Making functionality more general

Graham Hutton, University of Glasgow
Ed Voermans, Eindhoven University of Technology*

March 23, 1992

Abstract

The notion of functionality is not cast in stone, but depends upon what we have as types in our language. With *partial equivalence relations* (pers) as types we show that the functional relations are precisely those satisfying the simple equation $f = f \circ f^{\cup} \circ f$, where "$\cup$" is the relation converse operator. This article forms part of "A calculational theory of pers as types" [1].

1 Introduction

In calculational programming, programs are derived from specifications by a process of algebraic manipulation. Perhaps the best known calculational paradigm is the Bird–Meertens formalism, or to use its more colloquial name, Squiggol [2]. Programs in the Squiggol style work upon trees, lists, bags and sets, the so–called Boom hierarchy. The framework was uniformly extended to cover arbitrary recursive types by Malcolm in [3], by means of the F–algebra paradigm of type definition, and resulting catamorphic programming style. More recently, Backhouse et al [4] have made a further generalisation by choosing to work within the relational calculus.

Of fundamental importance to Backhouse's relational approach is the treatment of types not as separate entities from programs, but as special kinds of programs. Specifically, partial identity relations, called monotypes in [4], are adopted as types. Working in this style allows programs and type information to be manipulated withing the single framework of the relational calculus, and is fundamental to the use of type information to "inform and structure" program calculation. We in this paper take a more general approach than Backhouse, adopting partial equivalence relations as types. The extra generality allows us to handle types whose constructors are required to satisfy equations, so–called "types with laws". The present article forms part of [1], to which the reader is referred for omitted proofs.

In the relational style, both specifications and implementations are viewed as binary relations. Thinking of which binary relations should be accepted as implementations, functional (or deterministic) relations naturally come to mind. With

*Authors e–mail addresses: graham@dcs.glasgow.ac.uk, wsinedv@win.tue.nl.

pers as types however, we have a more general notion than normal of what constitutes an element of a type. It is natural then to expect a more general notion of what constitutes a function. We find that functional relations in the "pers as types" world are precisely those satisfying the simple equation $f = f \circ f^{\cup} \circ f$, where "$\cup$" is the relation converse operator. Such relations f are known as *difunctionals*.

1.1 Notation

Our per–based paradigm is developed within the axiomatic theory of relations as presented in [4]. The axiom system comprises three layers, respectively dealing with the *powerset lattice* structure of relations, the *composition* operator, and the *converse* operator. Collectively, the three axiomatic layers are known as the *spec calculus*. Since most readers will already have some knowledge of relational calculus, and there are only a few calculations in this article, we don't mention anything further about the axiom system. (See [4, 1] for more details.)

Rather than working with the normal "$\subseteq$", "$\cup$", and "$\cap$" operators from set–theory, Backhouse adopts in his axiomatic framework the more anonymous lattice–theoretic symbols "$\sqsubseteq$", "$\sqcup$" and "$\sqcap$". The term *spec* (abbreviating specification) is used by Backhouse in preference to *relation*, keeping a clear distinction between the axiomatic theory and the meta–language (predicate calculus.) Throughout this article capital letters $R, S, T, \ldots$ are used to denote arbitrary specs.

To avoid confusion with the use of the capital letter "T" to denote an arbitrary spec, the supreme spec (with respect to the implicit universe) is denoted by "$\top\!\top$" rather than the more usual lattice theoretic symbol "$\top$". The empty–spec is written as "$\bot\!\bot$". The composition operator for specs is denoted by "$\circ$", and the identity spec with respect to this operator by "I". For relation converse (an operator which swaps the components of each pair in a binary relation) the anonymous notation $R^{\cup}$ is used rather than the more usual R^{-1}; writing R^{-1} might have led us to suspect that R^{-1} is the inverse of R with respect to the composition operator "$\circ$", which is not true in general. For reference we give below pointwise definitions for the operators as used in this article. For R a binary relation, $x\ R\ y$ may be read as "x is related by R to y." Formally then, $x\ R\ y$ is just an abbreviation for $(x, y) \in R$.

$$
\begin{array}{lcl}
x \mathrel{\bot\!\bot} y & \widehat{=} & \mathit{false} \\
x \mathrel{\top\!\top} y & \widehat{=} & \mathit{true} \\
R \sqsupseteq S & \widehat{=} & \forall\, (x, y :: x\ R\ y \ \Leftarrow\ x\ S\ y) \\
x\ (R \sqcup S)\ y & \widehat{=} & x\ R\ y \ \vee\ x\ S\ y \\
x\ (R \sqcap S)\ y & \widehat{=} & x\ R\ y \ \wedge\ x\ S\ y \\
x\ (\neg R)\ y & \widehat{=} & \sim (x\ R\ y) \\
x\ (R \circ S)\ z & \widehat{=} & \exists\, (y :: x\ R\ y \ \wedge\ y\ S\ z) \\
x\ I\ y & \widehat{=} & x = y \\
x\ (R^{\cup})\ y & \widehat{=} & y\ R\ x
\end{array}
$$

1.2 Quantification

For quantified expressions, we adopt the *Eindhoven notation* [5]. The general pattern is $Q\ (x : p.x : t.x)$, where Q is some quantifier (e.g. "$\forall$" or "$\bigsqcup$"), x is a sequence of free variables (sometimes called *dummies*), $p.x$ is a predicate which must be satisfied by the dummies, and $t.x$ is an expression defined in terms of the dummies. For the special cases of "$\forall$" and "$\exists$", we have the following links with standard notation:

$$\forall\ (x : p.x : q.x) \quad \equiv \quad \forall x.\ (p.x \Rightarrow q.x)$$

$$\exists\ (x : p.x : q.x) \quad \equiv \quad \exists x.\ (p.x \wedge q.x)$$

1.3 Monotypes and domain operators

In the next section, we introduce our notion of types: "partial equivalence relations". Backhouse in [4] works with less general types: "partial identity relations". Following Backhouse, we refer to partial identity relations as *monotypes*.

Definition 1: spec A is a *monotype* $\ \widehat{=}\ \ I \sqsupseteq A$

To aid reading of formulae, monotypes are always denoted by letters A, B, C, etc. Set–theoretically, a monotype is just the identity relation on some set. For example, $\{(\mathsf{false}, \mathsf{false}), (\mathsf{true}, \mathsf{true})\}$ is a monotype representing a type of booleans.

We often need to refer to the domain and range of a spec. In order to avoid confusion with decision to have the input part of functional relations on their right–side (explained later on in this article), we use the terms *left domain* and *right domain*. In the case of functional relations then, the right domain is the input part. In the spec calculus, a left domain of a spec R is represented by a monotype A satisfying $A \circ R = R$; similarly a right domain of R is represented by a monotype B satisfying $R \circ B = R$. Specs in general have many domains; the smallest left and right domains are given by operators "$<$" and "$>$", defined as follows:

Definition 2:

$$R{<} \ \widehat{=}\ \ I \sqcap (R \circ R{\cup})$$

$$R{>} \ \widehat{=}\ \ I \sqcap (R{\cup} \circ R)$$

Using "$<$" and "$>$" we can define what it means for two specs to be disjoint:

Definition 3: $\mathit{disj}.R.S \ \widehat{=}\ R{<} \sqcap S{<} \ = \ R{>} \sqcap S{>} \ = \ \bot\!\!\bot$

2 Partial equivalence relations

A *partial equivalence relation* (per) on a set $\mathcal{S}$ is a symmetric and transitive relation on $\mathcal{S}$. To aid reading of formulae, we always use letters A, B and C for pers. In the spec–calculus, we have three equivalent definitions for the *per* notion.

Definition 4: $per.A \;\widehat{=}$

(a) $A = A\cup \;\wedge\; A \sqsupseteq A \circ A$

(b) $A = A\cup \;\wedge\; A = A \circ A$

(c) $A = A \circ A\cup$

Consider for example definition 4a. The calculation below shows that the $A \sqsupseteq A \circ A$ part corresponds to the set–theoretic notion of transitivity; a similar calculation would show that the $A = A\cup$ part corresponds to the symmetry condition.

$$
\begin{array}{ll}
 & A \sqsupseteq A \circ A \\
\equiv & \quad \{\ \text{def} \sqsupseteq\ \} \\
 & \forall\,(x, z ::\ x\ A\ z \Leftarrow x\ A{\circ}A\ z) \\
\equiv & \quad \{\ \text{def} \circ\ \} \\
 & \forall\,(x, z ::\ x\ A\ z \Leftarrow \exists\,(y :: x\ A\ y \wedge y\ A\ z)) \\
\equiv & \quad \{\ \exists \text{ elimination}\ \} \\
 & \forall\,(x, y, z ::\ x\ A\ z \Leftarrow x\ A\ y \wedge y\ A\ z)
\end{array}
$$

A per on a set $\mathcal{S}$ that is reflexive on $\mathcal{S}$ is called an *equivalence relation* on $\mathcal{S}$. We use the notation $\mathcal{S}^2$ for the *full relation* $\mathcal{S} \times \mathcal{S}$ on $\mathcal{S}$. Formally then we have $x\ \mathcal{S}^2\ y \;\widehat{=}\; x, y \in S$. For example, $\{a, b\}^2 = \{(a, a), (a, b), (b, a), (b, b)\}$. It is clear that $\mathcal{S}^2$ is an equivalence relation on $\mathcal{S}$, indeed it is the largest such. The set–theoretic notion of a full relation can be cast in the spec calculus as follows:

Definition 5: spec X is a *full relation* $\;\widehat{=}\; X = X \circ \top\!\top \circ X\cup$.

We always use letters X, Y and Z for full relations. A well known result is:

Lemma 6: every per can be written in a unique way as the union of disjoint full relations, each such relation representing an *equivalence class* of the per.

With "pers as types", it is these disjoint full relations that we refer to as *elements* of types. Aiming towards defining an "$\in$" operator for pers, we see that two properties of a set $\mathcal{S}$ are required such that $\mathcal{S}^2$ may be called an element of a per A:

(1) $\forall\,(x, y\ :\ x, y \in \mathcal{S}\ :\ x\ A\ y)$

(2) $\forall\,(x, y :\ x \in \mathcal{S}\ \wedge\ x\ A\ y\ :\ y \in \mathcal{S})$

The first clause says that $\mathcal{S}$ is an equivalence class of A, being just the set–theoretic expansion of $\mathcal{S}^2 \sqsubseteq A$. The second clause ensures that $\mathcal{S}$ is as big as possible. The following calculation shows that (2) is equivalent to $\mathcal{S}^2 \circ A \sqsubseteq \mathcal{S}^2$.

$$
\begin{array}{ll}
 & \mathcal{S}^2 \circ A \sqsubseteq \mathcal{S}^2 \\
\equiv & \quad \{ \text{ def } \sqsubseteq \ \} \\
 & \forall\, (x, z :\ z\ \mathcal{S}^2 \circ A\ x :\ z\ \mathcal{S}^2\ x) \\
\equiv & \quad \{ \text{ def } \circ \ \} \\
 & \forall\, (x, y, z :\ z\ \mathcal{S}^2\ y \wedge y\ A\ x :\ z\ \mathcal{S}^2\ x) \\
\equiv & \quad \{ \text{ def } \mathcal{S}^2 \ \} \\
 & \forall\, (x, y, z :\ y, z \in \mathcal{S} \wedge y\ A\ x :\ x, z \in \mathcal{S}) \\
\equiv & \quad \{ \text{ calculus } \} \\
 & \forall\, (x, y, z :\ y, z \in \mathcal{S} \wedge y\ A\ x :\ x \in \mathcal{S}) \\
\equiv & \quad \{ \ \forall\, (x, y :\ y \in \mathcal{S} \wedge y\ A\ x :\ \exists\, (z\ ::\ z \in \mathcal{S})) \ \} \\
 & \forall\, (x, y :\ y \in \mathcal{S} \wedge y\ A\ x :\ x \in \mathcal{S})
\end{array}
$$

Given $\mathcal{S}^2 \sqsubseteq A$ we find that $\mathcal{S}^2 \circ A \sqsubseteq \mathcal{S}^2$ is equivalent to the equality $\mathcal{S}^2 \circ A = \mathcal{S}^2$. Since the equality implies the containment, the following proof is sufficient:

$$
\begin{array}{ll}
 & \mathcal{S}^2 \circ A \\
\sqsupseteq & \quad \{ \text{ assumption: } A \sqsupseteq \mathcal{S}^2 \ \} \\
 & \mathcal{S}^2 \circ \mathcal{S}^2 \\
= & \quad \{ \text{ full relations } \} \\
 & \mathcal{S}^2
\end{array}
$$

We give now a spec–calculi definition for an "$\in$" operator:

Definition 7: For X a full relation and A a per,

$$X \in A \ \ \widehat{=} \ \ X \sqsubseteq A \ \wedge \ X \circ A = X$$

To clarify our interpretation of pers as types, let us consider an example. A *rational number* may be represented as a pair of integers (a, b), with $b \neq 0$. The interpretation is that (a, b) represents the rational a/b. This representation is not unique: pairs (a, b) and (c, d) represent the same rational iff $ad = bc$. We see then that rational numbers induce a per, RAT say, on the set $\mathbb{Z} \times \mathbb{Z}$:

$$(a, b)\ RAT\ (c, d) \ \ \widehat{=} \ \ ad = bc \ \wedge \ bd \neq 0$$

Partiality arises in the denominator of a rational being restricted to non–zero integers; equivalence classes arise in the representation as pairs of integers being non–unique. Viewing the per RAT as a type, it is important to understand that elements of type RAT are not simply pairs of integers, but full relations on equivalence classes of pairs of integers.

3 Partial orderings on types

Consider the types of (non–empty) *trees*, *lists* and *uplists*. (We use the term *uplist* for a list whose elements are ascending with respect to some partial ordering.) Elements of all three types can be built from constructors $[-]$ (making singleton elements) and "$+\!\!+$" (joining two elements together). The three types are different however in the properties required of the constructors. We introduce in this section three orderings on types; these orderings allow us to make precise such relationships.

When used as a constructor for trees, "$+\!\!+$" is not subject to any laws. When used with lists however, we require that "$+\!\!+$" be associative. For example, we require that $[1] +\!\!+ ([2] +\!\!+ [3]) = ([1] +\!\!+ [2]) +\!\!+ [3]$. More formally, while the two expressions form distinct (singleton) equivalence classes in the type of trees, they are the members of the same equivalence class in the type of lists. In standard list notation, this equivalence class is denoted by $[1, 2, 3]$. With uplists the "$+\!\!+$" constructor becomes partial, in that certain constructions are forbidden. In particular, $X +\!\!+ Y$ is only defined if the last element of X is at most the first element of Y. For example, although $[1, 3]$ and $[2, 4]$ are both uplists, their combination, $[1, 3, 2, 4]$ is not. More formally, $[1, 3, 2, 4]$ does not represent an equivalence class in the type of uplists.

We introduce now three orderings on types: "$\subseteq$", "$\lhd|$", and "$\lhd$". Writing *uplist* $\subseteq$ *list* would express that uplists are a special kind of list: every uplist is a list, but the reverse in general does not hold. Writing *list* $\lhd|$ *tree* would say that every list is an equivalence class of trees. Writing *uplist* $\lhd$ *tree* would combine the previous two statements: uplists are made from trees by coalescing and discarding equivalence classes. Each of the three operators is first defined using the "$\in$" operator, after which an equivalent but simpler formulation is given.

Definition 8: Given pers A and B,

$$A \subseteq B \;\widehat{=}\; \forall\, (X : X \in A : X \in B)$$

We read $A \subseteq B$ as "A is a subtype of B", i.e. the type A is formed by discarding elements of type B. We observe now that "$\subseteq$" is a partial ordering on pers, with $\perp\!\!\!\perp$ as a least element. (Pers in fact form a "$\sqcap$" semi–lattice under "$\subseteq$".)

Lemma 9: $A \subseteq B \;\equiv\; A \circ B = A \;\wedge\; A \sqsubseteq B$

Given a full relation X and a per A, we say that X *coalesces* A if all the elements of A are combined to form the single element of X. For example, $\{a, b, c\}^2$ coalesces $\{a, b\}^2 \cup \{c\}^2$. In the spec calculus, coalescence can be defined as follows:

Definition 10: For X a full relation and A a per,

$$X \textit{ coalesces } A \;\widehat{=}\; X = A \circ \top\!\!\top \circ A$$

This notion is used in the definition of the "$\lhd$" ordering:

Definition 11: Given pers A and B,

$$A \lhd B \;\widehat{=}\; \forall\,(X : X \in A : \exists\,(C : C \subseteq B : X \textit{ coalesces } C))$$

We read $A \lhd B$ as "A is a per on B" — the type A is formed by discarding and coalescing elements of type B. We note that "$\lhd$" is a partial ordering on pers, with "$\perp\!\!\!\perp$" as least element, and "I" as greatest element. (Pers in fact form a lattice under "$\lhd$".) We work in practice with a simpler formulation:

Lemma 12: $A \lhd B \;\equiv\; A \circ B = A$

From the symmetry of A and B (i.e. $A = A^{\cup}$ and $B = B^{\cup}$), it follows that $A \lhd B \;\equiv\; B \circ A = A$. We note also that "$\subseteq$" can be expressed in terms of "$\lhd$":

Lemma 13: $A \subseteq B \;\equiv\; A \lhd B \;\wedge\; A \sqsubseteq B$

Writing $A \lhd B$ expresses that A is a partial equivalence relation on B. If A is an equivalence relation on B (i.e. no elements have been discarded in moving to A from B) we write $A \lhd| B$. It can be shown that "$\lhd|$" is a partial ordering on types, with $\top\!\!\top$ as greatest element. (Pers in fact form a "$\sqcup$" semi–lattice under "$\lhd|$".)

Definition 14: Given pers A and B,

$$A \lhd| B \;\widehat{=}\; A \lhd B \;\wedge\; \forall\,(X : X \in B : \exists\,(Y : Y \in A : Y \sqsupseteq X))$$

Lemma 15: $A \lhd| B \;\equiv\; A \lhd B \wedge A \sqsupseteq B$

4 Typing judgements for specs

In set–theoretic approaches to relational calculus, it is common to write $R \subseteq A \times B$ (where A and B are sets) in the form of a *typing judgement* $R \in A \sim B$. In this section we seek to define the set $A \sim B$ in our approach to relational calculus, with A and B being pers. In writing $R \in A \sim B$, we require that R on its left–side respects the elements of A, and on its right–side respects the elements of B. Let us consider the A part in more detail. Recall that elements of pers are full relations. By a relation R "respecting on its left–side" a full relation $\mathcal{S}^2$, we mean that the image through R of each member of $\mathcal{S}$ is the same set; that is, we require that

$$\forall\,(a, b, c : a, b \in \mathcal{S} : a\ R\ c \;\equiv\; b\ R\ c)$$

(That such an image set may be empty means that R is not required to be total on A.) We call such a full relation $\mathcal{S}^2$ a *left equivalence class* of R. We find that the predicate above can be written in relational calculus as $R \sqsupseteq \mathcal{S}^2 \circ R$:

$$
\begin{array}{ll}
 & \mathcal{S}^2 \circ R \sqsubseteq R \\
\equiv & \quad \{\ \text{def} \sqsubseteq\ \} \\
 & \forall\,(a, b\ :\ a\ \mathcal{S}^2 \circ R\ b\ :\ a\ R\ b) \\
\equiv & \quad \{\ \text{def} \circ\ \} \\
 & \forall\,(a, b\ :\ \exists\,(c :: a\ \mathcal{S}^2\ c \wedge c\ R\ b)\ :\ a\ R\ b) \\
\equiv & \quad \{\ \exists \text{ elimination}\ \} \\
 & \forall\,(a, b, c\ :\ a\ \mathcal{S}^2\ c \wedge c\ R\ b\ :\ a\ R\ b) \\
\equiv & \quad \{\ \text{def}\ \mathcal{S}^2\ \} \\
 & \forall\,(a, b, c\ :\ a, c \in S \wedge c\ R\ b\ :\ a\ R\ b) \\
\equiv & \quad \{\ \text{shunting}\ \} \\
 & \forall\,(a, b, c\ :\ a, c \in S :\ c\ R\ b \Rightarrow a\ R\ b) \\
\equiv & \quad \{\ \text{predicate calculus}\ \} \\
 & \forall\,(a, b, c\ :\ a, c \in S :\ a\ R\ b \equiv c\ R\ b)
\end{array}
$$

Similarly we can define a *right equivalence class* of a relation R as a full relation $\mathcal{S}^2$ for which $R \sqsupseteq R \circ \mathcal{S}^2$. We make now two definitions in the spec calculus:

Definition 16:

A *left equivalence class* of a spec R is a full relation X for which $R \sqsupseteq X \circ R$;

A *right equivalence class* of R is such an X for which $R \sqsupseteq R \circ X$.

Using these notions we can define the "~" operator:

Definition 17: For A and B pers, $R \in A \sim B \ \widehat{=}$

$$
\begin{array}{l}
\forall\,(X\ :\ X \in A\ :\ R \sqsupseteq X \circ R)\ \wedge\ A \circ R \sqsupseteq R\ \wedge \\
\forall\,(X\ :\ X \in B\ :\ R \sqsupseteq R \circ X)\ \wedge\ R \circ B \sqsupseteq R
\end{array}
$$

We call such a per A a *left domain* of R, and B a *right domain*. The clauses $A \circ R \sqsupseteq R$ and $R \circ B \sqsupseteq R$ above, equivalent respectively to $A \sqsupseteq R^{<}$ and $B \sqsupseteq R^{>}$, are needed to ensure that pers A and B are big enough to serve as domains of R. In practice we work with a simpler but equivalent formulation of "~":

Lemma 18: $R \in A \sim B \ \equiv\ A \circ R = R = R \circ B$

The right–side of Lemma 18 can be written in fact as a single equality:

Lemma 19: $R \in A \sim B \ \equiv\ A \circ R \circ B = R$

5 Typing judgements for difunctionals

We view relational programming as a generalisation of functional programming: relations for us are the primitive notion, functions are regarded as special kinds of relation. In the last section we introduced a typing notation $R \in A\sim B$ for specs. In this section we introduce a typing notation $f \in A \leftarrow B$ for functional specs, and show that the functional specs are precisely those f satisfying $f = f \circ f\cup \circ f$.

Normally the "input part" of a functional relation is on its left–side. Following [4] however, we view the right–side of a functional relation as its input part, writing $A \leftarrow B$ for the set of all functions to A from B. This choice avoids confusion between written and diagrammatic order for composition of functions, and is consistent with placing the argument to the right of the function symbol during application.

> **Definition 20:** Give a spec R and a full relation X, the *image* of X back through R is given by the full relation $R.X \;\widehat{=}\; R \circ X \circ R\cup$

In writing $f \in A \leftarrow B$, we require two properties: f has a valid typing $A\sim B$ as a spec (i.e. typed functions are a special case of typed specs); applying f to an element of per B gives an element of per A (i.e. f is functional to A from B.)

> **Definition 21:**
>
> $$f \in A \leftarrow B \;\widehat{=}\; f \in A\sim B \;\wedge\; \forall\,(X \;:\; X \in B \;:\; f.X \in A)$$

In practice as usual we work with a simpler but equivalent formulation:

> **Lemma 22:** $f \in A \leftarrow B \;\equiv\; f \in A\sim B \;\wedge\; A \sqsupseteq f \circ f\cup$

In the monotypes world [4], functional specs are characterised as those f satisfying $I \sqsupseteq f \circ f\cup$. It is shown below how this definition corresponds to the normal set–theoretic definition of a relation being functional:

$$\begin{array}{ll} & I \sqsupseteq f \circ f\cup \\ \equiv & \quad \{\ \text{def} \sqsupseteq\ \} \\ & \forall\,(x,y ::\ x\ I\ y \Leftarrow x\ f \circ f\cup\ y) \\ \equiv & \quad \{\ \text{def}\ I, \circ\ \} \\ & \forall\,(x,y ::\ x = y \Leftarrow \exists\,(z ::\ x\ f\ z \wedge z\ f\cup\ y)) \\ \equiv & \quad \{\ \text{def} \cup\ \} \\ & \forall\,(x,y ::\ x = y \Leftarrow \exists\,(z ::\ x\ f\ z \wedge y\ f\ z)) \\ \equiv & \quad \{\ \exists\ \text{elimination}\ \} \\ & \forall\,(x,y,z ::\ x = y \;\Leftarrow\; x\ f\ z \wedge y\ f\ z) \end{array}$$

We see then that $I \sqsupseteq f \circ f\cup$ expresses that f is a partial function: for all x, there exists at most one y such that $y\ f\ x$. In keeping with [4], we use the term *imp* (abbreviating "implementation") for a spec f (recall, spec abbreviates "specification") satisfying $I \sqsupseteq f \circ f\cup$. Using the special term "imp" rather than simply "function" avoids confusion with the other notion of functionality introduced in this paper, that in the pers as types setting (for which we use the special term "difunctional".)

Definition 23: $imp.f \;\widehat{=}\; I \sqsupseteq f \circ f\cup$

If $f\cup$ is a imp, we refer to f itself as a *co–imp*; just as imps correspond to functional relations, so co–imps correspond to injective relations. (As noted by van Gasteren in [5], the symmetry between functional and injective relations is lost in many texts, through restricting the notion of injectivity to functions.)

Definition 24: $co\text{–}imp.f \;\widehat{=}\; imp.f\cup$

The notion of functionality is not case in stone, but depends upon our notion of type. Working with monotypes, we have $imp.f \;\widehat{=}\; I \sqsupseteq f \circ f\cup$. We assert that with pers as types, the functional specs are precisely those satisfying $f \sqsupseteq f \circ f\cup \circ f$. Such specs are known as *difunctionals* [6]. (Difunctional relations have also been called "regular" relations and "pseudo–invertible" relations [7].)

Definition 25: $difun.f \;\widehat{=}\; f \sqsupseteq f \circ f\cup \circ f$

We always use small letters $f, g, h, \ldots$ to denote difunctional specs. (This does not clash with the use of these letters to denote imps, since as we shall see, every imp is difunctional.) Since the containment $R \sqsubseteq R \circ R\cup \circ R$ holds for all specs [1], we are free to replace the containment in the above definition by an equality:

Lemma 26: $difun.f \;\equiv\; f = f \circ f\cup \circ f$

Our assertion that the functional specs are precisely the difunctionals is verified by the three results below. The first shows that every functionally typed spec is difunctional, the remaining two that every difunctional can be functionally typed.

Lemma 27: $f \in A \leftarrow B \;\Rightarrow\; difun.f$

Lemma 28: $difun.f \;\Rightarrow\; per.(f \circ f\cup) \;\wedge\; per.(f\cup \circ f)$

Lemma 29: $difun.f \;\Rightarrow\; f \in f \circ f\cup \leftarrow f\cup \circ f$

The last lemma above encodes that the pers $f \circ f\cup$ and $f\cup \circ f$ are respectively left and right domains for difunctional f. We conclude this section with the result that these pers are in fact the least domains for f under the "$\lhd$" ordering.

Lemma 30: $difun.f \;\Rightarrow\; (f \in A\sim B \;\equiv\; f \circ f\cup \lhd A \;\wedge\; f\cup \circ f \lhd B)$

6 More about difunctionals

In the last section we showed that with pers as types, the functional specs are precisely the difunctionals. In this section we document some properties of difunctionals. In particular, we give some conditions under which certain operators preserve difunctionality, and give three other ways to think about difunctionals.

Lemma 31: $per.A \;\Rightarrow\; difun.A$

Lemma 32: $imp.f \;\Rightarrow\; difun.f$

Lemma 33: $difun.f \;\equiv\; difun.f^{\cup}$

Lemma 33 is where the prefix "di" in difunctional comes from, telling us that every difunctional is not just functional, but in fact *bijective*. In conjunction with lemma 32, we see then that every co–imp is difunctional.

Difunctionals are closed under intersection:

Lemma 34: $difun.(f \sqcap g) \;\Leftarrow\; difun.f \;\wedge\; difun.g$

We might suspect difunctionals to be similarly preserved under "$\sqcup$", but this is not so. Disjointness provides however a simple condition to ensure preservation:

Lemma 35: $difun.(f \sqcup g) \;\Leftarrow\; difun.f \;\wedge\; difun.g \;\wedge\; disj.f.g$

In general, difunctionals are not closed under composition. For example,

$$f = \{(a, x), (a, y), (b, z)\}$$

$$g = \{(x, a), (y, b), (z, b)\}$$

are both difunctional (f is an imp, g is a co–imp), but their composition $f \circ g$ is not, as the reader may wish to verify. A simple "type check" however is all that is needed to ensure closure: the composition $(f \circ g)$ of two difunctionals is itself difunctional provided that the least right domain of f subsumes the least left domain of g:

Lemma 36: $difun.(f \circ g) \;\Leftarrow\; difun.f \;\wedge\; difun.g \;\wedge\; f^{\cup} \circ f \;\sqsupseteq\; g \circ g^{\cup}$

Flipping the role of least right and left domains also works:

Lemma 37: $difun.(f \circ g) \;\Leftarrow\; difun.f \;\wedge\; difun.g \;\wedge\; g \circ g^{\cup} \;\sqsupseteq\; f^{\cup} \circ f$

6.1 Disjoint bundles

The full relation on a set $\mathcal{S}$ is given by $\mathcal{S} \times \mathcal{S}$. Relaxing the constraint that both sides of a full relation must be the same set gives what we call *bundles*: relations of the form $\mathcal{S} \times \mathcal{T}$, where $\mathcal{S}$ and $\mathcal{T}$ may differ. Just as pers can be written in terms of full relations, so difunctionals can be written in terms of bundles:

> **Lemma 38:** every difunctional can be written in a unique way as the union of disjoint bundles.

The bundles interpretation of difunctionals is particularly useful in understanding conditions under which operators such as composition preserve difunctionality.

6.2 Co–imp/imp factorisation

Every relation can be factorised in the form $f \circ g\cup$, where f and g are imps. Flipping things around, it is a simple exercise to show that for imps f and g, the spec $f\cup \circ g$ is always difunctional. In fact, we have the following result:

> **Lemma 39:** precisely the difunctional specs can be factorised as the composition of a co–imp and an imp.

This property of difunctionals has important applications in deriving programs that take the form of "representation changers". (See [8] for more details.)

6.3 Invertible specs

An *inverse* of a spec $R \in A\sim B$ is a spec $S \in B\sim A$ satisfying $R \circ S = A$ and $S \circ R = B$. Not all specs $R \in A\sim B$ are invertible, but those that are have a unique inverse, namely $R\cup$. (See Lemma D22 in [4] for the proof.)

> **Lemma 40:** $S \in B\sim A$ is an inverse of $R \in A\sim B \;\Rightarrow\; S = R\cup$

Which specs can be inverted ? Precisely the difunctionals:

> **Lemma 41:** $R \in A\sim B$ is invertible $\Rightarrow$ *difun.R*

> **Lemma 42:** *difun.R* $\Rightarrow$ $R \in R \circ R\cup \sim R\cup \circ R$ is invertible

(That the typing is valid in the last result follows from Lemma 30.)

7 Summary and future directions

Backhouse et al in [4] present a calculational theory of programming, with binary relations as programs and partial identity relations (monotypes) as types. We in this article work within the same framework, but adopt a more general notion of type: partial equivalence relations (pers). Working with pers allows us to handle types with laws. (This is explained in detail in [1].) For example, recalling that $\mathcal{S}^2$ abbreviates the full relation $\mathcal{S} \times \mathcal{S}$, imposing the law $a = b$ on the monotype $\{a\}^2 \cup \{b\}^2 \cup \{c\}^2$ gives the per $\{a, b\}^2 \cup \{c\}^2$. A quite different treatment of types with laws in the *F–algebra* style of type definition has recently been given by Fokkinga [9]; the relationship with our approach is clearly an interesting topic for study.

Working with monotypes, the functional specs (imps) are precisely those f satisfying $I \sqsupseteq f \circ f^{\cup}$. With pers, we found the functional specs to be precisely those f satisfying $f = f \circ f^{\cup} \circ f$, the so–called difunctionals. Moreover, we presented three alternative ways to look at difunctionals: as the union of disjoint bundles, co–imp/imp factorisable specs, and invertible specs. Pers are being used in many areas in computing science (e.g. modelling the λ–calculus and in analysing lazy functional programs.) It seems likely then that difunctionals, not a familiar concept to most computing scientists, have many other interesting applications.

Difunctionals generalise imps; another such generalisation is *causal relations* [10], throwing away the constraint that inputs must always appear on the right–side of a functional relation, and outputs on the left–side. Causal relations are important in the derivation of programs that can be implemented in hardware, being "implementable relations" in Jones and Sheeran's calculational approach to circuit design known as Ruby [11]. A simulator for causal Ruby programs has been produced [12]. We are experimenting with implementing difunctional relations, with a view to producing a more general simulator based upon causal relations as a generalisation of difunctionals rather than imps. Since pers are now adopted as types in Ruby [13, 8], such a simulator should accept Ruby programs earlier in the design process, and in some cases even the initial specification itself.

Acknowledgements

Thanks to the referees and to Erik Meijer for useful comments.

References

[1] Graham Hutton and Ed Voermans. *A calculational theory of pers as types*. Appears in [14]. [To appear as technical reports from Eindhoven and Glasgow, January '92.]

[2] Richard Bird. *Lectures on constructive functional programming.* Oxford University 1988. (PRG–69)

[3] Grant Malcolm. *Algebraic data types and program transformation.* Ph.D. thesis, Groningen University, 1990.

[4] Roland Backhouse, Ed Voermans and Jaap van der Woude. *A relational theory of datatypes.* Appears in [14].

[5] A.J.M. van Gasteren. *On the shape of mathematical arguments.* LNCS 445, Springer–Verlag, Berlin, 1990.

[6] J. Riguet. *Relations Binaires, Fermetures, Correspondances de Galois.* Bulletin de la Societé mathématique de France. Volume 76, 1948.

[7] A. Jaoua, A. Mili, N. Boudriga, and J.L. Durieux. *Regularity of relations: A measure of uniformity.* Theoretical Computer Science 79, 323–339, 1991.

[8] G. Jones and M. Sheeran. *Designing arithmetic circuits by calculation.* Glasgow University, 1991. [In preparation]

[9] Maarten Fokkinga. *Datatype laws without signatures.* Appears in [14].

[10] Graham Hutton. *Functional programming with relations.* Proc. Third Glasgow Workshop on functional programming (ed. Kehler Holst, Hutton and Peyton Jones), Ullapool 1990, Springer Workshops in Computing.

[11] Geraint Jones and Mary Sheeran. *Relations + higher–order functions = hardware descriptions.* Proc. IEEE Comp Euro 87: VLSI and Computers, Hamburg, May 1987.

[12] Graham Hutton. *A simple guide to the Ruby simulator.* Glasgow University, August 1991.

[13] Geraint Jones. *Designing circuits by calculation.* Oxford University, April 1990. (PRG–TR–10–90)

[14] Proc. EURICS Workshop on Calculational Theories of Program Structure, Ameland, The Netherlands, September 1991.