Abstract Interpretations for Free

Kevin Backhouse[*] Roland Backhouse[†]

January 10, 2002

Abstract

A construction is given for extending Galois connections between base types (eg booleans and integers) to Galois connections between datatypes of arbitrary complexity. "Theorems-for-free" is used to show that the construction ensures safe abstract interpretation of parametrically polymorphic functions.

1 Introduction

Abstract interpretation is a technique for approximating the execution behaviour of a computer program. It is used, for example, for strictness analysis of functional 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. In this sense the approximation is "safe". The basis of abstract interpretation is the notion of a Galois connection between ordered sets [CC77, CC79].

"Theorems-for-free" [Rey83, Wad89] is a remarkable feature of parametrically polymorphic functions, namely that knowing just the type of a function it is possible to derive algebraic properties of the function. For example, it is possible to show that any function that has the same type as function composition must be associative.

In this paper we show that safety properties of abstract interpretations are "free theorems". The basis of this result is a construction for extending a collection of Galois connections on base types (eg booleans and integers) to Galois connections between datatypes of arbitrary complexity. This construction is then combined with "theorems-for-free" to establish the safety property.

[*]Computing Laboratory, University of Oxford, Oxford OX3 6QD, England, email: kevinb@comlab.ox.ac.uk, fax: +44(0) 1865 273839

[†]School of Computer Science and Information Technology, University of Nottingham, Nottingham NG8 1BB, England, email: rcb@cs.nott.ac.uk, fax: +44(0) 115 951 4254

There are several practical applications of this result. The theorem can be used to construct abstract interpretations. The theorem can also be used to focus the testing of programs (as in model checking) on those properties that are not for free. Most importantly, the theorem suggests a design principle for the development of programs involving a separation into generic and non-generic components. This latter aspect is illustrated briefly in section 6.

Connections between abstract interpretations and "theorems-for-free" —more specifically, so-called "logical relations"— have been observed by Abramsky [Abr90]; he proves a theorem (the "uniformization theorem") which is similar to our extension of base Galois connections to Galois connections between datatypes of arbitrary complexity. Also, a generic safety property of abstract interpretations has been proved by Burn, Hankin and Abramsky [BHA86]. However, so far as we can determine, the proof of safety properties by exploiting "theorems-for-free" is novel.

We have chosen in this paper to relegate proofs to an appendix. This has been done for two reasons. First, we want to emphasise the applicability of the results. Second, the beautifully concise relation-algebraic proofs are relatively unfamiliar in the literature and, for this reason, require some preparatory explanation before they can be presented. We have also chosen to give brief introductions to both abstract interpretations and to theorems-for-free in the hope that the paper will also be accessible to readers familiar with one but not the other (and possibly readers familiar with neither). The examples we develop in these introductions are expanded later.

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 $[m_0, \ldots, m_{n-1}]$ of a's, evaluates $((m \oslash m_0) \oslash \ldots) \oslash m_{n-1}$. So $\mathit{fold}.0.(+).ms$ sums a list ms of integer values, $\mathit{fold}.1.(\times).ms$ evaluates the product of the list of values, $\mathit{fold}.\mathit{true}.(\equiv).bs$ evaluates the associative (as opposed to conjunctional) equivalence of the list of booleans bs, and $\mathit{fold}.\mathit{false}.(\vee).bs$ 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

$$18975 \times 3 \times 2 \times 21$$

is even without performing the integer division. "Theorems-for-free" predicts that this can be done by evaluating

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

Formally, a "theorem for free" is

$$even.(\mathsf{fold}.1.(\times).ms) \;=\; \mathsf{fold}.\mathit{false}.(\vee).(\mathsf{List}.even.ms)$$

where $\mathsf{List}.even.ms$ maps the function $even$ over the list of numbers ms. (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.) Similarly, "theorems-for-free" also predicts that

$$even.(18975{+}3{+}2{+}21) \;=\; (even.18975 \equiv even.3 \equiv even.2 \equiv even.21) \quad .$$

Formally,

$$even.(\mathsf{fold}.0.(+).ms) \;=\; \mathsf{fold}.\mathit{true}.(\equiv).(\mathsf{List}.even.ms) \quad .$$

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". Then we will show that theorems-for-free predicts that

$$dk.(\mathsf{fold}.1.(\times).ms) \;\Leftarrow\; \mathsf{fold}.(k{=}1).(\otimes).(\mathsf{List}.dk.ms)$$

where

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

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 theorem-for-free here is an "if" ($\Leftarrow$) rather than an equality. We say that $\mathsf{fold}.(k{=}1).(\otimes).(\mathsf{List}.dk.ms)$ is an *abstract interpretation* of $dk.(\mathsf{fold}.1.(\times).ms)$ and its result is *safe* in the sense that if the abstract interpretation evaluates to true then it is the case that the given product is divisible by k; however, if the abstract interpretation evaluates to false then 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 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

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, as proposed by Hartmanis and Stearns [HS64, HS66] in what is most probably the very first application of Galois connections in the computing science literature.

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

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

and

$$(x,y) \in R \quad \equiv \quad x \preceq f^\sharp.y \quad .$$

We say that the pair of functions $(f, f^\sharp)$ is *a Galois connection between the posets* $(\mathcal{A}, \sqsubseteq)$ *and* $(\mathcal{B}, \preceq)$ if $f \in \mathcal{A} \leftarrow \mathcal{B}$ and $f^\sharp \in \mathcal{B} \leftarrow \mathcal{A}$ and, for all x in $\mathcal{B}$ and all y in $\mathcal{A}$,

$$f.x \sqsubseteq y \quad \equiv \quad x \preceq f^\sharp.y \quad .$$

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 involved 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,

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

(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 $true$.

Theorems for Free Based on Reynolds' abstraction theorem [Rey83], Wadler [Wad89] showed how to derive a theorem about a polymorphic function from its type. The key to such "theorems for free" 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 $A\leftarrow B$ and $C\leftarrow D$, respectively, to a binary relation $R\leftarrow S$ on functions f and g of type $A\leftarrow C$ and $B\leftarrow D$, respectively. Formally, $R\leftarrow S$ is defined by

(1) $\qquad (f,g)\in R\leftarrow S \quad\equiv\quad \langle\forall u,v :: (f.u\,,\,g.v)\in R \Leftarrow (u,v)\in S\rangle \quad .$

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

As an example, suppose $(\mathcal{A},\sqsubseteq)$ and $(\mathcal{B},\preceq)$ are partially ordered sets. Then we can instantiate R to $\sqsubseteq$ and S to $\preceq$ getting a relation $\sqsubseteq\leftarrow\preceq$ between functions f and g of type $\mathcal{A}\leftarrow\mathcal{B}$. In particular, switching to the usual infix notation for membership of an ordering relation,

$$(f,f)\in\,\sqsubseteq\leftarrow\preceq \quad\equiv\quad \langle\forall u,v :: f.u\sqsubseteq f.v \Leftarrow u\preceq v\rangle \quad .$$

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

It is useful to abbreviate a property of the form $(x,x)\in R$ to $x\in R$. In this way, the statement $f\in\,\sqsubseteq\leftarrow\preceq$ means that f is a monotonic function, the ordering relations on the range and domain of f being, respectively, $\sqsubseteq$ and $\preceq$.

Particularly relevant to this paper is that, if f and g are monotonic functions,

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

(The easy proof is left to the reader.) We thus recognize $\sqsubseteq\leftarrow\preceq$ as the usual pointwise ordering on monotonic functions of type $(\mathcal{A},\sqsubseteq)\leftarrow(\mathcal{B},\preceq)$.

"Theorems for free" is the property that, if f 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 $f\in t[a,\ldots,c := R_a,\ldots,R_c]$. An example of a "free theorem" is obtained by considering the composition operator on functions, which we denote by a raised infix dot as in " $f\bullet g$ ". Composition has parametric type

$$(a{\leftarrow}b) \leftarrow (a{\leftarrow}c) \leftarrow (c{\leftarrow}b) \quad .$$

The free theorem is that, for all relations R, S and T,

$$\bullet \;\in\; (R{\leftarrow}S) \leftarrow (R{\leftarrow}T) \leftarrow (T{\leftarrow}S) \quad .$$

In particular, substituting ordering relations $\sqsubseteq$, $\preceq$ and $\leq$ for R, S and T, respectively,

$$\bullet \;\in\; (\sqsubseteq \leftarrow \preceq) \leftarrow (\sqsubseteq \leftarrow \leq) \leftarrow (\leq \leftarrow \preceq) \quad .$$

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

$$f{\bullet}g \in \sqsubseteq \leftarrow \preceq \quad \Leftarrow \quad f \in \sqsubseteq \leftarrow \leq \;\wedge\; g \in \leq \leftarrow \preceq \quad .$$

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 function that has the same type as function composition will preserve the monotonicity property.

Note that, if we were to be completely precise, we should distinguish between the different instances of function composition. Also, if f is a parametrically polymorphic function of type t then the free theorem is actually that the pair $(f_{\mathcal{A}}, f_{\mathcal{B}})$ is an element of relation $t[a,\ldots,c := R_a, \ldots, R_c]$ where the subscripts indicate how the particular instances of f are determined (depending on the types of the relations R). These details are omitted in this discussion.

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[1]. For functions f and g, $f \leftarrow g^{\cup}$ is the function defined by

$$(f \leftarrow g^{\cup}).h \;=\; f{\bullet}h{\bullet}g \quad .$$

(The easy proof is left to the reader.)

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 Roland Backhouse to propose the notion of a *relator*[2] as the basis for a relational theory of datatypes [BBH+92, BVW92]. Briefly, a relator is a monotonic functor that commutes with converse. 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

[1]Functions are regarded as special cases of binary relations. For function f, $(x,y) \in f$ is the same as $x = f.y$. The converse $g^{\cup}$ of function g is thus such that $(x,y) \in g^{\cup}$ is the same as $y = g.x$.

[2]Not be confused with Abramsky and Jensen's [AJ90] and similar proposals made at about the same time.

source allegory to objects of the target allegory and the second element is from arrows (relations) of the source allegory to arrows of the target allegory. The mappings are required to have the properties that they form a functor from the underlying source category to the underlying target category and, for all arrows R and S,

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

and

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

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 ($n \geq 0$) and will write $F.(u \ldots v)$ for the application of the relator to arguments u, ..., v. Examples used later are the unary relator $List$, the binary relator $\times$, and a zero-ary relator $Production$. (Every object in an allegory defines a zero-ary relator. On arrows the relator maps each arrow to the identity arrow on the object.)

In this paper, we will 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. 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.

For more background on relators see [BBH+92, BVW92, BdM96]. For a formal treatment of relators in their full generality see [Hoo97].

Running Example Returning to our running example, the function $\mathtt{fold}$ has type

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

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

$$\mathsf{fold} \in P \leftarrow P \leftarrow (P \leftarrow P \times P) \leftarrow \mathsf{List}.P \quad .$$

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

The theorem we present predicts first that the function

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

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

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

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

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

(ordered pointwise by divisibility). The upper adjoint is the function

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

Our second theorem says that the free theorem for $\mathtt{fold}$ instantiated with the pair algebra for the $(dk\,,\,kd)$ Galois connection has the corollary 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 all $ms \in \mathsf{List}.\mathsf{PosInt}$,

$$dk.ConcreteValue \;\Leftarrow\; AbstractValue$$

where

$$ConcreteValue \;=\; \mathsf{fold}.m.(\otimes).ms$$

and

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

Substituting integer multiplication[3] for $\otimes$ we calculate that, for all boolean values b and c,

[3] denoted here by the symbol "$*$" in order to distinguish it from "$\times$", meaning cartesian product

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

In this way we have calculated 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 Types and Assignments to Variables

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

$$
TypeExp \quad ::= \quad (Relator\ TypeExp^{*}) \mid (TypeExp \leftarrow TypeExp) \mid Variable
$$

where $Variable$ and $Relator$ are (disjoint) finite sets. Variables are place-holders to be filled in various ways, as explained in detail later. Each element of $Relator$ denotes a relator of a certain arity. Examples include $\mathbb{1}$, which denotes the relator of arity zero corresponding to the unit type, and $List$, which denotes the unary relator corresponding to the list type constructor. Binary relators include cartesian product and disjoint sum.

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

$$
a \times List.b \ \leftarrow\ b \times List.a \ \leftarrow\ Production \quad .
$$

This uses the binary product relator, $\times$, the unary relator $List$ and the zero-ary relator $Production$. It describes the type of a semantic rule in an attribute grammar, with a and b being place-holders for the types of the synthesised and inherited attributes, respectively. (A semantic rule specifies, for each production in the attribute grammar, a function from an inherited attribute and a list of synthesised attributes to a synthesised attribute and a list of inherited attributes.)

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. A variable assignment will be denoted by an assignment statement as in, for example,

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

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

In the case of variable assignments whose ranges are sets of relations we need to define the converse and composition operators. The definitions are:

$$(2) \qquad (V^{\cup})_a \;=\; (V_a)^{\cup}$$

and

$$(3) \qquad (V \bullet W)_a \;=\; V_a \bullet W_a \quad .$$

We extend variable assignments to type expressions in two ways. The simpler of the two extends a variable assignment V inductively to all type expressions t: for all type expressions u, v

$$(4) \qquad V_{u \leftarrow v} \;=\; V_u \leftarrow V_v$$

and, for all type expressions u, ..., v,

$$(5) \qquad V_{F.(u \ldots v)} \;=\; F.(V_u \ldots V_v) \quad .$$

Note that these definitions make sense whenever the range of V is an assignment of posets or relations to the type variables. In the case that V is a poset assignment we define the $\leftarrow$ operator to map posets $\mathcal{A}$ and $\mathcal{B}$ into the *monotonic* functions mapping values of poset $\mathcal{B}$ into values of poset $\mathcal{A}$.

The more complicated extension uses two 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. Then the variable assignment $[V,W]$ is defined inductively as follows:

$$(6) \qquad [V,W]_a \;=\; V_a \quad ,$$

$$(7) \qquad [V,W]_{u \leftarrow v} \;=\; [V,W]_u \leftarrow [W,V]_v \quad ,$$

$$(8) \qquad [V,W]_{F.(u \ldots v)} \;=\; F.([V,W]_u \ldots [V,W]_v) \quad .$$

For example,

$$[(\mathrm{a} := \mathrm{dk}),(\mathrm{a} := \mathrm{kd}^{\cup})]_{\mathrm{a} \leftarrow \mathrm{a} \leftarrow (\mathrm{a} \leftarrow \mathrm{a}\times\mathrm{a}) \leftarrow \mathrm{List.a}}$$
$$= \mathrm{dk} \leftarrow \mathrm{kd}^{\cup} \leftarrow (\mathrm{kd}^{\cup} \leftarrow \mathrm{dk}\times\mathrm{dk}) \leftarrow \mathrm{List.kd}^{\cup} \quad .$$

From the above definitions, the following properties are easily proved for all type expressions t by induction on the structure of type expressions.

(9) $\quad V_t = [V,V]_t \quad ,$

(10) $\quad (V_t)^{\cup} = (V^{\cup})_t \quad ,$

(11) $\quad ([V,W]_t)^{\cup} = [V^{\cup}, W^{\cup}]_t \quad ,$

Because of (10) 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 $[V^{\cup}, W^{\cup}]$.

Note that it is not the case that $(V \bullet W)_t = V_t \bullet W_t$ for all type expressions t. Property (3) is only valid for type variables a.

5 Main Theorem

Let us suppose that an assignment of Galois connections ($\mathrm{abs}_a, \mathrm{con}_a$) between the posets ($\mathcal{A}_a, \sqsubseteq_a$) and ($\mathcal{B}_a, \preceq_a$) is given for each type variable a. In other words, we are given six functions $\mathcal{A}$, $\mathcal{B}$, $\sqsubseteq$, $\preceq$, abs and con on the type variables. Extending them to type expressions as detailed in section 4, we have the following theorems.

Theorem 12 (Galois Connections For Free) For all type expressions t, ($\mathcal{A}_t, \sqsubseteq_t$) is a partially ordered set, as is ($\mathcal{B}_t, \preceq_t$); moreover, the pair ($[\mathrm{abs}, \mathrm{con}^{\cup}]_t$, $[\mathrm{con}, \mathrm{abs}^{\cup}]_t$) is a Galois connection between these two posets.

□

Returning to our running example, an instance of theorem 12 is that the pair of functions

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

and

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

is a Galois connection between the posets

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

ordered by

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

and

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

ordered by

$$(/) \leftarrow (/) \leftarrow ((/) \leftarrow (/) \times (/)) \leftarrow \mathtt{List}.(/) \quad .$$

Theorem 13 Define the variable assignment P by

$$P_a \;=\; abs_a^{\cup} \bullet \sqsubseteq_a \quad .$$

Let P_t be its extension to arbitrary type expression t, as detailed above. Then, for all type expressions t,

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

That is, x and y are related by the relation P_t obtained by extending the pair algebras P_a to arbitrary type expressions exactly when x and y are related by the pair algebra defined by the constructed Galois connection.

□

Corollary 14 (Safety For Free) If expression e has type t and is parametrically polymorphic in all the type variables in t then

$$[abs\,,con^{\cup}]_t . e \;\sqsubseteq_t\; e \quad .$$

(The two occurrences of e are, of course, different instances of the given expression. See the discussion of the free theorem above.)

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

$$e \in P_t \quad .$$

The corollary is thus immediate from theorem 13.
□

Applying theorem 12 to our running example, we know that, for all a, given a pair algebra P_a,

$$\mathit{fold} \;\in\; P_{a \leftarrow a \leftarrow (a \leftarrow a \times a) \leftarrow \mathtt{List}.a} \quad .$$

This is true in particular for P_a defined to be $dk^{\cup} \bullet (\Leftarrow)$. Corollary 14 gives the property mentioned earlier, namely

$$dk.(\mathsf{fold}.m.(\times).ms) \;\Leftarrow\; \mathsf{fold}.(dk.m).(\otimes).(\mathsf{List}.dk.ms)$$

where

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

Specifically, the corollary states that

$$(dk \leftarrow kd^{\cup} \leftarrow (kd^{\cup} \leftarrow dk{\times}dk) \leftarrow \mathsf{List}.kd^{\cup}).\mathsf{fold} \;\dot{\Leftarrow}\; \mathsf{fold}$$

where $\dot{\Leftarrow}$ is the pointwise extension of the $\Leftarrow$ ordering on boolean-valued functions. Using the algebraic properties of the arrow operator this point-free formulation of the property of fold instantiates to the above pointwise property in the case that the binary operator supplied as parameter to fold is multiplication (which is, of course, monotonic with respect to divisibility).

6 CSP Traces

This section illustrates how our theorems might be applied to programming language semantics. The (much simplified) example is based on Hoare's CSP language [Hoa85] restricted to just choice and concurrency. Here are two simple processes, written in Hoare's notation:

$$\begin{array}{lcl} \mathit{Customer} & = & \mathit{coin} \rightarrow \mathit{choc} \rightarrow \mathit{Customer} \\ \mathit{Vendor} & = & \mathit{coin} \rightarrow (\mathit{choc} \rightarrow \mathit{Vendor} \mid \mathit{toffee} \rightarrow \mathit{Vendor}) \end{array}$$

Identifiers beginning with a capital letter denote *processes*; the other identifiers denote events. If P is a process and x an event, then $x \rightarrow P$ is a process that first engages in the event x and then behaves like P. The process $(x \rightarrow P \mid y \rightarrow Q)$ 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 $(x : B \rightarrow P(x))$. 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 only happen if both P and Q can simultaneously engage in that event. For example, we might want to run the $\mathit{Customer}$ and Vendor processes concurrently. This would lead to a process that is equivalent to the $\mathit{Customer}$ 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.

$$\textbf{type}\ \mathrm{CSP} = \left\langle \forall\tau,\ \alpha :: \begin{array}{lcl} \mathsf{choice} & : & \tau \leftarrow \mathsf{List}.(\alpha\times\tau) \\ (\,\|\,) & : & \tau \leftarrow \tau\times\tau \end{array} \right\rangle$$

The definition of $\rightarrow$ is:

$$\begin{array}{lcl} (\rightarrow) & :: & \langle \forall\tau,\ \alpha :: \ \tau \leftarrow \tau \leftarrow \alpha \leftarrow \mathrm{CSP}.\tau.\alpha \rangle \\ x \rightarrow_C P & = & \mathsf{choice}_C.[(x, P)] \end{array}$$

Let us now define the $\mathsf{Customer}$ and Vendor processes with our library. First we need to define the type of events:

$$\textbf{type}\ A = \mathsf{coin} \mid \mathsf{choc} \mid \mathsf{toffee}$$

Then the definitions of the processes are:

$$\begin{array}{lcl} \mathsf{Customer} & :: & \langle \forall\tau :: \ \tau \leftarrow \mathrm{CSP}.\tau.A \rangle \\ \mathsf{Customer}_C & = & \mathsf{coin} \rightarrow_C \mathsf{choc} \rightarrow_C \mathsf{Customer}_C \\ \\ \mathsf{Vendor} & :: & \langle \forall\tau :: \ \tau \leftarrow \mathrm{CSP}.\tau.A \rangle \\ \mathsf{Vendor}_C & = & \mathsf{coin} \rightarrow_C (\mathsf{choice}_C\ [(\mathsf{choc}, \mathsf{Vendor}_C), (\mathsf{toffee}, \mathsf{Vendor}_C)]) \end{array}$$

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 will use lists to represent traces. Traces are partially ordered:

(15) $\quad s \leq t \ \equiv\ \langle \exists u :: \ s +\!\!+ u = t \rangle \quad .$

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

$$\textbf{type}\ \mathrm{Proc} = \langle \forall\alpha :: \ \mathcal{P}.(\mathsf{List}.\alpha) \rangle \quad .$$

That is, if events have type α, then processes are sets of traces of α. If the set A of all events is finite, then $\mathit{Proc}.A$ 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 $\mathit{all}.A$, where all is defined as follows:

$$\begin{array}{lcl} \mathit{all} & :: & \langle \forall \alpha \ :: \ \ \mathit{Proc}.\alpha \leftarrow \mathcal{P}.\alpha \rangle \\ \mathit{all}.X & = & \{ s \mid \mathit{elems}.s \subseteq X \} \end{array}$$

(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:

$$\begin{array}{lcl} \mathit{elems} & :: & \langle \forall \alpha \ :: \ \ \mathcal{P}.\alpha \leftarrow \mathit{List}.\alpha \rangle \\ \mathit{elems}.[] & = & \emptyset \\ \mathit{elems}.(x{:}xs) & = & \{x\} \cup \mathit{elems}.xs \end{array}$$

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

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

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

We can use Tr to evaluate some of the processes that we defined earlier. For example, if we evaluate the $\mathit{Customer}$ process, we find that:

$$\mathit{Customer}_{\mathit{Tr}} = \{[]\} \cup \{\mathit{coin}{:}\mathit{choc}{:}s \mid s \in \mathit{Customer}_{\mathit{Tr}}\}$$

In other words, $\mathit{Customer}_{\mathit{Tr}}$ is the set $\{[], [\mathit{coin}, \mathit{choc}], \ldots\}$ as expected.

6.3 Analysing the alphabet

The $\mathit{Customer}$ and Vendor processes are very simple, so it is easy for us to see which events they might engage in. $\mathit{Customer}$ 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:

$$\begin{array}{lcl} \mathit{events} & :: & \langle \forall \alpha \ :: \ \ \mathcal{P}.\alpha \leftarrow \mathit{Proc}.\alpha \rangle \\ \mathit{events}.P & = & \langle \bigcup s : \ \ s \in P \ : \ \ \mathit{elems}.s \rangle \end{array}$$

Notice that events forms a Galois connection with all:

$$\mathit{events}.P \subseteq X \quad \equiv \quad P \subseteq \mathit{all}.X \quad .$$

This means that we can derive a new library Ev that calculates the set of events that a process engages in. From the types of choice and concurrency our theorem predicts that the libraries Tr and Ev should be related as follows:

$$\begin{array}{lcl} \mathsf{choice}_{\mathsf{Ev}} & = & (\mathit{events} \leftarrow \mathsf{List}.(\mathsf{id} \times \mathsf{all}^{\cup})).\mathsf{choice}_{\mathsf{Tr}} \\ (\ \|_{\mathsf{Ev}}\ ) & = & (\mathit{events} \leftarrow \mathsf{all}^{\cup} \times \mathsf{all}^{\cup}).(\ \|_{\mathsf{Tr}}\ ) \end{array}$$

Using the definition of the $\leftarrow$ operator this reduces to:

$$\begin{array}{lcl} \mathsf{choice}_{\mathsf{Ev}} & = & \mathit{events} \bullet \mathsf{choice}_{\mathsf{Tr}} \bullet \mathsf{List}.(\mathsf{id} \times \mathsf{all}) \\ (\ \|_{\mathsf{Ev}}\ ) & = & \mathit{events} \bullet (\ \|_{\mathsf{Tr}}\ ) \bullet (\mathsf{all} \times \mathsf{all}) \end{array}$$

Hence, the definition of Ev is:

$$\mathsf{Ev} \ :: \ \langle \forall \alpha \ :: \ \mathsf{CSP}.(\mathcal{P}.\alpha).\alpha \rangle$$

$$\mathsf{Ev} \ = \ \left[\begin{array}{lcl} \mathsf{choice}.[] & = & \emptyset \\ \mathsf{choice}.((x, P) : cs) & = & \{x\} \cup P \cup \mathsf{choice}.cs \\ & & \\ P \| Q & = & P \cap Q \end{array} \right.$$

Using this library, we evaluate the $\mathsf{Customer}$ process:

$$\mathsf{Customer}_{\mathsf{Ev}} = \{\mathsf{coin}\} \cup \{\mathsf{choc}\} \cup \mathsf{Customer}_{\mathsf{Ev}}$$

In other words, $\mathsf{Customer}_{\mathsf{Ev}} = \{\mathsf{coin}, \mathsf{choc}\}$, as expected. If we apply the library to Vendor, then we find that $\mathsf{Vendor}_{\mathsf{Ev}} = \{\mathsf{coin}, \mathsf{choc}, \mathsf{toffee}\}$, which is also as expected.

References

[Abr90] Samson Abramsky. Abstract interpretation, logical relations, and kan extensions. *J. Logic and Computation*, 1(1):5–41, 1990.

[AJ90] Samson Abramsky and Thomas P. Jensen. A relational approach to strictness analysis for higher-order polymorphic functions. In *???*, 1990.

[Bac90] R.C. Backhouse. On a relation on functions. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business*. Springer-Verlag, 1990.

[BBH+92] R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In M. Nivat, C.S. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST'91*, pages 303–326. Springer-Verlag, Workshops in Computing, 1992.

[BdM96] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall International, 1996.

[BHA86] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. *Science of Computer Programming*, 7:249–278, 1986.

[BVW92] R.C. Backhouse, T.S. Voermans, and J. van der Woude. A relational theory of datatypes. Available via World-Wide Web at `http://www.cs.nott.ac.uk/~rcb/MPC/papers` Available via anonymous ftp from `ftp.win.tue.nl` in directory `pub/math.prog.construction`, December 1992.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unifed lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, January 1979.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hoo97] Paul Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1997.

[HS64] J. Hartmanis and R.E. Stearns. Pair algebras and their application to automata theory. *Information and Control*, 7(4):485–507, 1964.

[HS66] J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, 1966.

[NNH98] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1998.

[Rey83] J.C. Reynolds. Types, abstraction and parametric polymorphism. In R.E. Mason, editor, *IFIP '83*, pages 513–523. Elsevier Science Publishers, 1983.

[Wad89] P. Wadler. Theorems for free! In *4'th Symposium on Functional Programming Languages and Computer Architecture, ACM, London*, September 1989.

7 Appendix

The proofs of theorems 12 and 13 use point-free relation algebra. This has the advantage of reducing the size of the formulae we have to manipulate, making the proofs easier to construct and to verify. On the other hand, point-free relation algebra is not widely known so we need to begin by introducing some basic rules.

The Arrow Operator To express the definition of $R \leftarrow S$ in point-free relation algebra, we view functions f and g as particular kinds of relations. (The characterising "shunting rule" for functions will be given shortly.) We use $^{\cup}$ as a postfix operator to denote the converse operation on relations, we use $R \bullet S$ for the composition of relations R and S, and we use the usual infix operators $\supseteq$ and $\subseteq$ to denote set inclusion. The defining property then becomes:

$$(16) \qquad (f,g) \in R \leftarrow S \quad \equiv \quad R \bullet g \supseteq f \bullet S \quad .$$

(The advantage of this formulation over the point-free definition (1) is that it eliminates the universal quantification as well as the bound variables u and v.)

From this definition (and basic properties of relation algebra) we can easily derive the following algebraic properties of the $\leftarrow$ operator. (See [Bac90] for the proofs.)

$$(17) \qquad (R \leftarrow S)^{\cup} \; = \; R^{\cup} \leftarrow S^{\cup} \quad ,$$

$$(18) \qquad (R \bullet S) \leftarrow (T \bullet U) \; \supseteq \; (R \leftarrow T) \bullet (S \leftarrow U) \quad \text{, and}$$

$$(19) \qquad (R \bullet f) \leftarrow (S \bullet g^{\cup}) \; = \; (R \leftarrow S) \bullet (f \leftarrow g^{\cup}) \quad .$$

We refer to these three properties as the *distributivity* properties of $\leftarrow$. Note carefully that the middle property expresses an inclusion whereas the other two are equalities. The $\leftarrow$ arrow operator is also monotonic in its left argument and anti-monotonic in its right argument. That is

$$(20) \qquad R \leftarrow S \; \supseteq \; T \leftarrow U \quad \Leftarrow \quad R \supseteq T \wedge S \subseteq U \quad .$$

Finally, using id_X to denote the identity relation on set X, we have:

$$(21) \qquad \mathsf{id}_{\mathcal{A}} \leftarrow \mathsf{id}_{\mathcal{B}} \; = \; \mathsf{id}_{\mathcal{A} \leftarrow \mathcal{B}} \quad .$$

Shunting of Functions Functions are very special sorts of relations because they enjoy a number of properties not generally enjoyed by relations proper. (For example, compare the middle and last distributivity properties above.) 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,

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

(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

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

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*

$$f \bullet f^{\cup} \subseteq id_A$$

and *entire*

$$id_B \subseteq f^{\cup} \bullet f$$

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.)

Galois Connections Expressed in point-free relation algebra, the pair algebra corresponding to the Galois connection (abs, con) between $(\mathcal{A}, \sqsubseteq)$ and $(\mathcal{B}, \preceq)$ is the relation $abs^{\cup} \bullet \sqsubseteq$. The fact that the functions are indeed Galois connected is expressed very succinctly by the property

$$abs^{\cup} \bullet \sqsubseteq \quad = \quad \preceq \bullet con \quad .$$

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

$$con \bullet \sqsubseteq \quad \subseteq \quad \preceq \bullet con \quad ,$$

$$abs \bullet \preceq \quad \subseteq \quad \sqsubseteq \bullet abs \quad ,$$

$$\preceq \quad \subseteq \quad \preceq \bullet \mathit{con} \bullet \mathit{abs}$$

and

$$\mathit{abs} \bullet \mathit{con} \bullet \sqsubseteq \quad \subseteq \quad \sqsubseteq \ .$$

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

7.1 Basic Constructions

The following lemma generalises what is known in the literature as the "independent attribute method" [NNH98, page 247]. Specifically, the independent attribute method is the case that the relator F is the product relator. For zero-ary relators the lemma is that, if $(\mathcal{A}, \leq)$ is a poset, the pair $(\mathsf{id}_A, \mathsf{id}_A)$ is a Galois connection between $(\mathcal{A}, \leq)$ and itself. (This important result is undoubtedly well-known but often omitted in the literature.) We state and prove the lemma only for a unary relator. The extension to non-unary relators is easy but requires extra apparatus. (Replace posets, relations and functions everywhere by vectors of posets, relations and functions.)

Lemma 22 Let $(\mathcal{A}, \leq)$ be a posets. Suppose F is a relator that preserves binary intersections. (That is, for all relations R and S, $F.(R\cap S) = F.R \cap F.S$.) Then $(F.\mathcal{A}, F.\leq)$ is a poset.

Proof It is easy to verify that transitivity and reflexivity of $F.\leq$ follows from transitivity and reflexivity of $\leq$. Antisymmetry of $F.\leq$ is the property that

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

This follows from the antisymmetry of $\leq$ $(\leq \cap \leq^{\cup} \subseteq \mathsf{id}_{\mathcal{A}})$ and that F commutes with converse and the assumption that F preserves binary intersections.
□

Lemma 23 Let $(\mathcal{A}, \leq)$ and $(\mathcal{B}, \preceq)$ be posets. Then $\leq \leftarrow \preceq$ is a partial ordering on the monotonic functions of type $\mathcal{A} \leftarrow \mathcal{B}$.

Proof The monotonic functions of type $\mathcal{A} \leftarrow \mathcal{B}$ are exactly the functions for which $\leq \leftarrow \preceq$ is reflexive. Transitivity follows from (18) and antisymmetry from reflexivity of $\leq$ and $\leq \leftarrow \preceq$.
□

Corollary 24 Let t be a type expression and let R assign partial ordering relations to the type variables in t. Let $\mathcal{A}$ be the corresponding poset assignment. (So for each type variable a, R_a is a partial ordering relation on $\mathcal{A}_a$.) Then R_t is a partial ordering on $\mathcal{A}_t$.

□

Lemma 25 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 a total function of type $(A \leftarrow D) \leftarrow (B \leftarrow C)$.

Proof First we prove that $f \leftarrow g^\cup$ is simple.

$$
\begin{array}{ll}
 & (f \leftarrow g^\cup) \bullet (f \leftarrow g^\cup)^\cup \\
= & \quad \{ \quad (17) \quad \} \\
 & (f \leftarrow g^\cup) \bullet (f^\cup \leftarrow g) \\
\subseteq & \quad \{ \quad (18) \quad \} \\
 & (f \bullet f^\cup) \leftarrow (g^\cup \bullet g) \\
\subseteq & \quad \{ \quad f \bullet f^\cup \subseteq id_A \,,\ g^\cup \bullet g \supseteq id_D \,,\ (20) \quad \} \\
 & id_A \leftarrow id_D \\
= & \quad \{ \quad (21) \quad \} \\
 & id_{A \leftarrow D} \quad .
\end{array}
$$

Now we prove that it is entire.

$$
\begin{array}{ll}
 & (f \leftarrow g^\cup)^\cup \bullet (f \leftarrow g^\cup) \\
= & \quad \{ \quad (17) \quad \} \\
 & (f^\cup \leftarrow g) \bullet (f \leftarrow g^\cup) \\
= & \quad \{ \quad (19) \quad \} \\
 & (f^\cup \bullet f) \leftarrow (g \bullet g^\cup) \\
\supseteq & \quad \{ \quad f^\cup \bullet f \supseteq id_B \,,\ g \bullet g^\cup \subseteq id_C \,,\ (20) \quad \} \\
 & id_B \leftarrow id_C \\
= & \quad \{ \quad (21) \quad \} \\
 & id_{B \leftarrow C} \quad .
\end{array}
$$

□

The following three properties will be used frequently; they are all easily proved by induction on the structure of type expressions. For all type expressions t and all assignments R and S of type variables to relations,

(26) $\quad R_t = [R,R]_t \quad ,$

(27) $\quad (R_t)^\cup = (R^\cup)_t \quad$, and

(28) $\quad ([R,S]_t)^\cup = [R^\cup, S^\cup]_t \quad .$

Lemma 29 Suppose f and g are type 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, $[f, g^\cup]_t$ is a total function of type $[A,D]_t \leftarrow [B,C]_t$.

Proof The proof is by induction on the structure of type expressions. First, if t is a type variable a, then $[f, g^\cup]_t = f_a$ which is a total function of type $A_a \leftarrow B_a$ (and hence of type $[A,D]_a \leftarrow [B,C]_a$). Second, consider the case where $t = u \leftarrow v$. Then

$$
\begin{array}{ll}
 & [f, g^\cup]_{u \leftarrow v} \\
= & \{ \quad \text{definition: (7)} \quad \} \\
 & [f, g^\cup]_u \leftarrow [g^\cup, f]_v \\
= & \{ \quad (28) \quad \} \\
 & [f, g^\cup]_u \leftarrow [g, f^\cup]_v^\cup \quad .
\end{array}
$$

But by the induction hypothesis and lemma 25, the above is a total function of type

$$([A,D]_u \leftarrow [D,A]_v) \leftarrow ([B,C]_u \leftarrow [C,B]_v) \quad .$$

That is, it is a total function of type $[A,D]_t \leftarrow [B,C]_t$.

Finally, if t is a relator application then, by definition,

$$[f, g^\cup]_{F.(u\ldots v)} = F.([f, g^\cup]_u \ldots [f, g^\cup]_v) \quad .$$

Relators preserve total functions, so by the induction hypothesis this is a total function. Also, its type is

$$F.([A,D]_u \ldots [A,D]_v) \leftarrow F.([B,C]_u \ldots [B,C]_v)$$

which, because F is a functor is the same as

$$[A,D]_{F.(u\ldots v)} \leftarrow [B,C]_{F.(u\ldots v)}$$

as required.

□

Lemma 30 (Monotonicity) Suppose R, S, T and U are type assignments such that for all type variables a:

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

Then, for all type expressions t,

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

Proof Straightforward induction on the structure of type expressions.
□

Lemma 31 (Factorisation of functions) Suppose R, S, f and g are type assignments such that, for all type variables a, f_a and g_a are total functions and the compositions $R_a \bullet f_a$ and $S_a \bullet g_a^{\cup}$ are well defined. Then, for all type expressions t,

(32) $\quad [R,S]_t \bullet \left[f, g^{\cup}\right]_t \supseteq \left[R\bullet f \,,\, S\bullet g^{\cup}\right]_t$, and

(33) $\quad [S,R]_t \bullet \left[g^{\cup}, f\right]_t \subseteq \left[S\bullet g^{\cup} \,,\, R\bullet f\right]_t$.

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 (6), we have

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

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

Second, consider the case where $t = u \leftarrow v$. For property (32) we reason:

$$
\begin{array}{ll}
 & [R,S]_{u\leftarrow v} \bullet [f, g^{\cup}]_{u\leftarrow v} \\
= & \{ \quad \text{definition: (7)} \quad \} \\
 & ([R,S]_u \leftarrow [S,R]_v) \bullet ([f, g^{\cup}]_u \leftarrow [g^{\cup}, f]_v) \\
= & \{ \quad \text{distributivity: (19), lemma 29 and (28)} \quad \} \\
 & ([R,S]_u \bullet [f, g^{\cup}]_u) \leftarrow ([S,R]_v \bullet [g^{\cup}, f]_v) \\
\supseteq & \{ \quad \text{monotonicity: (20), induction hypotheses: (32) and (33)} \quad \} \\
 & [R\bullet f \,,\, S\bullet g^{\cup}]_u \leftarrow [S\bullet g^{\cup} \,,\, R\bullet f]_v \\
= & \{ \quad \text{definition: (7)} \quad \} \\
 & [R\bullet f \,,\, S\bullet g^{\cup}]_{u\leftarrow v} \quad .
\end{array}
$$

For property (33) we reason:

$$
\begin{array}{ll}
 & [S,R]_{u\leftarrow v} \bullet [g^\cup , f]_{u\leftarrow v} \\
= & \qquad \{ \qquad \text{definition: (7)} \quad \} \\
 & ([S,R]_u \leftarrow [R,S]_v) \bullet ([g^\cup , f]_u \leftarrow [f , g^\cup]_v) \\
\subseteq & \qquad \{ \qquad \text{distributivity: (18)} \quad \} \\
 & ([S,R]_u \bullet [g^\cup , f]_u) \leftarrow ([R,S]_v \bullet [f , g^\cup]_v) \\
\subseteq & \qquad \{ \qquad \text{monotonicity: (20), induction hypotheses} \quad \} \\
 & [S \bullet g^\cup \ , \ R \bullet f]_u \leftarrow [R \bullet f \ , \ S \bullet g^\cup]_v \\
= & \qquad \{ \qquad \text{definition: (7)} \quad \} \\
 & [S \bullet g^\cup \ , \ R \bullet f]_{u\leftarrow v} \quad .
\end{array}
$$

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

7.2 The Theorems

Lemma 34 Suppose that 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,

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

Proof

$$
\begin{array}{ll}
 & ([f , g^\cup]_t \ , \ [h , k^\cup]_t) \in [R,U]_t \leftarrow [S,T]_t \\
= & \qquad \{ \qquad \text{definition of } \leftarrow \text{ (16)} \quad \} \\
 & [R,U]_t \bullet [h , k^\cup]_t \ \supseteq \ [f , g^\cup]_t \bullet [S,T]_t \\
\Leftarrow & \qquad \{ \qquad \text{(32), (33) together with the fact that} \\
 & \qquad \qquad \text{converse is a poset isomorphism, transitivity of } \supseteq \quad \} \\
 & [R \bullet h \ , \ U \bullet k^\cup]_t \ \supseteq \ [f \bullet S \ , \ g^\cup \bullet T]_t \\
\Leftarrow & \qquad \{ \qquad \text{monotonicity: (30)} \quad \} \\
 & \langle \forall a \ :: \ (R \bullet h)_a \ \supseteq \ (f \bullet S)_a \ \wedge \ (U \bullet k^\cup)_a \subseteq (g^\cup \bullet T)_a \rangle \\
= & \qquad \{ \qquad \text{definition: (3)} \quad \} \\
 & \langle \forall a \ :: \ R_a \bullet h_a \ \supseteq \ f_a \bullet S_a \ \wedge \ U_a \bullet k_a^\cup \subseteq g_a^\cup \bullet T_a \rangle \\
= & \qquad \{ \qquad \text{shunting of functions and definition of } \leftarrow \text{ (16)} \quad \} \\
 & \langle \forall a \ :: \ (f_a,h_a) \in R_a \leftarrow S_a \ \wedge \ (g_a,k_a) \in T_a \leftarrow U_a \rangle \quad .
\end{array}
$$

□

Corollary 36 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,

(37) $\left[f, g^{\cup}\right]_t \in R_t \leftarrow S_t$, and

(38) $\left[g, f^{\cup}\right]_t \in S_t \leftarrow R_t$.

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 lemma 34 and use (26).
□

Lemma 39 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, the following cancellation property holds:

$$g_a \bullet R_a \subseteq f_a^{\cup} \bullet R_a \ .$$

Then, for all type expressions t,

$$[g, f^{\cup}]_t \bullet R_t \subseteq (f^{\cup} \bullet R)_t \subseteq [f^{\cup}, g]_t \bullet R_t \ .$$

Proof

$$
\begin{array}{ll}
 & [g, f^{\cup}]_t \bullet R_t \subseteq (f^{\cup} \bullet R)_t \subseteq [f^{\cup}, g]_t \bullet R_t \\
\Leftarrow & \{ \quad \text{(26), (32), (33) together with the fact that} \\
 & \quad\quad \text{converse is a poset isomorphism, transitivity of } \supseteq \quad \} \\
 & [g \bullet R, f^{\cup} \bullet R]_t \subseteq [f^{\cup} \bullet R, f^{\cup} \bullet R]_t \subseteq [f^{\cup} \bullet R, g \bullet R]_t \\
\Leftarrow & \{ \quad \text{monotonicity: (30)} \quad \} \\
 & \langle \forall a :: g_a \bullet R_a \subseteq f_a^{\cup} \bullet R_a \subseteq f_a^{\cup} \bullet R_a \ \wedge \ f_a^{\cup} \bullet R_a \supseteq f_a^{\cup} \bullet R_a \supseteq g_a \bullet R_a \rangle \\
= & \{ \quad \text{reflexivity of } \subseteq \text{ and assumption: } g_a \bullet R_a \subseteq f_a^{\cup} \bullet R_a \quad \} \\
 & \mathit{true} \ .
\end{array}
$$

□

Lemma 40 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 α, the following cancellation property holds:

$$S_\alpha \bullet f_\alpha^\cup \subseteq S_\alpha \bullet g_\alpha \quad .$$

Then, for all type expressions t,

$$S_t \bullet [f^\cup, g]_t \subseteq (S \bullet g)_t \subseteq S_t \bullet [g, f^\cup]_t \quad .$$

Proof Similar to the proof of theorem 39.
□

Theorem 41 (Galois Connections For Free) Suppose that f and g assign total functions to type variables, and R and S assign partial ordering relations to type variables. Suppose further that, for all type variables α, the pair (f_α, g_α) is a Galois connection between the orderings R_α and S_α. Then, for all type expressions t, the pair $([f, g^\cup]_t\ ,\ [g, f^\cup]_t)$ is a Galois connection between the orderings R_t and S_t .

Proof First, by lemma 29 $[f, g^\cup]_t$ and $[g, f^\cup]_t$ are both total functions. Second, by lemma 24, R_t and S_t are partial orderings. Third, by corollary 36 $[f, g^\cup]_t$ and $[g, f^\cup]_t$ are monotonic functions between these orderings. Finally, by theorems 39 and 40 (and shunting of functions) they satisfy the required cancellation properties.
□

Theorem 42 Suppose that f and g assign total functions to type variables, and R and S assign partial ordering relations to type variables. Suppose further that, for all type variables α, the pair (f_α, g_α) is a Galois connection between the orderings R_α and S_α. Define the relation assignment P by $P_\alpha = (f^\cup \bullet R)_\alpha$. Then, for all type expressions t,

$$P_t = [f^\cup, g]_t \bullet R_t \quad .$$

Proof Assume that f_α has type $A_\alpha \leftarrow B_\alpha$ and g_α has type $C_\alpha \leftarrow D_\alpha$. From lemma 39 we know that

$$(f^\cup \bullet R)_t \subseteq [f^\cup, g]_t \bullet R_t \quad .$$

That is,

$$P_t \subseteq [f^\cup, g]_t \bullet R_t \quad .$$

So we have to prove the opposite inclusion. The proof is by induction on the structure of type expressions. For type variable α, by instantiating the definitions (3), (2) and (6), we have

$$P_a \;=\; (f^{\cup} \bullet R)_a \;=\; f^{\cup}_a \bullet R_a \;=\; [f^{\cup}, g]_a \bullet R_a \quad .$$

This establishes the basis of the proof. For the inductive step $t = u \leftarrow v$ we have:

$$
\begin{array}{ll}
 & [f^{\cup}, g]_t \bullet R_t \\
= & \{ \quad t = u \leftarrow v \text{, definitions (7) and (4)} \quad \} \\
 & ([f^{\cup}, g]_u \leftarrow [g, f^{\cup}]_v) \bullet (R_u \leftarrow R_v) \\
\subseteq & \{ \quad S_u \leftarrow S_v \text{ is reflexive (corollary 24)} \quad \} \\
 & (S_u \leftarrow S_v) \bullet ([f^{\cup}, g]_u \leftarrow [g, f^{\cup}]_v) \bullet (R_u \leftarrow R_v) \\
\subseteq & \{ \quad \text{distributivity: (18)} \quad \} \\
 & S_u \bullet [f^{\cup}, g]_u \bullet R_u \;\leftarrow\; S_v \bullet [g, f^{\cup}]_v \bullet R_v \\
\subseteq & \{ \quad \begin{array}{ll}
 & S_v \bullet [g, f^{\cup}]_v \bullet R_v \\
\supseteq & \{ \quad \text{lemma 29: } \mathrm{id}_{[A,D]_v} \supseteq [f, g^{\cup}]_v \bullet [f^{\cup}, g]_v \quad \} \\
 & S_v \bullet [g, f^{\cup}]_v \bullet [f, g^{\cup}]_v \bullet [f^{\cup}, g]_v \bullet R_v \\
\supseteq & \{ \quad \text{lemma 40} \quad \} \\
 & S_v \bullet [f^{\cup}, g]_v \bullet R_v \\
\supseteq & \{ \quad S_v \text{ is reflexive} \quad \} \\
 & [f^{\cup}, g]_v \bullet R_v \quad .
\end{array} \\
 & \quad \text{antimononoticity (20)} \quad \} \\
 & S_u \bullet [f^{\cup}, g]_u \bullet R_u \;\leftarrow\; [f^{\cup}, g]_v \bullet R_v \\
\subseteq & \{ \quad \begin{array}{ll}
 & S_u \bullet [f^{\cup}, g]_u \bullet R_u \\
\subseteq & \{ \quad [f^{\cup}, g]_u \text{ is monotonic (lemma 36)} \quad \} \\
 & [f^{\cup}, g]_u \bullet R_u \bullet [f, g^{\cup}]_u \bullet [f^{\cup}, g]_u \bullet R_u \\
\subseteq & \{ \quad \text{lemma 29: } [f, g^{\cup}]_u \bullet [f^{\cup}, g]_u \subseteq \mathrm{id}_{[A,D]_u} \quad \} \\
 & [f^{\cup}, g]_u \bullet R_u \bullet R_u \\
\subseteq & \{ \quad R_u \text{ is transitive (corollary 24)} \quad \} \\
 & [f^{\cup}, g]_u \bullet R_u \quad \}
\end{array} \\
 & [f^{\cup}, g]_u \bullet R_u \;\leftarrow\; [f^{\cup}, g]_v \bullet R_v \\
= & \{ \quad \text{induction hypothesis} \quad \} \\
 & P_u \leftarrow P_v \\
= & \{ \quad t = u \leftarrow v \text{, definition (7)} \quad \}
\end{array}
$$

P_t .

Finally, the case that $t = F.(u \ldots v)$, where F is a relator, is verified as follows:

$$
\begin{array}{ll}
 & [f^{\cup}, g]_t \bullet R_t \\
= & \{ \quad t = F.(u \ldots v) \text{ , definition } \quad \} \\
 & F.([f^{\cup}, g]_u \ldots [f^{\cup}, g]_v) \bullet F.(R_u \ldots R_v) \\
= & \{ \quad \text{relators distribute through composition} \quad \} \\
 & F.(([f^{\cup}, g]_u \bullet R_u) \ldots ([f^{\cup}, g]_v \bullet R_v) \\
= & \{ \quad \text{induction hypothesis} \quad \} \\
 & F.(P_u \ldots P_v) \\
= & \{ \quad t = F.(u \ldots v) \text{ , definition 5} \quad \} \\
 & P_t \quad .
\end{array}
$$

□