Compositional Characterization of Observable Program Properties

Bernhard Steffen * C. Barry Jay † Michael Mendler‡

Abstract

In this paper we model both program behaviours and abstractions between them as lax functors, which generalize abstract interpretations by exploiting the natural ordering of program properties. This generalization provides a framework in which correctness (safety) and completeness of abstract interpretations naturally arise from this order. Furthermore, it supports modular and stepwise refinement: given a program behaviour, its characterization, which is a "best" correct and complete denotational semantics for it, can be determined in a compositional way.

*University of Aarhus, Denmark
†LFCS, University of Edinburgh, Scotland
‡Universität Erlangen, Germany

1 Introduction

Abstract interpretation is a method for analyzing program *behaviours*, i.e. the relationship between programs and their observable properties [CC77a, CC77b, Nie86, AH87, JN90]. It *abstracts* from standard (denotational) semantics for programming languages to non-standard semantics, which are intended to retain correct (safe), but not necessarily complete, information about given properties of interest. This intention is hard to specify without a precise notion of behaviour, which, despite its primacy, was missing in the framework of abstract interpretation.

In this paper, the notion of behaviour is defined formally as a simple generalization of abstract interpretation, in which operations (specifically, sequential composition) are preserved up to a notion of inequality, which, intuitively, expresses precision of information. It can then be used to specify the properties of programs, which must be respected, both by abstract interpretations and the abstractions between them. This notion of behaviour is not restricted to programming languages, nor need it be derived from a standard denotational semantics. For example, abstractions between semantics can also be viewed as behaviours in our framework, replacing *logical relations* [Plo80], which are symmetric and do not compose, which is counter-intuitive [MJ86].

Moreover, *correctness* and *completeness* of one behaviour for another arise naturally from this precision ordering on properties, and so define a partial order on behaviours. This provides an intuitive and simple notion of correctness and completeness of one abstract interpretation for another via their behaviours, and generalizes the approach using *correctness correspondences* [JN90, MJ86], which aside from being complicated yields a non-transitive notion of correctness.

Unlike denotational semantics or abstract interpretations, behaviours are not, in general, compositional. However, compositionality can be systematically recovered by applying the *characterization functor*, which maps behaviours to the abstract interpretation that identifies programs, which behave identically in any context. This construction preserves *simultaneous observation* and *stepwise construction* of behaviours and therefore permits the hierarchical development of abstract interpretations from behavioural specifications.

The development in this paper is based on the categorical framework. This has two reasons. First, because it provides a very general and well-developed mathematical background for computer science in general, and typed programming languages in particular. Second, the inequalities, which are central to our concept of behaviour, have been studied extensively as lax functors [KS74]. However, neither of these reasons for using categories is imperative, as the point is that behaviours preserve operations up to inequality. This is equally meaningful for untyped languages, where the programs form a set equipped with some operations, and our behaviours are a form of “weak” homomorphism.

Altogether, the paper is structured as follows. After sketching our model in Section 2, we develop our notion of behaviour in Section 3. We introduce simulation relations in Section 3.1 in order to motivate the subsequent development, where behaviours are defined as lax functors (Section 3.3) between ordered categories (Section 3.2). Subsequently, we define the dual notions of correctness and completeness of one behaviour (abstract interpretation) wrt another in Section 3.4. Section 4 presents (Section 4.1) and illustrates

(Section 4.2) the main result of this paper, as well as too corollaries, which establish the modularity and functoriality of our framework (Section 4.3). Finally, Sections 5 and 6 mention conclusions and directions for future work.

2 The Model

Our model consists of ordered categories (similar to O-categories [SP82]), with behaviours corresponding to morphisms between them. It can be sketched by means of the following diagram:

$$\begin{tikzcd} \mathcal{L} \arrow[dr, "\mathtt{D}"] \arrow[dd, "\mathtt{B}"'] & \\ & \mathcal{D} \arrow[dl, "\mathtt{B}'"] \\ \mathcal{O} & \end{tikzcd}$$

$\mathcal{L}$ is a category, which we identify with a programming language: its objects are types and its morphisms are programs. Denotational semantics and abstract interpretations $\mathtt{D} : \mathcal{L} \rightarrow \mathcal{D}$ are both structure-preserving functors (into, say, a category of domains). For the purposes of this exposition, we consider the simplest case, where the only structure of $\mathcal{L}$ is composition.

$\mathcal{O}$ is an ordered category of observations or properties, i.e. its morphisms are ordered in a way compatible with composition, with smaller morphisms representing stronger properties. For example, for strictness analysis, one usually considers $\mathcal{O} = \Omega$, which has one object and two morphisms $\perp$ (reflecting *strictness* wrt the parameter under consideration) and $\top$ (reflecting that *no information* could be inferred) satisfying $\perp \leq \top$.

A behaviour $\mathtt{B}{:}\mathcal{L} \rightarrowtail \mathcal{O}$ is an assignment of properties to programs, which is weakly functorial or compositional, i.e. is a *lax functor* (Section 3.3). For example, the strictness of a composite program $f;g$ cannot be inferred from the strictness of its components f and g. Rather, we have for strictness behaviour $\mathtt{B}$

$$\mathtt{B}(f;g) \ \leq \ \mathtt{B}f\,;\,\mathtt{B}g$$

which allows us to infer correct, but incomplete information about $f;g$ from the behaviours of f and g, e.g. if f and g are both strict then so is $f;g$, but otherwise no information can be deduced.

Let now $\mathtt{B}' : \mathcal{D} \rightarrowtail \mathcal{O}$ be a given behaviour (lax functor) for the semantics $\mathtt{D}$ (e.g. strictness for continuous functions between domains). Then $\mathtt{D}$ is correct for $\mathtt{B}$ if

$$\mathtt{B} \ \leq \ \mathtt{D}\,;\,\mathtt{B}'$$

Completeness is exactly dual, i.e. $\mathtt{D}$ is complete for $\mathtt{B}$ if $\mathtt{D};\mathtt{B}' \leq \mathtt{B}$. Thus, $\mathtt{D}$ is correct and complete for $\mathtt{B}$ if $\mathtt{D};\mathtt{B}' = \mathtt{B}$, as indicated by the diagram above.

The Characterization Theorem 4.6 states that every behaviour has a "best" correct and complete abstract interpretation $\mathbf{Q}\mathtt{B}$, which is its characterization. More precisely, we factorize the lax functor $\mathtt{B}$ as a functor $\mathbf{Q}\mathtt{B}$ followed by a lax functor $\varepsilon_{\mathtt{B}}$. Data types are preserved by $\mathbf{Q}\mathtt{B}$, i.e. it is injective on objects, and it is *computationally relevant*, i.e. surjective on morphisms. Its effect is to identifiy those programs, which have the same behaviour in any context. That $\mathbf{Q}\mathtt{B}$ is the "best" possible such abstract interpretation refers to the following universal property:

$$
\begin{tikzcd}
\mathcal{L} \arrow[dr, "\mathbf{Q}\mathtt{B}"] \arrow[drr, "\mathtt{D}"] \arrow[dd, "\mathtt{B}"'] & & \\
& \mathcal{U} \arrow[dl, "\varepsilon_{\mathtt{B}}"'] & \mathcal{D} \arrow[l] \arrow[dll, "\mathtt{B}'"] \\
\mathcal{O} & &
\end{tikzcd}
$$

Let $\mathtt{D}$ be another abstract interpretation for $\mathtt{B}$, which is correct and complete, datatype preserving, and computationally relevant (Section 4.1). Then $\mathbf{Q}\mathtt{B}$ factors through $\mathtt{D}$ in a unique way.

Behaviours may have structure themselves: they may either represent the simultaneous observation of some more primitive behaviours, or they may be constructed by stepwise abstraction. In fact, this structure is preserved by the characterization functor $\mathbf{Q}$, as can be easily derived from the Characterization Theorem 4.6:

First, $\mathbf{Q}$ is *modular*, i.e. the characterization of a behaviour, which is the simultaneous observation of a pair of behaviours $\mathtt{B}_1$ and $\mathtt{B}_2$, is obtained from their characterizations using categorical products.

Second, $\mathbf{Q}$ is *functorial*. Hence, the characterization of the stepwise abstraction $\mathtt{B}_1; \mathtt{B}_2$ along two behaviours factors through the characterization of $\mathtt{B}_1$[1]

Thus correct and complete abstract interpretations can be constructed hierarchically along the structure of their behavioural specifications, which is reminiscent of the well-known paradigm of software development.

Related Approaches

[CC79, Ste87, Ste89] are concerned with the systematic development of abstract interpretations for imperative languages. Cousot/Cousot consider only the phenomenon of simultaneous observation. Moreover, they do not aim to obtain an abstract interpretation, which satisfies a specific behaviour. Rather, they consider a given abstraction function, and try to mimic the complete semantics (static semantics) on the corresponding codomain as precise as possible.

[1] This is particularly useful for data flow analysis since one can successively abstract from certain program properties, until the universal model $\mathcal{U}$ is decidable. Of course, properties like decidability are not covered by our framework. They must be investigated separately.

In contrast, like this paper, [Ste87, Ste89] are concerned with developing an abstract interpretation that satisfies a given program behaviour, or in their terminology, which cannot be distinguished from its specification on a given level of observation. Whereas [Ste87] only deals with functoriality, [Ste89] also considers modularity.

The categorical approach presented here generalizes and simplifies these approaches.

3 Behaviours of Programs

A programming language is represented by a category $\mathcal{L}$ in our setting; the types of the language are its objects (or if untyped then it has a single object) and the programs are its morphisms. Usually, the language will have further structure (e.g. λ-abstraction or fixpoints), which we would expect semantics to preserve (see Section 6), but here, for the sake of simplicity, we will refrain ¿from assuming more than sequential composition and empty programs, which are the composition and identities, respectively, of $\mathcal{L}$.

Thus, a denotational semantics for $\mathcal{L}$ is a functor $\mathcal{L} \to \mathcal{D}$. Typically, $\mathcal{D}$ is **Dom** the category of domains or, alternatively, one of its full subcategories. For some authors (e.g. [BHA86]) the semantics is represented as a single domain $+\mathcal{D}_\alpha$, namely, the coalesced sum of the objects of $\mathcal{D}$, but this suppression of typing information obscures the functoriality of the semantics. Abstract interpretations are also functors, and may be thought of as non-standard denotational semantics.

Each family of observable properties of $\mathcal{L}$ (e.g. {"strict", "no-information"}) is naturally ordered by implication so that these properties, or observations, form an ordered category (Section 3.2).

A behaviour maps programs (or perhaps denotations) into an ordered category of properties, or observations. The only behaviours of interest are those for which the property of a composite program is at least as strong as that determined by its parts, whence a behaviour is a lax functor (Section 3.3). Lax functors also arise as abstractions between abstract interpretations, e.g. the abstraction map *abs* for strictness of [BHA86]. Once the nature of behaviour is made explicit, the definition of correctness, and the dual notion of completeness, arise naturally from the ordering.

To motivate our definition of behaviours as lax functors into ordered categories, we will begin with simulation relations, which generate an important class of behaviours.

3.1 Simulation Relation

Definition 3.1 *Let $\mathcal{A}$ and $\mathcal{B}$ be categories. A* simulation relation $R : \mathcal{A} \rightarrowtail \mathcal{B}$ *from $\mathcal{A}$ to $\mathcal{B}$ consists of*

(i) *a function, also denoted R, from objects of $\mathcal{A}$ to objects of $\mathcal{B}$, and*

(ii) *for each pair of objects A and A' of $\mathcal{A}$, a relation*

$$R_{A,A'} : \mathcal{A}(A, A') \rightarrowtail \mathcal{B}(RA, RA')$$

between the homsets of $\mathcal{A}$ and $\mathcal{B}$, which together satisfy

(iii) *$g \in Rf$ and $g' \in Rf'$ implies $g; g' \in R(f; f')$ for morphisms*

$$A \xrightarrow{f} A' \xrightarrow{f'} A''$$

$$RA \xrightarrow{g} RA' \xrightarrow{g'} RA''$$

(iv) *for any object A in $\mathcal{A}$ then $id_{RA} \in R_{A,A}(id_A)$.*

Note that the simulation relations that are (partial) functions on the homsets are just (partial) functors.

Example 3.2 A subcategory $\mathcal{A}'$ of $\mathcal{A}$, which contains all of the objects of $\mathcal{A}$, is called *slim.* $\mathcal{A}'$ may be thought of as the collection of morphisms (programs) having some property satisfied by identities and preserved by composition. Then there is a simulation relation $R : \mathcal{A} \rightarrowtail \mathbf{1}$ ($\mathbf{1}$ is the terminal category with one object $*$ and whose sole morphism is id_*) defined by

(i) $RA =_{df} *$ for all objects A

(ii) if $f : A \to B$ then $R_{A,B}\ f =_{df} \begin{cases} \{id_*\} & \text{if } f \text{ in } \mathcal{A}' \\ \{\} & \text{otherwise} \end{cases}$

Conversely, such a simulation relation R determines a slim subcategory of $\mathcal{A}$, whose morphisms are those related to id_* by R. Thus simulation relations with codomain $\mathbf{1}$ directly correspond to program properties that are satisfied by identities and preserved by composition. More complicated behaviours for $\mathcal{A}$ are obtained by expanding the codomain of R.

Simulation relations compose, and so can be used to model stepwise abstraction. Let $R : \mathcal{A} \rightarrowtail \mathcal{B}$ and $S : \mathcal{B} \rightarrowtail \mathcal{C}$ be simulation relations. Then $R;S : \mathcal{A} \rightarrowtail \mathcal{C}$ is the simulation relation defined by

(i) $(R;S)A =_{df} S(R(A))$

(ii) if $f : A \to B$ then $(R;S)_{A,B}f =_{df} \bigcup\{S_{RA,RB}\, g \mid g \in R_{A,B}f\}$.

For example, R may represent a translation into another programming language, whose behaviour is given by S (Section 4.3). Note that $\mathcal{B}$ plays the role of observations for R and also that of language for S. This phenomenon was the reason for us to model observations by categories.

Mycroft and Jones [MJ86] modelled abstraction using logical relations, which are like simulation relations, except that they use a relation between the objects of the two categories instead of a function. This additional freedom allows a single type to be abstracted to a family of types, which is counter-intuitive for abstraction, as is the fact that composites of logical relations are not necessarily logical relations. We will introduce a notion of abstraction that generalizes simulation relations while avoiding these problems (Definition 3.9).

Categorical products are used to represent a pair of morphisms $\mathtt{B} : \mathcal{L} \rightarrowtail \mathcal{O}$ and $\mathtt{B}' : \mathcal{L} \rightarrowtail \mathcal{O}'$ by a single morphism $< \mathtt{B}, \mathtt{B}' >: \mathcal{L} \rightarrowtail \mathcal{O} \times \mathcal{O}'$. The original morphisms are recovered by projection. Thus, if the morphisms are behaviours then the induced behaviour into the product represents their simultaneous observation. Therefore, we claim that any adequate category of behaviours must have products in order to allow the modular construction of complex behaviours from its components. This excludes the category of simulation relations:

Proposition 3.3 *The category of simulation relations does not have binary products.*

Proof: It suffices to show that there is no product of $\mathbf{1}$ with itself, i.e. there is no category $\mathcal{X}$ such that for each category $\mathcal{A}$, the simulation relations $\mathcal{A} \rightarrowtail\!\!\longrightarrow \mathcal{X}$ are in bijection with pairs of slim subcategories of $\mathcal{A}$ (Example 3.2). Assume that such a category $\mathcal{X}$ exists. $\mathbf{1}$ has a unique slim subcategory. Thus there is a unique simulation relation $\mathbf{1} \rightarrowtail\!\!\longrightarrow \mathcal{X}$, which forces $\mathcal{X}$ to have a unique object $*$, whose monoid of endomorphisms $\mathcal{X}(*, *)$ has a unique submonoid, i.e. is trivial. Thus $\mathcal{X}$ is isomorphic to $\mathbf{1}$. On the other hand, simulation relations into $\mathbf{1}$ are in bijection with individual slim subcategories, which yields a contradiction. □

In order to guarantee the modularity of the framework, one must generalize from relations to *lax functors* between ordered categories, whose definition is our next goal.

3.2 Ordered Categories

One abstract interpretation is correct (or safe) wrt another if the denotations of the former have weaker (fewer) properties. To capture this ordering of properties we interpret programming languages in ordered categories, which generalize categories of domains.

Definition 3.4 *An* ordered category *is a category, whose homsets are partially ordered, with composition preserving the order, i.e. if $f \leq g : A \to B$ and $f' \leq g' : B \to C$ then $f; f' \leq g; g' : A \to C$. In short, an ordered category is a category enriched over partial orders [KS74].*

Example 3.5

1. Let **Dom** be the category of domains. With the pointwise ordering of continuous functions it is an ordered category.

2. Let $\mathcal{O}$ be any category. Then its *power category* $\mathbf{P}\mathcal{O}$ has the same objects as $\mathcal{O}$ with homsets given by the powersets of those of $\mathcal{O}$

$$\mathbf{P}\mathcal{O}(A, B) =_{df} \mathbf{P}(\mathcal{O}(A, B))$$

ordered by subset inclusion. The identity for an object A is $\{id_A\}$ and composition is computed pointwise: given two morphisms $f = \{f_i | i \in I\} : A \to B$ and $g = \{g_j | j \in J\} : B \to C$ of $\mathbf{P}\mathcal{O}$ then

$$f; g =_{df} \{f_i; g_j | i \in I, j \in J\}.$$

For instance, let $\mathbf{1}$ be the terminal category with one object $*$ and whose sole morphism is id_*. Then $\mathbf{2} =_{df} \mathbf{P1}$ is the category with one object $*$ and two morphisms $\{\} \leq \{id_*\}$. If $\mathcal{O}$ is a category of properties then $\mathbf{P}\mathcal{O}$ is a category of families of properties, with larger morphisms representing more properties.

3. If $\mathcal{B}$ is an ordered category then $\mathcal{B}^{co}$, the *local dual* of $\mathcal{O}$, is the ordered category obtained by reversing the orders on the homsets. Thus the local duals of power categories represent stronger properties by smaller morphisms, as is usual. For example, in $\mathbf{2}^{co}$ we have $\{id_*\} \leq \{\}$. This ordered category will be used to represent a single property, and so deserves special terminology: we call it Ω and denote $\{id_*\}$ by $\perp$ and $\{\}$ by $\top$.

4. Any ordinary category $\mathcal{L}$ may be coerced to a *discrete ordered category* by giving its homsets the discrete order, i.e. $f \leq g$ iff $f = g$. Then, $\mathcal{L}^{co} = \mathcal{L}$.

5. Categories and simulation relations form an ordered category in the obvious way: Composition is defined in Section 3.1 and clearly, the identity functors are the identity simulation relations. Simulation relations are ordered by letting

$$R \leq S : \mathcal{A} \rightarrowtail B$$

if they agree on objects and if $R_{A,B}f \subseteq S_{A,B}f$ for every morphism $f : A \to B$.

3.3 Lax Functors

Lax functors are a weak notion of functor appropriate to ordered categories and the study of behaviour. The laxness of the functor reflects the loss of information that arises when approximating the behaviour of a large program by composing the behaviours of its parts.

Definition 3.6 *Let $\mathcal{A}$ and $\mathcal{B}$ be ordered categories. A* lax functor *or* behaviour $F : \mathcal{A} \rightarrowtail \mathcal{B}$ *consists of*

(i) *a function, also called F, from objects of $\mathcal{A}$ to objects of $\mathcal{B}$, and*

(ii) *for each pair of objects A and A' of $\mathcal{A}$, an order preserving function*

$$F_{A,A'} : \mathcal{A}(A, A') \to \mathcal{B}(FA, FA')$$

which together satisfy

(iii) *given morphisms $f : A \to A'$ and $g : A' \to A''$ then*

$$F(f;g) \leq Ff;Fg$$

(iv) *given an object A of $\mathcal{A}$ then*

$$Fid_A \leq id_{FA}$$

If these inequalities are actually equalities then F is an *ordered* or *rigid* functor. Also, if the inequalities (iii) and (iv) are reversed (so that $Ff;Fg \leq F(f;g)$ and $id_{FA} \leq \ F\,id_A$) then F is called an *oplax* functor. Note, the oplax functors $F : \mathcal{A} \rightarrowtail \mathcal{B}$ are just lax functors $\mathcal{A}^{co} \rightarrowtail \mathcal{B}^{co}$.

Given a fixed start state, a typical behaviour for an imperative language would be to simply consider the effects of programs on a distinct variable that we regard as input-output parameter. This behaviour is certainly not compositional (i.e. does not define a rigid functor), because side effects of the first program part on other variables may change the effect of the second program part. Thus the behaviour of a composite cannot be inferred from its component behaviours. However, it can be safely approximated by "no information", which guarantees the properties of a lax functor. Another example is the strictness behaviour of functional languages. We will concentrate on this example in the sequel:

Example 3.7

1. Strictness ([AH87]) for **Dom** is given by the lax functor $\mathtt{B}' : \mathbf{Dom} \rightarrowtail \Omega$, which maps all domains to $*$ and which is defined for a continuous function $f : X \to Y$ by

   $$\mathtt{B}'\, f = \begin{cases} \bot & \textit{if } f(\bot) = \bot \\ \top & \textit{otherwise} \end{cases}$$

   Subcategories of **Dom** inherit this behaviour, by composition with the inclusion functor.

2. Let $\mathcal{L}$ be a programming language, i.e. a discrete ordered category. Then every denotational semantics $\mathtt{D} : \mathcal{L} \to \mathbf{Dom}$ yields a strictness behaviour for $\mathcal{L}$:

   $$\mathtt{D}; \mathtt{B}' : \mathcal{L} \rightarrowtail \Omega$$

3. Any functor $F : \mathcal{A} \to \mathcal{B}$ is a lax functor if we regard $\mathcal{A}$ and $\mathcal{B}$ as discrete ordered categories.

4. Composites of lax functors are lax. They can be used for stepwise construction of behaviours. For example, if $T : \mathcal{L} \to \mathcal{L}'$ is a functor (perhaps realizing a translation from $\mathcal{L}$ into $\mathcal{L}'$) and $\mathtt{B}' : \mathcal{L}' \rightarrowtail \mathtt{B}$ is a behaviour for $\mathcal{L}'$ then $T; \mathtt{B}'$ is a behaviour for $\mathcal{L}$ (see Section 4.3).

5. Let $R : \mathcal{L} \rightarrowtail \mathcal{O}$ be a simulation relation. It can be thought of as an oplax functor $\mathcal{L} \rightarrowtail \mathbf{P}\mathcal{O}$ or equivalently, a lax functor $\mathcal{L} \rightarrowtail \mathbf{P}\mathcal{O}^{co}$ (since $\mathcal{L}$ is discrete). For example, slim subcategories of $\mathcal{L}$ correspond to lax functors $\mathcal{L} \rightarrowtail \Omega$.

The ordered categories and lax functors themselves form an ordered category $\mathtt{Ord}$, wherein $F \leq G : \mathcal{A} \rightarrowtail \mathcal{B}$ if F and G agree on objects and $Ff \leq Gf$ for each morphism f. In contrast to simulation relations, lax functors can represent simultaneous observations, as can be inferred from:

Proposition 3.8 $\mathtt{Ord}$ *has cartesian products. The cartesian product of ordered categories $\mathcal{O}$ and $\mathcal{O}'$ is their cartesian product $\mathcal{O} \times \mathcal{O}'$ as ordinary categories with pointwise ordering on the homsets.*

Proof: First note that the pointwise ordering in $\mathcal{O} \times \mathcal{O}'$ ensures that the ordinary projections $\pi_1 : \mathcal{O} \times \mathcal{O}' \to \mathcal{O}$ and $\pi_2 : \mathcal{O} \times \mathcal{O}' \to \mathcal{O}'$ are lax, in fact, rigid functors. Now let $\mathtt{F} : \mathcal{C} \rightarrowtail \mathcal{O}$ and $\mathtt{F}' : \mathcal{C} \rightarrowtail \mathcal{O}'$ be lax functors. Then, pointwise pairing of objects and morphisms defines a lax functor

$$< \mathtt{F}, \mathtt{F}' >: \mathcal{C} \rightarrowtail \mathcal{O} \times \mathcal{O}'$$

It is easy to establish that this lax functor has the universal properties that make $\mathcal{O} \times \mathcal{O}'$ into a categorical product in $\mathtt{Ord}$. $\square$

This proposition can be extended to arbitrary limits, so that general methods of combining observations are possible, e.g. pullbacks could be used to represent sharing constraints.

Lax functors are the promised elaboration of simulation relations, which constitute an adequate notion of abstraction between behaviours, and in particular, abstract interpretations:

Definition 3.9 *Let* $\mathtt{B} : \mathcal{L} \rightarrowtail \mathcal{O}$ *and* $\mathtt{B}' : \mathcal{L} \rightarrowtail \mathcal{O}'$ *be behaviours for* $\mathcal{L}$*. An* abstraction $F : \mathtt{B}' \rightarrowtail \mathtt{B}$ *is a lax functor making the following diagram commute:*

$$
\begin{tikzcd}
 & \mathcal{L} \arrow[dl, dotted, "\mathtt{B}"'] \arrow[dr, dotted, "\mathtt{B}'"] & \\
\mathcal{O} & & \mathcal{O}' \arrow[ll, "F"]
\end{tikzcd}
$$

The behaviours for $\mathcal{L}$ *and the abstractions between them form its* category of behaviours, *denoted* $\mathbf{B}(\mathcal{L})$*. It is also known as the comma category* $\mathcal{L}/\mathtt{Ord}$*.*

3.4 Correctness and Completeness

Let $\mathtt{B}, \mathtt{B}' : \mathcal{L} \rightarrowtail \mathcal{O}$ be two behaviours. As established above, we consider small morphisms in $\mathcal{O}$ to be more informative than large ones. Thus $\mathtt{B}'$ is *correct* (or *safe*) for $\mathtt{B}$ if

$$\mathtt{B} \leq \mathtt{B}'$$

Dually, it is *complete* for $\mathtt{B}$ if $\mathtt{B}' \leq \mathtt{B}$. Correctness implies that $\mathtt{B}'$ yields no more information than $\mathtt{B}$, while completeness implies that it yields at least as much.

Now, fix a programming language $\mathcal{L}$, which we regard as a discrete ordered category, and consider the following diagram of lax functors:

$$
\begin{tikzcd}
\mathcal{L} \arrow[dd, "\mathtt{B}"'] \arrow[dr, "\mathtt{D}"] & \\
 & \mathcal{D} \arrow[dl, "\mathtt{B}'"] \\
\mathcal{O} &
\end{tikzcd}
$$

Then $\mathtt{D}$ is *correct and complete* for $\mathtt{B}$ iff there is a lax functor $\mathtt{B}'$ such that $\mathtt{D};\mathtt{B}'$ is both correct and complete for $\mathtt{B}$, i.e. iff there exists a morphism $\mathtt{B}' : \mathtt{D} \rightarrowtail \mathtt{B}$ in $\mathbf{B}(\mathcal{L})$. — Of particular interest are decidable correct and complete abstract interpretations for $\mathtt{B}$, because they specify complete data flow analysis algorithms for $\mathtt{B}$.

It does not make sense to define either correctness or completeness separately, without first specifying $\mathtt{B}'$, e.g. strictness for domains, since almost every abstract interpretation $\mathtt{D}$ would be correct (complete) for some behaviour on $\mathcal{D}$. This is true in all approaches, though often the behaviour is merely implicit. Logical relations improve on the general situation, but still account for $\mathtt{B}'$ indirectly, at the technical level of domain equations ([JN90, MJ86]). Here $\mathtt{B}'$ is accounted for directly, which yields greater clarity and flexibility: $\mathtt{D}$ is *correct* (*complete*) for $\mathtt{B}$ if $\mathtt{D};\mathtt{B}'$ is correct (complete) for $\mathtt{B}$. This definition of correctness (completeness) is transitive and non-symmetric, as can be illustrated by the following example, involving higher-order strictness analysis. The formalism used here is new: the proofs are in the original paper [BHA86].

Let $\mathcal{L}$ be a programming language generated from a single type A and equipped with a denotational semantics $\mathtt{D} : \mathcal{L} \to \mathcal{D}$, where $\mathcal{D}$ is the full subcategory of **Dom** generated by the image of A in **Dom**. The standard strictness behaviour $\mathtt{B}'$ for $\mathcal{D}$ inherited from **Dom** (Example 3.7(2)) yields strictness for $\mathcal{L}$ via

$$\mathtt{B} =_{df} \mathtt{D};\mathtt{B}'$$

Thus, $\mathtt{D}$ is correct and complete for $\mathtt{B}$ by definition. Let $\mathcal{B}$ be the full subcategory of domains generated by $2 =_{df} \{\bot \leq \top\}$. There is an abstraction $abs : \mathcal{D} \rightarrowtail \mathcal{B}$, which is correct for the strictness behaviour $\mathtt{B}'$.

$$\begin{tikzcd}
\mathcal{L} \arrow[dd, "\mathtt{B}"'] \arrow[dr, "\mathtt{D}'"] \arrow[drr, "\mathtt{D}"] & & \\
& \mathcal{B} \arrow[dl, "\mathtt{B}'"] & \mathcal{D} \arrow[l, "abs"'] \arrow[dll, "\mathtt{B}'"] \\
\Omega & &
\end{tikzcd}$$

¿From this (or directly) can be constructed a (smallest) rigid functor (an abstract interpretation) $\mathtt{D}' : \mathcal{L} \to \mathcal{B}$, which is correct for $\mathtt{D}; abs$. A short diagram-chase now shows that $\mathtt{D}'$ is also correct for $\mathtt{B}$ since $\mathtt{D}';\mathtt{B}' \geq \mathtt{D}; abs; \mathtt{B}' \geq \mathtt{D};\mathtt{B}' = \mathtt{B}$.

Correctness is the critical notion for abstract interpretation, because the safety of a program transformation depends on the correctness of the properties it is based on. Completeness naturally arises as the exact dual of correctness in our framework. Of course, for "standard behaviours", complete abstract interpretations are usually undecidable, and so completeness was neglected. However, there may well be decidable abstract interpretations for "nonstandard behaviours". Thus, completeness can express useful minimal requirements for data flow analysis algorithms. Further, there are situations, where completeness is critical. For example, in data refinement (e.g. [HJ88]) an implementation must have at least the properties of the specifying abstract data type. We conjecture that these properties define a behaviour in our framework, and that successful data refinement is simply completeness there.

4 Characterization of Behaviour

We wish to construct an abstract interpretation from a behaviour. Each behaviour yields an equivalence relation on the programs obtained by relating those programs, which behave identically. Abstract interpretations are behaviours that are characterized by yielding a congruence relation.

The point of the characterization functor is to associate each behaviour with an abstract interpretation that corresponds to the largest congruence, which refines the equivalence relation of the behaviour, i.e. which relates programs that have the same behaviour in any context. This yields a categorical congruence (cf. [Mac71, BW85]) on the category of programs, whose quotient will be the desired characterization of the original behaviour.

4.1 The Characterization Functor

Definition 4.1 *Let $\mathcal{C}$ be a category. A* congruence *on $\mathcal{C}$ is a family $E_{A,B}$ of equivalence relations on the homsets $\mathcal{C}(A,B)$ (where $E_{A,B}(f,f')$ is written $f \equiv f'$ when the congruence E is understood) satisfying, for $f, f' : A \to B$ and $g, g' : B \to C$*

$$f \equiv f' \text{ and } g \equiv g' \text{ imply } f;g \equiv f';g'$$

Given a congruence E on a category $\mathcal{C}$ there is a *quotient category* $\mathcal{C}(E)$ having the same objects as $\mathcal{C}$ whose morphisms are the equivalence classes of morphisms in $\mathcal{C}$.

Of course, there is also a *quotient functor* $Q : \mathcal{C} \to \mathcal{C}(E)$, which maps each morphism to its congruence class. It is injective on objects (*preserves datatypes*) and is also surjective on objects and morphisms (is *computationally relevant*). The *category of quotients* $\mathbf{Q}(\mathcal{L})$ is the full subcategory of $\mathbf{B}(\mathcal{L})$ of quotient functors, where we consider quotient functors as lax functors between discrete ordered categories (see Example 3.3.3). The universal property of quotient functors is given by

Proposition 4.2 *Let E be a congruence on $\mathcal{C}$ with quotient Q. If $H : \mathcal{C} \leftarrowtail\!\!\!\rightarrow \mathcal{O}$ is a lax functor such that for all morphisms $f, f' : A \to B$ in $\mathcal{C}$*

$$f \equiv f' \text{ implies } Hf = Hf'$$

then there is a unique lax functor $H' : \mathcal{C}(E) \leftarrowtail\!\!\!\rightarrow \mathcal{O}$ satisfying $Q; H' = H$.

$$\begin{tikzcd}
 & \mathcal{C} \arrow[dl, dotted, "H"'] \arrow[dr, dotted, "Q"] & \\
\mathcal{O} & & \mathcal{C}(E) \arrow[ll, dotted, "H'"]
\end{tikzcd}$$

Moreover, if H is an rigid functor then H' is an rigid functor too.

Proof: (Sketch) H' agrees with H on objects, and maps a congruence class $[f]$ of morphisms to Hf. $\square$

Example 4.3 Let $\mathtt{D} : \mathcal{L} \to \mathcal{D}$ be a denotational semantics and define two parallel morphisms f and f' to be *denotationally equivalent*, $E_{\mathtt{D}}(f, f')$, if $\mathtt{D}f = \mathtt{D}f'$. Then $\mathtt{D}$ factorizes through the corresponding quotient functor $Q\mathtt{D}$ in a unique way:

$$\begin{tikzcd}
& \mathcal{L} \arrow[dl, dotted, "\mathtt{D}"'] \arrow[dr, dotted, "Q\mathtt{D}"] & \\
\mathcal{D} & & \mathcal{L}(E_{\mathtt{D}}) \arrow[ll, dotted, "F"]
\end{tikzcd}$$

We have:

Proposition 4.4 *$\mathbf{Q}(\mathcal{L})$ is a meet semi-lattice.*

Proof: Let $Q : \mathcal{L} \to \mathcal{U}$ and $Q' : \mathcal{L} \to \mathcal{U}'$ be quotient functors arising from congruences E and E' respectively. It follows from Proposition 4.2 that there is at most one lax functor $F : \mathcal{U} \rightarrowtail \mathcal{U}'$ satisfying $Q; F = Q'$, which must then be a quotient. We then say $Q \leq Q'$. Such an F exists iff $E \subseteq E'$ that is, $E(f, f')$ implies $E'(f, f')$. The meet of Q and Q' (their categorical cartesian product) is the quotient corresponding to $E \cap E'$. $\square$

Definition 4.5 *Let $\mathtt{B} : \mathcal{L} \rightarrowtail \mathcal{O}$ be a behaviour. Morphisms $f, f' : A \to B$ in $\mathcal{L}$ are* behaviourally congruent *if, for all morphisms $g : A' \to A$ and $h : B \to B'$ we have*

$$\mathtt{B}(g; f; h) = \mathtt{B}(g; f'; h)$$

that is, f and f' have the same behaviour in every (input-output) context. Then the quotient functor $\mathbf{Q}\mathtt{B} : \mathcal{L} \to \mathcal{U}$ corresponding to this congruence is the characterization *of the behaviour* $\mathtt{B}$.

Applying Proposition 4.2 to the behavioural congruence on $\mathcal{L}$ generated by $\mathtt{B}$ with $H = \mathtt{B}$ shows that $\mathtt{B} = \mathbf{Q}\mathtt{B}; \varepsilon_{\mathtt{B}}$ for some behaviour $\varepsilon_{\mathtt{B}}$, i.e. $\mathbf{Q}\mathtt{B}$ is correct and complete for $\mathtt{B}$. This characterization of behaviours is the object part of the functor $\mathbf{Q}$ specified in the following theorem:

Theorem 4.6 (Characterization Theorem)
$\mathbf{Q}(\mathcal{L})$ is a coreflective subcategory of $\mathbf{B}(\mathcal{L})$, i.e. the inclusion of $\mathbf{Q}(\mathcal{L})$ in $\mathbf{B}(\mathcal{L})$ has a right adjoint $\mathbf{Q}$, the characterization functor.

$$
\begin{tikzcd}
\mathcal{L} \arrow[dr, "\mathbf{Q}\mathtt{B}"] \arrow[drr, "Q'"] \arrow[dd, "\mathtt{B}"'] & & \\
 & \mathcal{U} \arrow[dl, "\varepsilon_{\mathtt{B}}"'] & \mathcal{U}' \arrow[l, "F"] \arrow[dll, "\mathtt{B}'"] \\
\mathcal{O} & &
\end{tikzcd}
$$

Proof: Let $\mathtt{B} : \mathcal{L} \rightarrowtail \mathcal{O}$ be a behaviour. Then its image under $\mathbf{Q}$ is defined to be the quotient functor $\mathbf{Q}\mathtt{B} : \mathcal{L} \to \mathcal{U}$ as described in Definition 4.5.

The counit of the coreflection is $\varepsilon_{\mathtt{B}} : \mathbf{Q}\mathtt{B} \rightarrowtail \mathtt{B}$. To see its universal property, let $Q' : \mathcal{L} \to \mathcal{U}'$ be another quotient functor, which is correct and complete for $\mathtt{B}$, i.e. $Q'; \mathtt{B}' = \mathtt{B}$ for some behaviour $\mathtt{B}'$. Then $Q'f = Q'g$ implies that f and g are behaviourally congruent since Q' is a functor. Thus, $\mathbf{Q}\mathtt{B}(f) = \mathbf{Q}\mathtt{B}(g)$ and so applying Proposition 4.2 with Q' as quotient shows there is a unique functor $F : \mathcal{U}' \to \mathcal{U}$ making all triangles in the diagram above commute. $\square$

Note that the universal property is more general than it may at first appear, since Example 4.3 shows that every abstract interpretation factorises through some quotient functor.

The Characterization Theorem 4.6 generalizes the well-known result that there exists a unique largest congruence relation in every equivalence relation. — Let us now illustrate the situation obtained so far by means of strictness analysis.

4.2 Strictness Analysis

The behaviour of a program is usually given by the behaviour of its denotation, but may also be determined in other ways, e.g. by first manipulating the syntax. Here both methods are used to obtain strictness analyses [AH87] for some simple languages, which illustrate the main features of this framework. First, we consider the behaviour of the denotations.

Let $\mathcal{D}$ be the full subcategory of domains generated by $\mathbf{N}_\perp$ the flat natural numbers. Its behaviour $\mathtt{B}' : \mathcal{D} \rightarrowtail \Omega$ is induced by the strictness behaviour of **Dom** (Example 3.7(2)). The structure of **Dom** is so rich that it prevents identifications through behavioural congruence (unlike many languages).

Lemma 4.7 *The characterization for the strictness behaviour* $\mathtt{B}' : \mathbf{Dom} \rightarrowtail \Omega$ *on domains is the identity.*

Proof: Let $f, g : D \to D'$ be continuous functions, which are behaviourally congruent. Given $x \in D$ let $h : D' \to 2$ be the unique continuous function specified by $h^{-1}(\perp)$ is the down-closure of $f(x)$ in D'. Then $f \equiv g$ implies $\mathtt{B}'(hf(x)) = \mathtt{B}'(hg(x))$ whence $g(x) \leq f(x)$. By symmetry, $f(x) \leq g(x)$ and so $f(x) = g(x)$. $\square$

Consider a simply typed λ-calculus with natural numbers whose types are freely generated by a type N (natural numbers) and which is equipped with *zero* $0 : N$ and *successor* $s : N \to N$, and perhaps some other constants. Let $\mathcal{L}$ be the corresponding category whose objects are the types, and whose morphisms $X \to Y$ are equivalence classes under α-conversions of terms $t : Y$ equipped with a context Γ of type X. Additional conversions (e.g. the β- and η- conversions, which would make the category cartesian closed [LS86]) are not imposed since they are not syntactic, but arise from the behaviour.

The standard denotational semantics for $\mathcal{L}$ is given by $\mathtt{D} : \mathcal{L} \to \mathcal{D}$, where N is mapped to $\mathbf{N}_\perp$ and constants, including zero and successor, receive their standard interpretation as lifted functions (though non-deterministic choice requires powerdomains, see below). The behaviour for $\mathcal{L}$ is then given by $\mathtt{B} =_{df} \mathtt{D}\,;\mathtt{B}' : \mathcal{L} \rightarrowtail \Omega$.

The constant numerals of $\mathcal{L}$, e.g. $0, s0, \ldots$, when regarded as morphisms $N \to N$ with free variable $x : N$, e.g. $\lambda x.0\,, \lambda x.s0, \ldots$ are all non-strict, while the denotation of a variable x is the identity, which is strict. Thus, numerals and variables are not behaviourally congruent. If the language is pure, i.e. there are no other constant symbols, then an inductive argument shows that the constant numerals are all behaviourally congruent. However, in the presence of additional constants, more distinctions can be made. Consider, for example

(i) addition, $+ : N \times N \to N$

(ii) bottom, $\perp : N$

(iii) non-deterministic choice, $\mid\; : N \times N \to N$

(The denotation of non-deterministic choice requires powerdomains, though its strictness behaviour is clear: it is strict iff both of its arguments are.)

There are now six separate congruence classes of morphisms $N \times N \to N$ (equivalently, $N \to N \to N$), represented by the following λ-terms:

$$\begin{tikzcd}
& 0 \arrow[d, dash] & \\
& x \mid y \arrow[dl, dash] \arrow[dr, dash] & \\
x \arrow[dr, dash] & & y \arrow[dl, dash] \\
& x + y \arrow[d, dash] & \\
& \perp &
\end{tikzcd}$$

They correspond to the strictness values of Burn, Hankin and Abramsky [BHA86] for this type, which form the domain $2 \to 2 \to 2$. However, if the language has fewer constants then there are fewer congruence classes, which is not reflected in their model since it is independent of the language.

Conversely, more constants may yield more distinctions. For example, let $\mathcal{L}$ also have a conditional, $c : N \to N \to N \to N$, whose denotation is given by

$$\mathtt{D}(c)\; b\; m\; n = \begin{cases} m & \textit{if } b = 1 \\ n & \textit{if } b = 0 \\ \perp & \textit{otherwise} \end{cases}$$

Then truth values (where *true* and *false* are represented by 1 and 0 respectively) are distinguished from each other and from the other constant numerals. By contrast, their abstraction *abs* (Section 3.4) identifies all the numerals. Thus *abs* is incomplete for $\mathtt{B}$, or more precisely, $\mathtt{D}\,;\, abs;\, \mathtt{B}' > \mathtt{B}$.

Note that often the strictness of first-order functions is all that we are interested in. However, the characterization of the corresponding behaviour is the same as that of $\mathtt{B}$ since higher-order morphisms yield first-order morphisms in appropriate contexts. Thus, the behaviour of interest may be extremely simple, and yet specify complicated abstract intepretations.

4.3 Compositionality of the Characterization

Behaviours may be constructed by means of simultaneous observation $< \mathtt{B}, \mathtt{B}' >$ and stepwise abstraction $\mathtt{B}'; \mathtt{B}''$. In this section, we establish how to hierarchically develop the characterization of a behaviour along this structure. This is based on two corollaries to the Characterization Theorem 4.6.

Corollary 4.8 (Modularity)
Q preserves all limits in $\mathbf{B}(\mathcal{L})$. In particular, given two behaviours $\mathtt{B} : \mathcal{L} \rightarrowtail \mathcal{O}$ and $\mathtt{B}' : \mathcal{L} \rightarrowtail \mathcal{O}'$ then the characterization $Q < \mathtt{B}, \mathtt{B}' >$ of their simultaneous observation is the meet of $\mathbf{Q}\mathtt{B}$ and $\mathbf{Q}\mathtt{B}'$.

Proof: Right adjoints preserve limits. □

This result generalizes the well-known fact that the intersection of two congruence relations is a congruence relation itself.

Example 4.9 Let $\mathcal{L}$ be the richest language considered in Section 4.2. For $m > 0$ we define a non-standard denotational semantics $\mathtt{D}_m : \mathcal{L} \to \mathcal{D}$, which differs from $\mathtt{D}$ in that $\mathtt{D}(s)$ is the successor *mod m*, i.e. the lifted function $n \mapsto n{+}1$ (*mod m*). Let $\mathtt{B}_m =_{df} \mathtt{D}_m; \mathtt{B}'$ be the corresponding behaviour of $\mathcal{L}$. Then the congruence classes of numerals are those of *mod-m* arithmetic and $\{\perp\}$. These cannot be identified since every numeral can be mapped to the congruence class of 0 (= *"false"*) by sufficient applications of s.

Simultaneous observation of $\mathtt{B}_m$ and $\mathtt{B}_n$ is characterized by $Q\mathtt{B}_q$, where q is the least common multiple of m and n. Note that $Q\mathtt{B}_q$ distinguishes only those programs, which need to be distinguished for realizing simultaneous *mod-m* and *mod-n* observations.

Corollary 4.10 (Functoriality)
Let $\mathtt{B} = \mathtt{B}'\,;\, \mathtt{B}'' : \mathcal{L} \rightarrowtail \mathcal{O} \rightarrowtail \mathcal{O}'$ be a composite of lax functors. Then we have

$$\mathbf{Q}\mathtt{B} \;=\; \mathbf{Q}\mathtt{B}'\,;\, Q_{\mathtt{B}',\mathtt{B}}(\mathtt{B}'') \;=\; \mathbf{Q}\mathtt{B}';\mathbf{Q}(\varepsilon_{\mathtt{B}'};\mathtt{B}'')$$

In particular, $Q\mathtt{B}$ factors through $Q\mathtt{B}'$.

Proof: The lax functor $\mathtt{B}'' : \mathtt{B}' \rightarrowtail \mathtt{B}$ is a morphism of $\mathbf{B}(\mathcal{L})$. Since functors preserve domain and codomain of morphisms, we have $Q_{\mathtt{B}',\mathtt{B}}(\mathtt{B}'') : \mathbf{Q}\mathtt{B}' \to \mathbf{Q}\mathtt{B}$, which yields the result. $\square$

Stepwise abstraction of behaviours arises naturally in the search for the right level of abstraction. Consider data flow analysis: Decidable abstract interpretations directly specify data flow analysis algorithms. Usually however, the abstract interpretation associated with a certain data flow problem is not decidable. Thus further abstractions are necessary. A common such abstraction step is to interpret conditional branching by non-deterministic choice. It can be realized by a syntactic translation as in the following

Example 4.11 The conditional $c\,x\,y\,z$ can be translated into the non-deterministic choice $y \mid z$. This syntactic translation determines a functor $\mathtt{T} : \mathcal{L} \to \mathcal{L}$ (which is not mirrored by any endo-functor on the category of denotations). It yields a new behaviour $\mathtt{B}_1 = \mathtt{T};\mathtt{B}$ on $\mathcal{L}$, which is correct for $\mathtt{B}$ without being complete for it. Thus, given $\varepsilon_{\mathtt{B}_1}$ then $Q\mathtt{B}_1$ is correct for $\mathtt{B}$ and complete for $\mathtt{B}_1$. Now functoriality shows that $Q\mathtt{B}_1$ decomposes as $Q(\mathtt{T}); Q(\varepsilon_{\mathtt{T}}; B)$, which may thus simplify its calculation.

5 Conclusion

We have presented a language independent framework for abstract interpretation that explicitly deals with behaviours of programs, with the benefit that the notion of correctness is simplified and the notion of completeness naturally arises as its dual. These improvements do not require considering observations (properties) as morphisms of a category. The usual relational approach with sets of observations would do. However, our framework additionally supports the hierarchical development of abstract interpretations and data flow analysis algorithms along the structure of the specifying program behaviour, by means of stepwise and modular refinement in the categorical framework. All these features have been illustrated by means of some simple strictness analyses.

6 Future Work

In this paper the characterization of a behaviour is universal amongst quotient functors. It therefore focuses on substitution as a language construct and on datatype preserving abstract interpretations. This suggests two directions for generalizations. First, other language constructs such as fixpoints, products, general limits, or λ-abstraction should be considered. We believe that the development in this paper can be reformulated for quotient functors that also preserve these language constructs, to achieve this generalization. Second, one could generalize to abstract interpretations that do not necessarily preserve data types. Here an approach using “coequalizers” rather than“quotient functors” seems appropriate.

7 Acknowledgements

The development of this paper has been strongly influenced by discussions with Eugenio Moggi. Furthermore, we would like to thank Yves Lafont, Don Sannella and Terry Stroup

for helpful comments, and Norbert Götz for giving us a hand in typing the manuscript.

References

[AH87] S. Abramsky and C. L. Hankin, editors. *Abstract interpretation of declarative languages*. Ellis-Horwood, 1987.

[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.

[BW85] M. Barr and C. Wells. *Toposes, Triples and Theories*. Springer Verlag, 1985.

[CC77a] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, 1977.

[CC77b] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *ACM Sigplan Notices*, 12:1–12, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282, 1979.

[HJ88] C.A.R. Hoare and H. Jifeng. Data refinement in a categorical setting. Technical Report February, Oxford Univ. Computing Lab., 1988.

[JN90] N. D. Jones and F. Nielson. Abstract interpretation: A semantics based tool for program analysis. In *Handbook of logic in computer science*. to appear, 1990.

[KS74] G. M. Kelly and R. Street. Review of the elements of 2-categories. In G. M. Kelly, editor, *Proceedings Sydney Category Theory Seminar 1972/1973*, pages 75–103. Springer-Verlag, 1974.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[Mac71] S. MacLane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[MJ86] A. Mycroft and N. D. Jones. A relational framework for abstract interpretation. In *proceedings, 'programs as data objects'*. Springer Verlag, LNCS 217, 1986.

[Nie86] F. Nielson. A bibliography on abstract interpretations. *ACM Sigplan Notices*, 21:31–38, 1986.

[Plo80] G.D. Plotkin. Lambda definability in the full type hierarchy. In R. Hindley and J. Seldin, editors, *To H.B. Curry: essays in Combinatory Logic, lambda calculus and Formalisms*. Academic Press, 1980.

[SP82] M. Smith and G. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11, 1982.

[Ste87] B. Steffen. Optimal run time optimization - proved by a new look at abstract interpretations. In *TAPSOFT '87*, pages 52–68. LNCS 249, 1987.

[Ste89] B. Steffen. Optimal data flow analysis via observable equivalence. In *MFCS'89*, 1989.