConstEnv.Value. \\
consts\ .\text{“}\cdot\text{”} & .[Bot, y] & = Bot, \\
consts\ .\text{“}\cdot\text{”} & .[x, Bot] & = Bot, \\
consts\ .\text{“}\cdot\text{”} & .[IVal.x, IVal.y] & = IVal.(x \cdot y), \\
consts\ .\text{“}-\text{”} & .[Bot, y] & = Bot, \\
consts\ .\text{“}-\text{”} & .[x, Bot] & = Bot, \\
consts\ .\text{“}-\text{”} & .[IVal.x, IVal.y] & = IVal.(x - y), \\
consts\ .\text{“}\geqslant\text{”} & .[Bot, y] & = Bot, \\
consts\ .\text{“}\geqslant\text{”} & .[x, Bot] & = Bot, \\
consts\ .\text{“}\geqslant\text{”} & .[IVal.x, IVal.y] & = BVal.(x \geqslant y), \\
consts\ .\text{“if”} & .[Bot, x, y] & = Bot, \\
consts\ .\text{“if”} & .[BVal.b, x, y] & = \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y, \\
consts\ .\text{“true”} & .[] & = BVal.True, \\
consts\ .\text{“false”} & .[] & = BVal.False, \\
consts\ .\text{“error”} & .xs & = Bot, \\
consts\ .num & .[] & = IVal.(intOfString.num), \\
consts\ .s & .[\ldots, Top, \ldots] & = Top, \\
consts\ .s & .xs & = Bot.
\end{array}$$

Pattern matching is used in this definition, with clauses higher up the list taking higher priority. For example, the multiplication operator $*$ fails if either operand has failed, so the first two clauses of its definition check that neither operand is *Bot*. The penultimate clause states that if any of the parameters is *Top*, then the result is *Top* (provided that no earlier clause is applicable). This clause is required for the function to be monotonic, because, for instance, without it $Top * Top$ would equal *Bot*, which is less than the result of $IVal\ 2 * IVal\ 3$. Additionally, a couple of informal shortcuts have been used, for brevity. The text “*num*”, in the last-but-two clause, matches any string of digits. Also, if an undefined constant is used, or the wrong number of parameters is passed to a constant, pattern matching falls through to the final clause and the result is *Bot* (unless one of the list elements is *Top*). The constant “error” is provided so that the interpreter can itself signal an error, for example when an undefined variable is used.

The environment for functions has the same type as the environment for constants:

$$\textbf{type}\ FunEnv.v = v \leftarrow List.v \leftarrow String.$$

The environment for variables is represented as an association list, so that new bindings can be easily added, by appending them to the front of the list:

type $VarEnv.v = List.(v \times String)$.

A function called *lookup* is used to find values in association lists. It has type:

$\langle \forall v :: v \leftarrow String \leftarrow VarEnv.v \leftarrow v \rangle$.

The first parameter is a failure value to be returned if the string is not found in the variable environment. Parametric polymorphism of *lookup* is facilitated by including "error" in the constants environment; the failure value is the interpretation of the "error" constant.

Finally, it is sometimes convenient to package the three environments as a tuple:

type $Envs.v = ConstEnv.v \times FunEnv.v \times VarEnv.v$.

7.5. Interpreters for expressions, functions and programs

Expressions, functions and programs are syntactically distinct in the first-order language, so it is easiest to define three interpreters: one for expressions, one for functions and one for programs. The expression and function interpreters are subroutines of the program interpreter.

Expressions depend on all three kinds of environment, so the expression interpreter *evalExp* is passed a tuple of type *Envs.v*:

$$
\begin{array}{l}
\quad evalExp \quad :: \quad \langle \forall v :: v \leftarrow Exp \leftarrow Envs.v \rangle. \\
evalExp.envs.e \quad = \\
\quad \textbf{let } (constEnv, funEnv, varEnv) = envs \textbf{ in} \\
\quad \textbf{case } e \textbf{ of} \\
\qquad Const.f.es \rightarrow constEnv.f.(map.(evalExp.envs).es), \\
\qquad Call.f.es \rightarrow funEnv.f.(map.(evalExp.envs).es), \\
\qquad Var.x \rightarrow lookup.(constEnv.\text{“error”}.[\,]).varEnv.x.
\end{array}
$$

If the expression is a constant, the interpreter first evaluates the arguments (by calling itself recursively) and then invokes the environment of constants. Similarly, if the expression is a function, the interpreter first evaluates the arguments and then invokes the environment of functions. Finally, if the expression is a variable then its value is obtained from the environment of variables; should the variable not be found, the result is the interpretation of the "error" constant.

The task of the function interpreter is to construct an entry for the function environment. Functions can, however, call each other, so the function environment is also a parameter of the function interpreter! (This circularity is resolved later by the program interpreter, *evalProg*.)

$$
\begin{array}{lll}
evalFun & :: & \langle \forall v :: ((v \leftarrow List.v) \times String) \leftarrow FunDef \leftarrow \\
 & & \qquad FunEnv.v \leftarrow ConstEnv.v \rangle.
\end{array}
$$

$$evalFun\,.\,constEnv\,.\,funEnv\,.\,(name,\ params,\ e)\ =\ (\langle \lambda\, vs :: evalExp\,.\,(constEnv,\ funEnv,\ zip\,.\,vs\,.\,params).e\rangle,\ name).$$

An entry in the function environment consists of the function's name and its value. Its value is a function: given a list of values *vs* (the parameters to the function), it computes the value of the body of the function, with the values of the parameters bound in the variable environment. The function *zip* (see the Haskell Report [20, p. 110]) is used to construct the variable environment.

The interpreter for programs uses *evalExp* and *evalFun* as subroutines and is parameterised by the environment of constants:

$$\begin{aligned} &evalProg\ ::\ \langle \forall v :: v \leftarrow List.v \leftarrow Prog \leftarrow ConstEnv.v\rangle.\\ &evalProg\,.\,constEnv\,.\,p\ =\\ &\quad \textbf{let}\ fail = constEnv.\text{“error”}\ \textbf{in}\\ &\quad \textbf{let}\ funEnv =\\ &\qquad lookup\,.\,fail\,.\,(map\,.\,(evalFun\,.\,constEnv\,.\,funEnv)\,.\,p)\\ &\qquad\qquad \textbf{in}\ funEnv\,.\,\text{“main”}. \end{aligned}$$

The program is evaluated by evaluating the *main* function, so this definition sets up the function environment and then applies it to the string "main". Functions are allowed to be mutually recursive, so *evalFun* needs access to the full function environment. This leads to a cyclic definition of *funEnv*; formally, *funEnv* is the least fixed point of the equation. The type v must, therefore, be a poset in which least fixed points exist (such as the datatype *Value*, defined earlier). Moreover, application of "safety-for-free" depends on the validity of Theorem 48.

7.6. Abstraction and concretisation functions for strictness analysis

In abstract interpretation, Galois connections are used to relate the *concrete* domain to the *abstract* domain. In the interpreter for the first-order language, the concrete domain is the type *Value*. The abstract domain used in strictness analysis is the Boolean domain: *False* represents *Bot* and *True* represents any other value. Booleans form a complete lattice with implication as the ordering: a is less than or equal to b if $a \Rightarrow b$. (In other words, *False* is less than *True*.) The abstraction function converts values from concrete to abstract:

$$\begin{aligned} &abs\ ::\ Bool \leftarrow Value.\\ &abs\,.\,Bot = False,\\ &abs\,.\,v\ = True. \end{aligned}$$

Information is lost going from concrete to abstract, so the concretisation function has to be conservative, meaning that it computes a value that is greater than or equal to the original value.

$$\begin{aligned} &con\ ::\ Value \leftarrow Bool.\\ &con\,.\,False = Bot,\\ &con\,.\,True\ = Top. \end{aligned}$$

It is a straightforward case analysis to prove that *abs* and *con* form a Galois connection:

$$\langle \forall v, b :: abs.v \Rightarrow b \quad \equiv \quad v \preccurlyeq con.b \rangle.$$

In the following section, this Galois connection is used to derive a strictness analysis for the first-order language.

7.7. Using safety-for-free

The derivation of the abstract interpreter is a straightforward application of safety-for-free. Recall the type of *evalProg*:

$$evalProg \quad :: \quad \langle \forall v :: v \leftarrow List.v \leftarrow Prog \leftarrow ConstEnv.v \rangle.$$

Therefore, safety-for-free states that, if f_a and g_a form a Galois connection for all type variables a,

$$evalProg \quad \preccurlyeq_t \quad [g, f^{\cup}]_t.evalProg, \tag{51}$$

with $t = a \leftarrow List.a \leftarrow Prog \leftarrow ConstEnv.a$. Lemma 45 can be used to simplify the ordering $\preccurlyeq_t$:

$$\preccurlyeq_{a \leftarrow List.a \leftarrow Prog \leftarrow ConstEnv.a} \; = \; \preccurlyeq_{a \leftarrow \mathsf{id} \leftarrow \mathsf{id} \leftarrow \mathsf{id} \leftarrow \mathsf{id}} \; .$$

Therefore, (51) is equivalent to:

$$\langle \forall cs, p, vs :: evalProg.cs.p.vs \preccurlyeq_a [g, f^{\cup}]_t.(evalProg.cs.p.vs) \rangle. \tag{52}$$

All that remains is to expand the definition of the function $[g, f^{\cup}]_t$. Using the rules of the operator $[\_, \_]$, this is an entirely mechanical expansion:

$$\begin{array}{ll}
 & [g, f^{\cup}]_t.evalProg.consts.p.vs \\
= & \quad \{t = a \leftarrow List.a \leftarrow Prog \leftarrow ConstEnv.a, \\
 & \quad\quad (17),\ (5) \text{ and } (20), \\
 & \quad\quad \text{where } consts' = [f, g^{\cup}]_{ConstEnv.a}.consts\} \\
 & [g, f^{\cup}]_{a \leftarrow List.a \leftarrow Prog}.(evalProg.consts').p.vs \\
= & \quad \{(17),\ (5),\ (20) \text{ and } (18), \\
 & \quad\quad Prog \text{ is a zero-ary relator,} \\
 & \quad\quad \text{so } [f, g^{\cup}]_{Prog} \text{ is the identity function}\} \\
 & [g, f^{\cup}]_{a \leftarrow List.a}.(evalProg.consts'.p).vs \\
= & \quad \{(17),\ (15),\ (20) \text{ and } (18)\} \\
 & g_a.(evalProg.consts'.p.(List.f_a.vs))
\end{array}$$

The result is that:

$$
\begin{array}{ll}
 & [g, f^{\cup}]_t.evalProg.consts.p.vs \\
= & \qquad \{(16),\ (17),\ (18),\ (5) \text{ and } (20)\} \\
 & g_a.(evalProg.consts'.p.(List.f_a.vs))
\end{array}
$$

where

$$consts' = [f, g^{\cup}]_{ConstEnv.a}.consts.$$

Recalling that $ConstEnv.a$ is a type synonym for $a \leftarrow List.a \leftarrow String$, the definition of $consts'$ expands to

$$consts'.s = f_a \bullet consts.s \bullet List.g_a.$$

Substituting the functions *abs* and *con*, defined in Section 7.6, for f_a and g_a, *evalProg* satisfies the following property, for all *fail*, *consts*, and *p*:

$$evalProg.consts.p.vs \preccurlyeq con.(evalProg.consts'.p.(List.abs.vs)) \tag{53}$$

where

$$consts'.s = abs \bullet consts.s \bullet List.con.$$

This property formalises the correspondence between *evalProg.consts*, which is the concrete interpreter of the language, and *evalProg.consts'*, which is the abstract interpreter. The former evaluates a program and produces a *Value*. The latter evaluates a program and produces a *Bool*. The property formalises the notion that the abstract interpreter is a *safe* approximation of the concrete interpreter. This means that the results of the abstract interpreter are never wrong, but they may sometimes be overly conservative. Suppose *b* is the Boolean value produced by the abstract semantics. If *b* is *False*, the program will always evaluate to *Bot*. On the other hand, if *b* is *True* then the program will evaluate to a value at most *Top*. Every value (including *Bot*) is at most *Top*, so this statement says nothing about the behaviour of the program. The analysis is *conservative*, because it might compute *True* when the program actually evaluates to *Bot*.

Expanding safety-for-free on concrete examples such as *evalProg* is an entirely mechanical process, and a tool that does this is likely to be simple to build.

7.8. Constants for the abstract semantics

The only work that remains to be done is the derivation of *consts'* using the specification given above. The definitions of *consts*, *abs* and *con* are all known, so this is

a straightforward expansion, leading to the following result:

$$
\begin{array}{llll}
\mathit{consts}' & & & :: \mathit{ConstEnv}.\mathit{Bool}. \\
\mathit{consts}' & ."\cdot" & .[\mathit{False}, y] & = \mathit{False}, \\
\mathit{consts}' & ."\cdot" & .[x, \mathit{False}] & = \mathit{False}, \\
\mathit{consts}' & ."-" & .[\mathit{False}, y] & = \mathit{False}, \\
\mathit{consts}' & ."-" & .[x, \mathit{False}] & = \mathit{False}, \\
\mathit{consts}' & ."\geqslant" & .[\mathit{False}, y] & = \mathit{False}, \\
\mathit{consts}' & ."\geqslant" & .[x, \mathit{False}] & = \mathit{False}, \\
\mathit{consts}' & ."\text{if}" & .[\mathit{False}, x, y] & = \mathit{False}, \\
\mathit{consts}' & ."\text{true}" & .[] & = \mathit{True}, \\
\mathit{consts}' & ."\text{false}" & .[] & = \mathit{True}, \\
\mathit{consts}' & ."\text{error}" & .xs & = \mathit{False}, \\
\mathit{consts}' & .num & .[] & = \mathit{True}, \\
\mathit{consts}' & .s & .[\ldots, \mathit{True}, \ldots] & = \mathit{True}, \\
\mathit{consts}' & .s & .xs & = \mathit{False}.
\end{array}
$$

Evaluating the factorial function (p. 35) with these constants produces a function taking $[\mathit{Top}]$ to Top and $[\mathit{Bot}]$ to Bot, as expected. This indicates that the factorial program is strict, which means that it can be implemented without a thunk. Applying the analysis to the incorrect factorial function (p. 35) produces a function that takes $[\mathit{Top}]$ to Bot and $[\mathit{Bot}]$ to Bot, which indicates that the function always fails to terminate. Finally, applying the analysis to the non-strict multiplication example (p. 35) produces a function that takes $[\mathit{Top}, \mathit{Top}]$ to Top, $[\mathit{Top}, \mathit{Bot}]$ to Top, $[\mathit{Bot}, \mathit{Top}]$ to Bot, and $[\mathit{Bot}, \mathit{Bot}]$ to Bot. This indicates that the program is strict in its first argument, but says nothing about its behaviour with respect to its second argument.

8. Conclusion

We have shown how to extend a collection of Galois connections to a Galois connection of arbitrary higher-order type. The construction is more general than any that we are aware of. In addition, we have shown that the construction is defined by a logical relation, hopefully clarifying misunderstandings about the relationship between higher-order Galois connections, their safety properties and logical relations.

Use of our theorems is facilitated by a programming methodology that emphasises the decomposition of programs into specific and parametric components. We have demonstrated this methodology with two substantial examples of its application to abstraction interpretation. We envisage that the methodology can be profitably applied in other ways, for example to program refinement, and would encourage further research in this direction.

The calculations we have presented demonstrate the effectiveness of point-free relation algebra. Avoidance of unnecessary quantifications, with the accompanying bound variables, keeps details to a minimum. It is likely that our calculations would have been substantially longer if presented at the same level of formality using pointwise arguments.

Acknowledgements

Our thanks go to the anonymous referees for their very careful and thorough review.

Appendix

Suppose a function is parametrically polymorphic, with type

$$(a \leftarrow b) \leftarrow (a \leftarrow c) \times (c \leftarrow b)$$

for all types a, b and c. For clarity, we give the function the name "*comp*", but also write $f \circ g$ for its application to functions f and g. Its dinaturality property is derived as follows. We let f range over assignments of total functions to the type variables a, b and c. Then

$$\begin{array}{ll}
 & comp \in \langle \forall a,b,c :: \ (a \leftarrow b) \leftarrow (a \leftarrow c) \times (c \leftarrow b) \rangle \\
\Rightarrow & \quad \{\text{dinaturality: Corollary 34} \\
 & \qquad \text{with } u,v := a \leftarrow b, (a \leftarrow c) \times (c \leftarrow b)\} \\
 & \quad \langle \forall f :: \\
 & \qquad [\mathsf{id}, f^{\cup}]_{a \leftarrow b} \bullet comp \bullet [f, \mathsf{id}]_{(a \leftarrow c) \times (c \leftarrow b)} \\
 & \qquad = \quad [f, \mathsf{id}]_{a \leftarrow b} \bullet comp \bullet [\mathsf{id}, f^{\cup}]_{(a \leftarrow c) \times (c \leftarrow b)} \\
 & \quad \rangle \\
= & \quad \{\text{applying (17) and (18), followed by (5)}: \\
 & \qquad [\mathsf{id}, f^{\cup}]_{a \leftarrow b} = \mathsf{id} \leftarrow (f_b)^{\cup} = (\bullet f_b), \\
 & \qquad [f, \mathsf{id}]_{(a \leftarrow c) \times (c \leftarrow b)} = (f_a \leftarrow \mathsf{id}) \times (f_c \leftarrow \mathsf{id}) = (f_a \bullet) \times (f_c \bullet), \\
 & \qquad [f, \mathsf{id}]_{a \leftarrow b} = f_a \leftarrow \mathsf{id} = (f_a \bullet), \\
 & \qquad [\mathsf{id}, f^{\cup}]_{(a \leftarrow c) \times (c \leftarrow b)} = (\mathsf{id} \leftarrow (f_c)^{\cup}) \times (\mathsf{id} \leftarrow (f_b)^{\cup}) = (\bullet f_c) \times (\bullet f_b). \\
 & \qquad \} \\
 & \quad \langle \forall f :: (\bullet f_b) \bullet comp \bullet (f_a \bullet) \times (f_c \bullet) = (f_a \bullet) \bullet comp \bullet (\bullet f_c) \times (\bullet f_b) \rangle \\
= & \quad \{\text{equality of functions}\} \\
 & \quad \langle \forall f,g,h :: \\
 & \qquad ((\bullet f_b) \bullet comp \bullet (f_a \bullet) \times (f_c \bullet)).(g,h) \\
 & \qquad = \quad ((f_a \bullet) \bullet comp \bullet (\bullet f_c) \times (\bullet f_b)).(g,h) \\
 & \quad \rangle \\
= & \quad \{\text{definitions of } \times, (k\bullet) \text{ and } (\bullet k), \text{ and,} \\
 & \qquad \text{for all } k, \ m, \ comp.(k,m) = k \circ m\} \\
 & \quad \langle \forall f,g,h :: ((f_a \bullet g) \circ (f_c \bullet h)) \bullet f_b = f_a \bullet ((g \bullet f_c) \circ (h \bullet f_b)) \rangle.
\end{array}$$

References

[1] S. Abramsky, Abstract interpretation, logical relations, and Kan extensions, J. Logic Comput. 1 (1) (1990) 5–41.

[2] K.S. Backhouse, A functional semantics of attribute grammars, in: International Conference on Tools and Algorithms for Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 2280, Springer, Berlin, 2002, pp. 142–157.

[3] K. Backhouse, Abstract interpretation of domain-specific embedded languages, Ph.D. Thesis, Computing Laboratory, University of Oxford, 2002.
[4] R.C. Backhouse, On a relation on functions, in: W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, J. Misra (Eds.), Beauty is Our Business, Springer, Berlin, 1990.
[5] R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, J. van der Woude, Polynomial relators, in: M. Nivat, C.S. Rattray, T. Rus, G. Scollo (Eds.), Proc. Second Conf. on Algebraic Methodology and Software Technology, AMAST'91, Workshops in Computing, Springer, Berlin, 1992, pp. 303–326.
[6] R.C. Backhouse, T.S. Voermans, J. van der Woude, A relational theory of datatypes, http://www.cs.nott.ac.uk/~rcb/MPC/papers, December 1992.
[7] P.J. de Bruin, Inductive types in constructive languages, Ph.D. Thesis, Rijksuniversiteit, Groningen, 1995.
[8] G.L. Burn, C. Hankin, S. Abramsky, Strictness analysis for higher-order functions, Sci. Comput. Programming 7 (1986) 249–278.
[9] P. Cousot, R. Cousot, Abstract interpretation: a unifed lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conf. Record of the Fourth Annu. ACM Symp. on Principles of Programming Languages, Los Angeles, CA, January 1977, pp. 238–252.
[10] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Conf. Record of the Sixth Annu. ACM Symp. on Principles of Programming Languages, San Antonio, TX, January 1979, pp. 269–282.
[11] P. Cousot, R. Cousot, Higher-order abstract interpretations (and application to comportment analysis generalizing strictness, termination, projection and per analysis of functional languages), in: Proc. ICCL'94, IEEE, 1994, pp. 95–112.
[12] E.W. Dijkstra, C.S. Scholten, Predicate Calculus and Program Semantics, Texts and Monographs in Computer Science, Springer, Berlin, 1990.
[13] J. Hartmanis, R.E. Stearns, Pair algebras and their application to automata theory, Inform. and Control 7 (4) (1964) 485–507.
[14] J. Hartmanis, R.E. Stearns, Algebraic Structure Theory of Sequential Machines, Prentice-Hall, Englewood Cliffs, NJ, 1966.
[15] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, Englewood Cliffs, NJ, 1985.
[16] P. Hoogendijk, A generic theory of datatypes, Ph.D. Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1997.
[17] R. Milner, A theory of type polymorphism in programming, J. Comput. Syst. Sci. 17 (1978) 348–375.
[18] A. Mycroft, Abstract interpretation and optimising transformations for applicative programs, Ph.D. Thesis, Department of Computer Science, University of Edinburgh, 1981.
[19] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer, Berlin, 1998.
[20] S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, P. Wadler, Report on the programming language Haskell 98, http://www.haskell.org/, 1999.
[21] S.L. Peyton Jones, The implementation of functional programming languages, Series in Computer Science, Prentice-Hall, Englewood Cliffs, NJ, 1987.
[22] G.D. Plotkin, Lambda-definability in the full type hierarchy, in: J.P. Seldin, J.R. Hindley (Eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, London, 1980.
[23] J.C. Reynolds, Types, abstraction and parametric polymorphism, in: R.E. Mason (Ed.), IFIP '83, Elsevier, Amsterdam, 1983, pp. 513–523.
[24] P. Wadler, Theorems for free!, in: Fourth Symp. on Functional Programming Languages and Computer Architecture, ACM, London, September 1989.