Theoretical Computer Science 121 (1993) 113–143
Elsevier

Type inference, abstract interpretation and strictness analysis*

Mario Coppo and Alberto Ferrari

Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy

Abstract

Coppo, M. and A. Ferrari, Type inference, abstract interpretation and strictness analysis, Theoretical Computer Science 121 (1993) 113–143.

Filter domains (Coppo et al., 1984) can be seen as abstract domains for the interpretation of (functional) type-free programming languages. What is remarkable is the fact that in filter domains the interpretation of a term is given by the set of its types in the intersection type discipline with inclusion, thus reducing the computation of an abstract interpretation to typechecking. As a main example, an abstract filter domain for strictness analysis of type-free functional languages is presented. The inclusion relation between types representing strictness properties has a complete recursive axiomatization. Type inference rules cannot be complete (strictness being a Π^0_1 property), but a complete extension of the type inference system is presented.

1. Introduction

Abstract interpretation is an elegant and useful framework to study a number of methods for extracting informations from programs, usually with the aim of performing complile time optimizations. The basic idea is to define interpretations of the source language in "abstract" (usually finite) domains whose elements represent, roughly speaking, properties of elements of the initial "standard" domain, which can be mapped homomorphically in the abstract domain. A basic request to this mapping is that of being safe, in the sense that if an element x is mapped to x' then the property represented by x' is surely a property of x. What is usually not required from abstract

Correspondence to: M. Coppo, Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy. Email addresses of the authors: coppo@di.unito.it and ferrari@di.unito.it.

* Research partially supported by M.P.I. 60%, comitato per la matematica.

interpretations is completeness since this, in most cases, immediately leads to undecidability while abstract interpretations are expected to be computable in a reasonable time. Abstract interpretation has been applied to the study of several kinds of program analysis like data flow analysis [25] and strictness analysis [9]. Strictness analysis, in particular, is one of the most interesting applications of abstract interpretation for functional languages.

Abstract interpretation has been developed mainly for first-order, simply typed languages. In this case, in fact, the number of elements of both the basic and the functional abstract domains is small, and the computation of abstract interpretations is easy. For instance, taking a language defined on the flat c.p.o. $N_\perp$ of integers the abstract domain for strictness analysis A has only two elements: $\{\perp\}$ and $\{N_\perp\}$ representing, respectively, the property of "being the value $\perp$" and that of "being any value" (possibly $\perp$). The space of continuous functions $[A \to A]$ has then only three elements: the function constantly $\perp$ (which represents the abstract interpretation of the function constantly $\perp$ on the standard domain), the identity (which represents the abstract interpretation of strict functions) and the function constantly $\top$ (which can be the abstract interpretation of any function). When languages with higher-order functions are considered as in [7], the number of elements of the abstract domain increase exponentially with the complexity of types and the standard techniques for the evaluation of abstract interpretation become quickly impractical. It seems more difficult to generalize the notion of abstract interpretation to polymorphic languages. A possible way out is to develop abstract interpretations for typed higher-order languages [7] and to use the notion of polymorphic invariance [1, 4]. It seems even more difficult to define a notion of abstract interpretation for type-free languages. No satisfactory attempt of doing this is known to the authors.

In this paper, we use filter domains [6] to define a notion of abstract semantics for (higher-order) type-free or polymorphic functional languages, in which the same term can be applied to arguments of different types. The elements of a filter domain are defined as sets of formal types representing properties of values closed under implication and conjunction. Types are defined from a set of basic types by the $\to$ (function space) and $\wedge$ (intersection) type constructors, and are associated with terms by a formal type inference system with inclusion. The interpretation of types as subsets determines an inclusion relation that can often be axiomatized in a simple way and characterizes the properties of the type assignment system and of the associated filter domain (our "abstract" domain).

A basic feature of our approach is that, instead of introducing a simplified abstract language to compute abstract interpretations (see e.g. [9]), we interpret directly the basic language in the abstract domain reducing the computation of abstract interpretation to typechecking. This view is especially interesting in the case of higher-order functions where types are a natural way of representing complex functional properties. The inference procedures for a specific type system (like the one for strictness analysis), moreover, can be combined with standard type inference algorithm [16] in order to reduce the total amount of compile time effort. Typechecking of

intersection types is in general undecidable [6], but a complete inference procedure is known [12, 26] and decidable restrictions have been studied [14].

The fact that domains can be defined by taking as elements the collection of subsets (types) of a given family closed under inclusion and intersection is well known. An approach to the theory of domains based on this idea has been given in [28], and then further developed in [29], leading to the notion of information system. Filter domains, indeed, can be seen as particular cases of information systems in which the interpretation of the elements of the formal filters as types is made explicit. The basic properties of filter domains have been investigated in [11]. The aim of this paper is that of linking the notion of filter domain with that of abstract interpretation.

A similar approach to the theory of domains has been developed, in a more general framework, in [2], where a notion of domain logic as a tool to reason about elements of domains based on Stone duality is introduced. The technique developed in [2] gives a tool to find a "logical" representation of a given domain via an isomorphism in which each point of the domain is represented by a set of formulae characterizing it. In our approach, we are interested, instead, in finding out the (abstract) domain considering only *some* properties of its elements (the ones under investigation). This can lead, for instance, to the identification of points which share the same properties. What we find is not, in general, an isomorphism but rather an embedding of the standard domain into the abstract one. The structure of a filter domain is determined essentially by the inclusion relation between types induced by the properties under investigation. Filter domains, however, can turn out to be isomorphic to solutions of recursive domain equations (as it happens for the examples of this paper).

The basic example developed in this paper is strictness analysis. The abstract (reflexive) domain will be obtained as the filter model determined by a type system for the study of strictness properties of lazy functional languages. We will show, in particular, that the inclusion relation determined by the interpretation of types as strictness properties has a complete finite axiomatization.

An application of domain logic [2] to the investigation of strictness properties of a simply typed functional language has been given in [18]. The main achievement of [18] is a logical representation of the abstract domains defined in [7] in which the logical formulae (types) are interpreted as ideals over the (abstract) domains. This approach is useful in order to use inference to prove strictness properties of simply typed terms, but cannot be extended to polymorphic or type-free languages. In [18], moreover, no link is established between abstract domains and standard ones (for this purpose the paper relies on the results of [7]).

The idea of using type inference to prove strictness properties of higher-order functional languages was first introduced by Kuo and Mishra [19, 20], but there are many substantial differences between their approach and ours. The type system of Kuo and Mishra is sound for a head reduction evaluation strategy (for which $\lambda x.M$ diverges if M diverges), while we consider a lazy evaluation in the sense of [3] (in which $\lambda x.M$ is not divergent even if M is). We also prove stronger completeness results for our inference system, with respect to both the inclusion relation and the type assignment rules.

The type assignment system for strictness properties however is not complete (even if it determines a filter domain which is indeed a λ-model) in the sense that not all types that represent a property true of a term can be assigned to it. This is a consequence of the fact that strictness for higher-order languages (based on lambda-calculus) is a Π_1^0 property while the finitary inference system is Σ_1^0. A complete extension of the inference system can be defined by adding an infinitary rule based on the notion of approximant of a term [5]. This extended system will be used, for instance, to justify our treatment of fixed points in the abstract interpretation.

Section 2 is devoted to the introduction of the basic notions about type inference and type interpretation, while filter models and their connections with abstract interpretation are presented in Section 3. Section 4 presents the inference system for the study of strictness properties. The basic completeness theorems are proved in Section 5.

2. Type assignment and type interpretation

In this section we give a short survey of the intersection type system for terms of a functional programming language. For more details and insight, see [8]. We assume that the reader is familiar with the basic type inference system of ML-like languages.

Intersection types can naturally be introduced in type inference systems in which the same term can have many different types. In this context we can introduce an operator of type intersection $\wedge$ which allows to assign different types to the same term. A type of the form $\alpha \wedge \beta$ can be interpreted as the type of terms which have *both* type α *and* type β.

Let us consider, for instance, the operator of self-application $\lambda x.xx$, which has no type in the basic system because x should be assigned a type α such that $\alpha = \alpha \to \beta$, which is clearly impossible. Using $\wedge$ we can assume $x : \alpha \wedge (\alpha \to \beta)$ (i.e. x has both types α *and* $\alpha \to \beta$) from which we can deduce $x : \alpha \to \beta$, $x : \alpha$ and, using $(\to \mathrm{E})$, $(xx) : \beta$. Then using $(\to \mathrm{I})$ we have $\lambda x.xx : \alpha \wedge (\alpha \to \beta) \to \beta$.

It is useful, moreover, to assume a basic type ω to be interpreted as a "universal" type that can be assigned to any term (including the unsolvable ones). For instance, let $P = \lambda y.\lambda x.x$. A natural type for P is $\omega \to \alpha \to \alpha$ for all types α, meaning that P can be applied to any term giving a term of type $\alpha \to \alpha$. This is particularly sensible in a language with lazy evaluation. In fact, since P will never evaluate its argument, there is no need to assume any type constraint for it.

Let K be a set of *basic types* (like **int** and **bool**). The set of types over K is the least set T^K (T when K is understood) such that

(1) $K \subseteq T^K$,
(2) $\omega \in T^K$,
(3) $\alpha, \beta \in T^K \Rightarrow \alpha \to \beta \in T^K$,
(4) $\alpha, \beta \in T^K \Rightarrow \alpha \wedge \beta \in T^K$.

We agree that $\wedge$ takes precedence over $\rightarrow$, i.e. $\gamma \wedge \alpha \rightarrow \beta$ is equivalent to $(\gamma \wedge \alpha) \rightarrow \beta$. Intersection types like $\gamma \wedge \alpha$ are considered modulo permutations and repetitions of the same type, i.e. $\alpha \wedge \beta$ is identified with $\beta \wedge \alpha$ and $\alpha \wedge \alpha$ with α.

2.1. Inclusion theories

The interpretation of types as sets of values leads naturally to the notion of type inclusion. We have, for instance, that (the interpretation of) type $\alpha \wedge \beta$ is included in (the interpretation of) types α and β. We introduce a formal relation $\leqslant$ to represent type inclusion.

An *inclusion statement* is an expression $\alpha \leqslant \beta$ (where $\alpha, \beta \in T^K$) whose intended meaning is that (the interpretation of) α is a subset of (the interpretation of) β. An *inclusion context* (over a set of types T^K) is a set Σ of inclusion statements. Type inclusions between types are proved in a formal deduction system where we have judgements of the shape $\Sigma \vdash \alpha \leqslant \beta$, where Σ is an inclusion context, meaning that $\alpha \leqslant \beta$ can be proved from the assumptions in Σ. Usually, T^K is implicit in Σ and will not be formally mentioned. The following definition gives the axiomatization of the relation $\leqslant$.

Definition 2.1. (i) The formal system for inclusion judgements is defined by the following rules.

(Ax1) $\Sigma \vdash \alpha \leqslant \beta$ if $\alpha \leqslant \beta \in \Sigma$

(Ax2) $\Sigma \vdash \alpha \leqslant \omega$ for all $\alpha \in T$

(Ax3) $\Sigma \vdash \alpha \rightarrow \omega \leqslant \omega \rightarrow \omega$ for all $\alpha \in T$

(Ax4) $\Sigma \vdash \alpha \leqslant \alpha \wedge \alpha$

(Ax5) $\Sigma \vdash \alpha \leqslant \alpha$

(Ax6) $\Sigma \vdash \alpha \wedge \beta \leqslant \alpha \qquad \Sigma \vdash \alpha \wedge \beta \leqslant \beta$

(Ax7) $\Sigma \vdash (\alpha \rightarrow \gamma) \wedge (\alpha \rightarrow \beta) \leqslant \alpha \rightarrow \gamma \wedge \beta$

$$(\wedge) \qquad \frac{\Sigma \vdash \alpha \leqslant \alpha' \qquad \Sigma \vdash \beta \leqslant \beta'}{\Sigma \vdash \alpha \wedge \beta \leqslant \alpha' \wedge \beta'}$$

$$(\rightarrow) \qquad \frac{\Sigma \vdash \alpha' \leqslant \alpha \qquad \Sigma \vdash \beta \leqslant \beta'}{\Sigma \vdash \alpha \rightarrow \beta \leqslant \alpha' \rightarrow \beta'}$$

$$(\text{trans}) \qquad \frac{\Sigma \vdash \gamma \leqslant \alpha \qquad \Sigma \vdash \alpha \leqslant \beta}{\Sigma \vdash \gamma \leqslant \beta}$$

(ii) Let $\Sigma \vdash \alpha \sim \beta$ denote $\Sigma \vdash \alpha \leqslant \beta$ and $\Sigma \vdash \beta \leqslant \alpha$.

The less intuitive points in the definition of type inclusion are perhaps Ax3, Ax7 and rule (→). In case of Ax3 note that ω is intended to represent the whole domain, and that in our language any function can be seen as a function on the whole domain (giving possibly an "error" result). Indeed, using Ax2, Ax3 and (→), we have $\Sigma \vdash \alpha \to \omega \sim \omega \to \omega$ (for any Σ). In the case of Ax7 observe that if an object maps α into γ and α into β then it maps α into the intersection of γ and β, i.e. in $\gamma \wedge \beta$. Using the other axioms we can prove $\Sigma \vdash (\alpha \to \gamma) \wedge (\alpha \to \beta) \sim \alpha \to \gamma \wedge \beta$. Rule (→) represents the antimonotonic–monotonic behaviour of →.

Given an inclusion context Σ its *inclusion theory* $\underline{\Sigma}$ is the set of all inclusions $\alpha \leqslant \beta$ provable from Σ. We will sometimes identify $\underline{\Sigma}$ with Σ.

2.2. *Type assignment rules*

Our basic language is the (type-free) lambda-calculus with a set C of constants. Let Λ_C denote the set of its terms defined by the grammar

$$M ::= C \mid V \mid (MM) \mid \lambda v.M$$

where V is a set of term variables. A formal notion of convertibility $\overset{\beta}{=}$ between terms can be defined as usual [5]. Additional conversion rules for the usual arithmetical and boolean constants can also be considered if necessary.

The formal rules for type assignment include, besides the usual arrow introduction and elimination rules, two rules for intersection introduction and elimination, one rule to handle type inclusion and one for ω. We associate with each basic constant c a set of types τ_c which represents the intended types of c.

A *typing statement* is an expression of the form $M : \alpha$ where α is a type and M a type-free λ-term.

A *typing context* B is a set of statements of the form $\{x_1 : \alpha_1, \ldots x_n : \alpha_n\}$ where each subject x_i is a variable. All variables in a typing context are assumed to be distinct.

A typing judgement is written in the form $\Sigma; B \vdash M : \alpha$ where Σ and B are, respectively, an inclusion and a typing context.

Definition 2.2. Type assignment rules.

(Var) $\Sigma; B \vdash x : \alpha$ if $x : \alpha \in B$

(Const) $\Sigma; B \vdash c : \alpha$ if $\alpha \in \tau_c$

(ω) $\Sigma; B \vdash M : \omega$

(→E) $$\frac{\Sigma; B \vdash M : \alpha \to \beta \qquad \Sigma; B \vdash N : \alpha}{\Sigma; B \vdash (MN) : \beta}$$

(→I) $$\frac{\Sigma; B \cup \{x : \alpha\} \vdash M : \beta}{\Sigma; B \vdash \lambda x.M : \alpha \to \beta}$$

$$(\wedge \mathrm{I}) \qquad \frac{\Sigma; B \vdash M; \alpha \qquad \Sigma; B \vdash M: \beta}{\Sigma; B \vdash M: \alpha \wedge \beta}$$

$$(\wedge \mathrm{E}) \qquad \frac{\Sigma; B \vdash M: \alpha \wedge \beta}{\Sigma; B \vdash M: \alpha \qquad \Sigma; B \vdash M: \beta}$$

$$(\leqslant) \qquad \frac{\Sigma; B \vdash M: \alpha \qquad \Sigma \vdash \alpha \leqslant \beta}{\Sigma; B \vdash M: \beta}$$

The syntactic properties of the system depends heavily on Σ. If we take $\Sigma = \emptyset$, for instance, we can prove (a number of) normalization theorems (see [6]) which are not true for arbitrary inclusion contexts. For more details on the syntactic properties of the system, see [8]. We will often omit Σ or B when they are empty.

2.3. Semantic structures

As remarked before, a suitable notion of semantics for this system can be given interpreting both terms and types in a model of the type-free language. We need a definition as general as possible in order to include most interesting examples. The following definition of model is based essentially on the notion of environment model introduced in [22] (see also [5, Section 5]).

We first define the interpretations of Λ_C in *structures* $\mathscr{M} = \langle D, F, G \rangle$ where D is a set containing at least two elements and F and G are two functions:

$$F: D \to (D \to D), \qquad G: (D \to D) \to D$$

and $(D \to D)$ is some collection of functions from D to itself.

In a structure $\mathscr{M}$ each element of D can be interpreted as a function via F. We say that a function $f: D \to D$ is *representable* in $\mathscr{M}$ if $f = F(d)$ for some $d \in D$. Obviously, by Cantor's theorem, not *all* functions from D to D can be representable but only a subset of them.

Given a structure, an interpretation of λ-terms can be defined in a canonical way, provided that $(D \to D)$ contains enough functions to define the meaning of all λ-terms. A structure is a model if it satisfies an additional condition which ensures that the interpretation of terms is preserved by β-convertibility.

Given a set D, let an *environment* be a function $\rho: V \to D$, assigning values to variables. $\rho[x/v]$ is the environment which is like ρ except for assigning to the variable x the element v of D. Let **Env** denote the set of all environments. Finally, let $\mathscr{B}: \Lambda_C \to D$ be a function that interprets the constants of Λ_C.

Definition 2.3. (i) An interpretation of Λ_C in a structure $\mathscr{M} = \langle D, F, G \rangle$ is a partial function

$$[\![-]\!]^{\mathscr{M}}: \Lambda_C \rightharpoonup \mathbf{Env} \to D$$

such that

(1) $[\![x]\!]^{\mathscr{M}}\rho=\rho(x)$,

(2) $[\![c]\!]^{\mathscr{M}}\rho=\mathscr{B}(c)$,

(3) $[\![MN]\!]^{\mathscr{M}}\rho=F([\![M]\!]^{\mathscr{M}}\rho)([\![N]\!]^{\mathscr{M}}\rho)$,

(4) $[\![\lambda x.M]\!]^{\mathscr{M}}\rho=G(f)$ where $f=\lambda v\in D.[\![M]\!]^{\mathscr{M}}\rho[x/v]$ provided $f\in(D\to D)$.

(ii) A structure $\mathscr{M}=\langle D, F, G\rangle$ is a *premodel* if $[\![-]\!]^{\mathscr{M}}$ is always defined, i.e. if

$$\lambda v\in D.[\![M]\!]^{\mathscr{M}}\rho[x/v]\in(D\to D)$$

for all terms M and environments **Env**.

(iii) A premodel $\mathscr{M}=\langle D, F, G\rangle$ is a *model* of Λ_C if $F\circ G=\mathbf{id}_{(D\to D)}$.

If $F\circ G=\mathbf{id}_{(D\to D)}$, F and G define a retraction of D onto $(D\to D)$. It can be proved that the interpretation of terms in models is preserved under β-conversion (this is not necessarily true for arbitrary structures). If D is a domain one usually takes $(D\to D)=[D\to D]$ (the domain of continuous functions from D to itself). Take, for instance, a domain D satisfying the equation

$$D\cong A+[D\to D],$$

where A is any domain of basic values and $+$ represents disjoint sum. A canonical choice for F and G, in this case, is the following:

$$F(d)(e)=\begin{cases} f(e) & \text{if } d=\mathbf{in}_{[D\to D]}(f) \text{ for some } f\in[D\to D],\\ \bot_D & \text{if } d=\bot_D,\\ ? & \text{otherwise,}\end{cases}$$

$$G=\mathbf{in}_{[D\to D]},$$

where $\mathbf{in}_{[D\to D]}$ is the injection of $[D\to D]$ in D and ? is a distinguished element of A that represents an "error" element. It is well known that $[D\to D]$ is rich enough to contain all the functions arising from the interpretation of terms. Moreover, it is immediate to see that $F\circ G=\mathbf{id}_{[D\to D]}$, so this structure is a model.

2.4. Type systems and type interpretation

Given a model $\mathscr{M}=\langle D, F, G\rangle$, types can be interpreted as subsets of D starting from the interpretation of basic types. This is formalized in the following definition of type system.

The choice of the interpretation of the basic types characterizes to a great extent the kind of properties represented by types. So, for isntance, if a basic type ξ is interpreted as the set of all elments of D which have a defined value (i.e. different from $\bot$ in a topological model), the interpretation of $\xi\to\xi$ will denote all the elements of D which represent total functions on D, i.e. functions which give a defined result whenever applied to a defined value, and so on.

Another very important point is the interpretation of the $\rightarrow$ type constructor: it defines which elements of D we want to have a meaning with respect to the operation of application. For instance, integers are usually considered values which cannot be applied to anything in a meaningful program and, for this purpose, an internal "error" element is usually defined in D in order to map in it all incorrect applications. But this is no longer possible, for instance, in the case of the "undefined" element of a domain ($\perp$) since an undefined term applied to any other value cannot produce a defined value, not even "error". So we can consider $\perp$ as a value for which application can be meaningful.

In the following definition of type system we include a parameter Φ which represents the elements that we want to include in the interpretation of functional types. Obviously, Φ must include at least all the elements of D which are representative of functions.

Definition 2.4. (i) A *type system* is a tuple $\mathscr{T} = \langle \mathscr{M}, K, \mathscr{K}, \Phi \rangle$, where $\mathscr{M} = \langle D, F, G \rangle$ is a structure, K a collection of basic types, $\mathscr{K} : K \rightarrow 2^D$ an interpretation of the basic types K in $\mathscr{M}$ and Φ is a subset of D such that $G \circ F(D) \subseteq \Phi$.

(ii) The *interpretation* of types in a type system $\mathscr{T} = \langle \mathscr{M}, K, \mathscr{K}, \Phi \rangle$ is the function $[\![-]\!]^{\mathscr{T}} : T^K \rightarrow 2^D$ defined by

(1) $[\![\kappa]\!]^{\mathscr{T}} = \mathscr{K}(\kappa)$,

(2) $[\![\omega]\!]^{\mathscr{T}} = D$,

(3) $[\![\sigma \rightarrow \tau]\!]^{\mathscr{T}} = \{d \in \Phi \mid e \in [\![\sigma]\!]^{\mathscr{T}} \Rightarrow F(d)(e) \in [\![\tau]\!]^{\mathscr{T}}\}$,

(4) $[\![\alpha \wedge \beta]\!]^{\mathscr{T}} = [\![\alpha]\!]^{\mathscr{T}} \cap [\![\beta]\!]^{\mathscr{T}}$.

We will write simply $[\![\alpha]\!]$ when $\mathscr{T}$ is understood. We obviously assume that the interpretation of types is consistent with the interpretation of the constants of the language, i.e. $[\![c]\!]^{\mathscr{M}} \in [\![\alpha]\!]^{\mathscr{T}}$ for all $\alpha \in \tau_c$.

Note that $\Phi = [\![\omega \rightarrow \omega]\!]^{\mathscr{T}}$ in all type systems. There are some canonical choices for Φ proposed in the literature. One (called by some authors the "F" semantics of types [17]) is to define $\Phi = F \circ G(D)$ taking exactly all elements which are images of functions via G. Another popular choice, known as the "simple" semantics [17, 23], is to take $\Phi = D$. In this case we consider any application to be meaningful, as in the models of the pure λ-calculus.

Note that in interpreting types according to conditions (1)–(4) of Definition 2.4(ii) it may happen that the interpretation of some intersection type is empty. In this case some types have a trivial interpretation. For instance, if $[\![\alpha]\!]^{\mathscr{T}} = \emptyset$ then $[\![\beta \rightarrow \alpha]\!]^{\mathscr{T}} = \emptyset$ while $[\![\alpha \rightarrow \beta]\!]^{\mathscr{T}} = \Phi$ whichever is the interpretation of β. Obviously, the interpretations of types which can be assigned to closed terms are never empty.

The presence of empty types can be avoided by interpreting types in a collection of subsets of D such that intersection is always nonempty, as the collection of all ideals over D. A more general approach is that of restricting the set of types in such a way as to rule out all types that would have an empty interpretation. In this paper we will take the former approach to avoid the introduction of further technical details. This

will sometime lead us to obtain lattices instead of c.p.o.s, but this is not relevant in the present context. All results of this paper can be formulated in a more general framework in which not all types need to be defined (see [8]).

A particular type system determines an inclusion relation between types.

Definition 2.5. Let $\mathscr{T}$ be a type system.

(i) The inclusion theory determined by $\mathscr{T}$, denoted by Σ^T is defined by

$$\Sigma^T = \{\alpha \leqslant \beta \mid [\![\alpha]\!]^{\mathscr{T}} \subseteq [\![\beta]\!]^{\mathscr{T}}\}.$$

(ii) $\mathscr{T}$ *satisfies* an inclusion context Σ if $\Sigma \subseteq \Sigma^T$. A $\mathscr{T}$-inclusion context (theory) is any inclusion context which is satisfied by $\mathscr{T}$.

It is easy to prove that Σ^T is indeed an inclusion theory, i.e. $\Sigma^T = \underline{\Sigma^T}$. Indeed Σ^T is the largest inclusion theory satisfied by $\mathscr{T}$.

We write $\mathscr{T} \models \alpha \leqslant \beta$ if $\alpha \leqslant \beta \in \Sigma^T$. $\Sigma \models \alpha \leqslant \beta$ if $\mathscr{T} \models \alpha \leqslant \beta$ for all type systems $\mathscr{T}$ satisfying Σ.

Let $\mathscr{M}$ be a model and $\mathscr{T} = \langle \mathscr{M}, K, \mathscr{K}, \Phi \rangle$ a type system over $\mathscr{M}$. If ρ is a term environment (i.e. a mapping $V \to D$), we say that ρ *satisfies* a context $B = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$ if $\rho(x_i) \in [\![\alpha_i]\!]^{\mathscr{T}}$ for all i. $\mathscr{T}; B \models M : \alpha$ means that for all environments γ satisfying B, one has $[\![M]\!]^{\mathscr{M}} \rho \in [\![\alpha]\!]^{\mathscr{T}}$ and $\Sigma; B \models M : \alpha$ that for all type systems $\mathscr{T}$ satisfying Σ one has $\mathscr{T}; B \models M : \alpha$. We will assume implicitly that in all interpretations $\mathscr{M}$ and all type systems $\mathscr{T}$ over $\mathscr{M}$ the interpretation of a constant c of the language is correct with respect to the types in τ_c, i.e. $[\![c]\!]^{\mathscr{M}} \rho \in [\![\alpha]\!]^{\mathscr{T}}$ for all $\alpha \in \tau_c$.

An easy induction on derivation shows that inclusion and type assignment rules are sound with respect to this notion of semantics.

Theorem 2.6. (i) $\Sigma \vdash \alpha \leqslant \beta \Rightarrow \Sigma \models \alpha \leqslant \beta$,

(ii) $\Sigma; B \vdash M : \alpha \Rightarrow \Sigma; B \models M : \alpha$.

Completeness has been proved [13], for pure λ-terms, using term models.

Theorem 2.7. *Let M be a term without constants. Then*

(i) $\Sigma \models \alpha \leqslant \beta \Rightarrow \Sigma \vdash \alpha \leqslant \beta$,

(ii) $\Sigma; B \models M : \alpha \Rightarrow \Sigma; B \vdash M : \alpha$.

3. Filter models as abstract interpretations

Filter models have been introduced in [6]. The basic observation [28] is that, under certain conditions, the interpretation of, intersection types with inclusion can be seen as a basis for a topology whose abstract points determine a domain. The interpretation

of a term in this domain is given by the set of its types. Filter domains can also be seen as a kind of Scott's information systems [29] where both elements and consistent finite sets are represented by types. In this section we start with a brief review of the construction of filter domains. An extensive study of filter domains has been done in [11], to which we refer for more details and proofs.

Let us first define a collection $\mathscr{F}$ (the abstract filters) of subsets of T^K.

Definition 3.1. (i) Let Σ be an inclusion context over a set T^K of types. An *abstract filter* over T^K in Σ is a subset d of T^K such that

(1) $\omega \in d$,

(2) if $\alpha, \beta \in d$ then $\alpha \wedge \beta \in d$,

(3) if $\alpha \in d$ and $\Sigma \vdash \alpha \leqslant \beta$ then $\beta \in d$.

(ii) Let $\mathscr{F}^\Sigma$ ($\mathscr{F}$ for short when K and Σ are understood) be the set of all abstract filters over T^K in Σ. $\mathscr{F}$ is the *filter domain* determined by K and Σ.

Note that the set of all types which can be assigned to a same term (from a given context) satisfies the closure conditions (2) and (3) by rules ($\wedge$I) and ($\leqslant$) and contains ω by (ω).

Given any $A \subseteq T^K$ let $\uparrow^\Sigma\{A\}$ (the filter generated by A) its closure under conditions (1)–(3) of Definition 3.1.

It is just routine to show that $\mathscr{F}^\Sigma$ is a domain.

Lemma 3.2. *$\mathscr{F}^\Sigma$ is a consistently complete, countably based algebraic lattice (a domain) ordered by set inclusion.*

In particular, the bottom element of $\mathscr{F}$ is $\uparrow^\Sigma\{\omega\}$. If $d, e \in \mathscr{F}$, we have $d \sqcap e = d \cap e$ and $d \sqcup e = \uparrow^\Sigma\{d \cup e\}$ (note that, in general, the union of two filters is not a filter). Note that $\Sigma \vdash \alpha \leqslant \beta$ implies that $\uparrow^\Sigma\{\beta\} \subseteq \uparrow^\Sigma\{\sigma\}$ and $\Sigma \vdash \alpha \sim \beta$ implies that $\uparrow^\Sigma\{\beta\} = \uparrow^\Sigma\{\alpha\}$.

Given a filter domain $\mathscr{F}$, there is a "canonical" way to obtain a structure by defining two mappings $F_\to : \mathscr{F} \to [\mathscr{F} \to \mathscr{F}]$ and $G_\to : [\mathscr{F} \to \mathscr{F}] \to \mathscr{F}$ in the following way.

Definition 3.3. Let $\mathscr{F} = \mathscr{F}^\Sigma$ be a filter domain. Define

(i) $F_\to(d)(e) = \uparrow^\Sigma\{\beta \mid \exists \alpha.\alpha \to \beta \in d \text{ and } \alpha \in e\}$,

(ii) $G_\to(f) = \uparrow^\Sigma\{\alpha \to \beta \mid \beta \in f(\uparrow^\Sigma\{\alpha\})\}$.

It is just routine to show that $F_\to$ and $G_\to$ are well defined. In particular, for all $d \in D$, $F(d)$ is a continuous function on $\mathscr{F}$ and $F_\to(d)(e) \in \mathscr{F}$. We will write $d \cdot e$ for $F_\to(d)(e)$.

From the definition of G it turns out that $G(f)$ is defined for all $f \in [\mathscr{F} \to \mathscr{F}]$, so $\langle \mathscr{F}, F_\to, G_\to \rangle$ is always a premodel in the sense of Definition 2.3. Since the choice of $F_\to$ and $G_\to$ is canonical, we will identify $\mathscr{F}$ with the structure $\langle \mathscr{F}, F_\to, G_\to \rangle$.

However, $\mathscr{F}$, in general, is not a model. In particular, we only have $G_{\to} \circ F_{\to} \sqsubseteq \mathbf{id}_{\mathscr{F}}$ and $\mathbf{id}_{[\mathscr{F}\to\mathscr{F}]} \sqsubseteq F_{\to} \circ G_{\to}$ where $\sqsubseteq$ is the extensional order on functions (see [11, Propositions 2.8, 2.10]).

There are interesting cases, however, in which all continuous functions are representable in $\mathscr{F}$ (i.e. in which $F(\mathscr{F}) = [\mathscr{F} \to \mathscr{F}]$). In this case it is easy to see (an explicit proof is given in the appendix) that $F_{\to} \circ G_{\to} = \mathbf{id}_{[\mathscr{F}\to\mathscr{F}]}$ and so F is a model for Λ_C (see also [5, Section 5]). A first example is when $\Sigma = \emptyset$ (for any choice of K). This is the case of the filter model introduced in [6].

To prove that a filter domain $\mathscr{F}^{\Sigma}$ is a model, it is then enough (but not necessary, see [11]) to show that all continuous functions are representable within it. We will give in the next definition a set of sufficient conditions on an inclusion context to guarantee this. The representability of all continuous functions over $\mathscr{F}$ is possible iff all step functions over $\mathscr{F}$ are representable. This is possible only if $\alpha \to \beta \leqslant \gamma \to \delta$ implies $\gamma \leqslant \alpha$ and $\beta \leqslant \delta$ (a complete characterization is given in the appendix). The definition below gives a set of conditions on Σ which assure that the inclusion statements introduced in Σ do not destroy this property. This is achieved by requiring that the basic types in Σ do not interfere with the inclusion relation between functional types. The condition is rather technical but, despite its "ad hoc" nature, it is enough to handle many interesting cases (like, for instance, all the examples of this paper).

Definition 3.4. Let Σ be an inclusion context over a set $K = \{\kappa_1, \ldots, \kappa_n\}$ of basic types. Let Σ^* denote the reflexive and transitive closure of Σ with respect to $\leqslant$.

(i) A basic type κ_i is *plain* in Σ if $\alpha \to \beta \leqslant \kappa_i \notin \Sigma^*$ for any types α, β.

(ii) A basic type κ_i is *functional* in Σ if $\alpha \to \beta \sim \kappa_i \in \Sigma^*$.

(iii) An inclusion context Σ is *safe* if it satisfies the following conditions:

(1) all basic types are either plain or functional,

(2) all the inclusion statements in Σ have one of the following shapes:

- $\alpha \wedge \kappa \leqslant \beta$ or $\kappa \leqslant \beta$ for κ plain and $\alpha, \beta \in T^K$,
- $\kappa \sim \alpha \to \beta$ for $\kappa \in K$ and $\alpha, \beta \in T^K$,
- $\kappa_i \leqslant \kappa_j$ for $\kappa_i, \kappa_j \in K$,
- $\omega \leqslant \omega \to \omega$.

(3) If $\alpha \to \beta \leqslant \gamma \to \delta \in \Sigma^*$ then both $\gamma \leqslant \alpha$ and $\beta \leqslant \delta$ are in Σ^*.

Note that in a safe context we allow an atomic type to be equivalent to a functional type provided this equivalence is consistent with the inclusion between atomic types.

Now we can prove the following theorem.

Theorem 3.5. *If Σ is a safe inclusion context over K then $F_{\to} \circ G_{\to} = \mathbf{id}_{[\mathscr{F}^{\Sigma}\to\mathscr{F}^{\Sigma}]}$ and so $\mathscr{F}^{\Sigma}$ is a model of Λ_C.*

The proof of this theorem is rather technical and it will be given in the appendix.

Example 3.6. Let us consider the set of types $T^{\#} = \mathrm{T}^{\{\mathbf{int}, \mathbf{bool}\}}$ and $\Sigma^{\#}$ as the set

$$\{\mathbf{int} \wedge (\alpha \to \beta) \leqslant \gamma \mid \alpha, \beta, \gamma \in T^{\#}\} \cup \{\mathbf{bool} \wedge (\alpha \to \beta) \leqslant \gamma \mid \alpha, \beta, \gamma \in T^{\#}\}$$

$$\cup \{\mathbf{bool} \wedge \mathbf{int} \leqslant \gamma \mid \gamma \in T^{\#}\} \cup \{\omega \leqslant \omega \to \omega\}.$$

Note that $\Sigma^{\#}$ is the inclusion context satisfied by the interpretation of types as ideals [21] over a domain satisfying $D \cong N_{\perp} \oplus B_{\perp} \oplus [D \to D]$, where $N_{\perp}$ and $B_{\perp}$ are the flat c.p.o.s of integers and booleans values and $\oplus$ denotes coalesced sum.

It is immediate to verify that both **int** and **bool** are plain so that $\Sigma^{\#}$ is safe. The filter domain $\mathscr{F}^{\#} = \mathscr{F}^{\Sigma^{\#}}$ is a domain which satisfies

$$D \cong \{\mathbf{int}\}^{\top}_{\perp} \oplus \{\mathbf{bool}\}^{\top}_{\perp} \oplus [\mathscr{F}^{\#} \to \mathscr{F}^{\#}]^{\top},$$

where $\{\mathbf{int}\}$ and $\{\mathbf{bool}\}$ represent one-element lattices and $\oplus$ represents coalesced sum of lattices. An element of $\mathscr{F}^{\#}$, in fact, can only be of one of the following:

(1) $\uparrow^{\Sigma^{\#}}\{\omega\}$ (the bottom element of $\mathscr{F}^{\#}$, which also represents the function constantly bottom),

(2) $\uparrow^{\Sigma^{\#}}\{\kappa\}$ where κ is **int** or **bool**,

(3) a filter $d_{\to}$ containing only ω and "arrow" types, which is the representative on a function via G,

(4) the whole set $T^{\{\mathbf{int}, \mathbf{bool}\}}$ of types (the top element of $\mathscr{F}^{\#}$).

Each element of $\mathscr{F}^{\#}$ which contains, for instance, $\mathbf{int} \wedge (\alpha \to \beta)$ must contain also **bool** and all other types in $T^{\{\mathbf{int}, \mathbf{bool}\}}$. The fact that there is a one–one correspondence between the elements $d_{\to}$ and the continuous functions from $\mathscr{F}$ to $\mathscr{F}$ is a consequence of the results in [11, 6].

There is a close connection between the interpretation of Λ_C in a filter domain $\mathscr{F}$ and the type assignment system presented in Section 2, namely, roughly speaking, that the interpretation of a term M in $\mathscr{F}$ is given by the set of its types.

Note that we directly interpret the basic language in the "abstract" domain. Obviously, in this interpretation many elements which are distinct in the concrete domain are identified. For instance, in the case of Example 3.6, all integer values are interpreted in the element $\uparrow^{\Sigma^{\#}}\{\mathbf{int}\}$ of $\mathscr{F}^{\#}$.

In particular, if $\xi : V \to \mathscr{F}$ is an environment assigning to variables values in $\mathscr{F}$, define B_{ξ} as the set of contexts built by assigning to each variable x a type (possibly an intersection type) belonging to $\xi(x)$, i.e.

$$B_{\xi} = \{B \mid x : \sigma \in B \Rightarrow \sigma \in \xi(x)\}.$$

Λ_C can be interpreted in any $\mathscr{T}$-filter domain $\mathscr{F}$ by interpreting the basic constants by

$$[\![c]\!]^{\mathscr{F}} = \uparrow^{\Sigma}\{\tau_c\}.$$

We have the following property [11].

Theorem 3.7. *Let $\mathscr{F}=\mathscr{F}^{\Sigma}$ be a filter domain. Then*

$$[\![M]\!]^{\mathscr{F}}\xi=\{\sigma \mid \Sigma; B \vdash M:\sigma \text{ for some } B\in B_{\xi}\}.$$

The interpretation in filter domain then is effectively defined by the type assignment rules introduced in Section 2.

In the following definition we introduce the notion of filter model determined by a type system.

Definition 3.8. Let $\mathscr{M}=\langle D, F, G\rangle$ be a premodel and $\mathscr{T}=\langle \mathscr{M}, K, \mathscr{K}, \Phi\rangle$ a type system over $\mathscr{M}$. Then

(i) a $\mathscr{T}$-filter domain is any filter domain $\mathscr{F}^{\Sigma}$ such that Σ is a $\mathscr{T}$-inclusion context, i.e. an inclusion context satisfied by $\mathscr{T}$,

(ii) $\mathscr{F}^{\mathscr{T}}$ denotes $\mathscr{F}^{\Sigma^{\mathscr{T}}}$.

$\mathscr{F}^{\mathscr{T}}$ is the least $\mathscr{T}$-filter domain in the sense that for any $\mathscr{T}$-filter model $\mathscr{F}$ there is a unique embedding $i:\mathscr{F}\to\mathscr{F}^{\Sigma^{\mathscr{T}}}$.

A $\mathscr{T}$-filter domain $\mathscr{F}$ can then be seen as an "abstract" domain whose elements represent (sets of) properties of the elements of D. There is a natural way of relating D (the domain of concrete interpretation) and $\mathscr{F}$ by a map $\mathbf{abs}_{\mathscr{T}}:D\to\mathscr{F}$ as

$$\mathbf{abs}_{\mathscr{T}}(v)=\{\sigma \mid v\in[\![\sigma]\!]^{\mathscr{K}}\}$$

which maps $v\in D$ to the set of all types that represent properties of v. It is easy to verify that $\mathbf{abs}_{\mathscr{T}}(v)\in\mathscr{F}^{\Sigma}$, i.e. $\mathbf{abs}_{\mathscr{T}}(v)$ is an abstract filter. $\mathbf{abs}_{\mathscr{T}}$ is not, in general, an injective mapping. If we see $\mathbf{abs}_{\mathscr{T}}$ as a relation $\mathbf{ABS}_{\mathscr{T}}\subseteq D\times\mathscr{F}$, its inverse relation $\mathbf{CONC}_{\mathscr{T}}=\mathbf{ABS}_{\mathscr{T}}^{-1}$ represents a kind of "concretization" relation from the abstract domain to the concrete one. We have that the relations $\mathbf{ABS}_{\mathscr{T}}$ and $\mathbf{CONC}_{\mathscr{T}}$ satisfy

$$\mathbf{CONC}_{\mathscr{T}}\circ\mathbf{ABS}_{\mathscr{T}}\supseteq\mathbf{id}_{D},$$

$$\mathbf{ABS}_{\mathscr{T}}\circ\mathbf{CONC}_{\mathscr{T}}=\mathbf{id}_{\mathscr{F}},$$

where $\mathbf{id}_{D}$, $\mathbf{id}_{\mathscr{F}}$ represent, respectively, the identity relations over D and $\mathscr{F}$. These are the usual relations between the abstraction and concretization maps in the theory of abstract interpretation for functional languages [24] (see also [4] for a survey of the basic concepts).

The relation $\mathbf{CONC}_{\mathscr{T}}$ is in general not a function but induces a map $\mathbf{conc}_{\mathscr{T}}:\mathscr{F}\to 2^{D}$ defined as

$$\mathbf{conc}_{\mathscr{T}}(d)=\{v \mid \mathbf{abs}_{\mathscr{T}}(v)=d\},$$

i.e. $\mathbf{conc}_{\mathscr{T}}(D)$ is the set of all elements of D whose abstraction is represented by d.

As a simple consequence of the soundness theorem (Theorem 2.6), we have the following relation between the interpretations $[\![-]\!]^{\mathscr{M}}$, $[\![-]\!]^{\mathscr{F}^{\Sigma}}$, which express the soundness of our abstract interpretation concepts.

Theorem 3.9. *Let $\mathscr{F} = \mathscr{F}^{\Sigma}$ be a $\mathscr{T}$-filter domain where $\mathscr{T} = \langle \mathscr{M}, K, \mathscr{K}, \Phi \rangle$. Then*

$$[\![M]\!]^{\mathscr{F}} \xi \subseteq \mathbf{abs}_{\mathscr{T}}([\![M]\!]^{\mathscr{M}} \rho)$$

for all environments ρ respecting ξ (i.e. such that $\xi(x) \subseteq \mathbf{abs}_T(\rho(x))$ for all variables x).

The meaning of this theorem, which is only a reformulation of the soundness theorem (Theorem 2.6), is that we are able, using the inclusion context determined by a given type system, to deduce only properties which are true M. The abstract semantics, in this sense, is "safe". We remark, however, that $[\![M]\!]^{\mathscr{F}} \xi$ need not contain all the properties (i.e. types) that are true of M in the intended semantics (under the assumptions in ξ); i.e. $[\![M]\!]^{\mathscr{F}} \xi$ is in general only a (proper) subset of $\mathbf{abs}_T([\![M]\!]^{\mathscr{M}} \rho)$. This kind of incompleteness is typical of abstract interpretation. The finitary nature of the type inference rules, in fact, can be an essential limit in the determination of the properties of M. We will see an example of this in the section about strictness analysis, where it is shown that only an infinitary rule can produce all the strictness information about a term.

Remark 3.10. Usually in the results relating abstract and concrete interpretations [4], an intermediate notion of "collecting" the interpretation is introduced. In the collecting interpretation the standard semantics is lifted to operate over sets of values rather than on values themselves. In the present approach, indeed, the relations **ABS** and **CONC** can be seen as mapping between a collection of subsets of D and $\mathscr{F}$ rather than D and $\mathscr{F}$. A theory of collecting interpretation could perhaps be developed, following [24], using a suitable notion of powerdomain. It is not clear to the authors, however, if this would be useful for the investigation of the relations between concrete and abstract semantics.

An interesting case arises when $\mathscr{F}$ is a model. In this case we have, for instance, that the types of a term M (in any typing context) are invariant under formal β-conversion (since in a model the interpretation is invariant under equality). Moreover, we can exploit the structure of the model to prove properties of the assignment system, as it will be done in the next section.

Example 3.11. The following example shows how to build an interpretation that gives information about the sign of a term [4].

We consider a language with only integer values interpreted in a model satisfying $D \cong N_{\perp} \oplus [D \to D]$.

Take the type system $\mathscr{T}^{+} = \langle D, K^{+}, \mathscr{I}, \Phi \rangle$ where K^{+} is the set {**nonpos**, **nonneg**, **int**}, $\mathscr{I}$ the interpretation of basic types which maps **nonpos** in the ideal of all non positive number (including $\perp$) and **nonneg** in the ideal of the nonnegative ones (including $\perp$) and $\Phi = D$.

Let Σ^+ be the inclusion context $\{\mathbf{nonpos} \leqslant \mathbf{int},\ \mathbf{nonneg} \leqslant \mathbf{int}\}$. We associate the appropriate types with any number and with the basic functions on integers like

$$\tau_* = \left\{\begin{array}{l} \mathbf{nonpos} \to \mathbf{nonpos} \to \mathbf{nonneg} \\ \mathbf{nonneg} \to \mathbf{nonneg} \to \mathbf{nonneg} \\ \mathbf{nonpos} \to \mathbf{nonneg} \to \mathbf{nonpos} \\ \mathbf{nonneg} \to \mathbf{nonpos} \to \mathbf{nonpos} \end{array}\right\}, \qquad \tau_+ = \left\{\begin{array}{l} \mathbf{nonpos} \to \mathbf{nonpos} \to \mathbf{nonpos} \\ \mathbf{nonneg} \to \mathbf{nonneg} \to \mathbf{nonneg} \\ \mathbf{nonpos} \to \mathbf{nonneg} \to \mathbf{int} \\ \mathbf{nonneg} \to \mathbf{nonpos} \to \mathbf{int} \end{array}\right\},$$

$$\tau_{\text{unary}^-} = \left\{\begin{array}{l} \mathbf{nonpos} \to \mathbf{nonneg} \\ \mathbf{nonneg} \to \mathbf{nonpos} \\ \mathbf{int} \to \mathbf{int} \end{array}\right\}.$$

It is easy to check that the system infers correct information for terms as in

$$\Sigma^+ \vdash 151515 * (-235) : \mathbf{nonpos}.$$

As another example we have

$$\lambda f.\lambda x.\lambda y.f(fxx)(fyy) : (\mathbf{nonneg} \to \mathbf{nonneg} \to \mathbf{nonneg}) \to \mathbf{nonneg} \to \mathbf{nonneg} \to \mathbf{nonneg}$$

with obvious meanings.

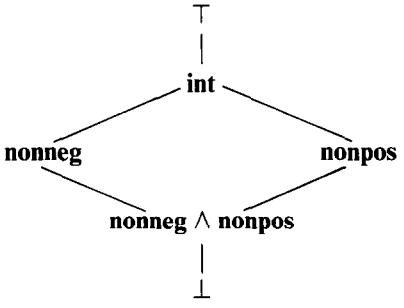
We can now extend Σ^+ by defining the safe context:

$$\Sigma_1^+ = \Sigma^+ \cup \{\mathbf{int} \wedge (\alpha \to \beta) \leqslant \gamma \mid \alpha, \beta, \gamma \in T^{K+}\} \cup \{\omega \leqslant \omega \to \omega\}.$$

Let $\mathscr{F}^+ = \mathscr{F}^{\Sigma_1^+}$. Arguing, as in Example 3.6 we see that $\mathscr{F}^+$ is a domain satisfying

$$\mathscr{F}^+ = A \oplus [\mathscr{F}^+ \to \mathscr{F}^+]^\top,$$

where A is the finite lattice



This is the straightforward generalization to type-free languages of the abstract interpretation of the "rule of sign" for first-order languages.

Note that the filter domain construction is interesting also for inclusion contexts Σ strictly weaker than Σ^T. In fact, what is essentially expected from an abstract interpretation is that it gives safe information, rather than complete. What is relevant

is how natural Σ is and how easily it can be used to extract information in the more interesting cases.

For instance, even $\mathscr{F}^{\Sigma^+}$ could, for practical purposes, be a good inclusion context for the rule of sign. A simple exercise shows that its associated abstract domain satisfies the equation

$$\mathscr{F}^{\Sigma^+} \cong A \times [\mathscr{F}^{\Sigma^+} \to \mathscr{F}^{\Sigma^+}],$$

which, however, seems less interesting as an abstract domain.

Abstract domains can be defined also for type system which are based on models that are not c.p.o.s. The following example, which is taken from [13], introduces a type system and an abstract domain for the study of normalization properties of pure λ-terms.

Example 3.12. Let us consider the pure lambda-calculus Λ and the term model of β-equality $\mathscr{M}(\beta)$ [5]. $\mathscr{M}(\beta)$ can be seen as a model in the sense of Definition 2.3 taking F as formal application and G as lambda abstraction. Let N be the type system $\langle \mathscr{M}(\beta), K^N, \mathscr{K}^N, \mathscr{M}(\beta) \rangle$ where $K^N = \{\mathbf{0}, \mathbf{1}\}$ and $\mathscr{K}^N : \Lambda \to 2^{\mathscr{M}(\beta)}$ is defined by

- $\mathscr{K}^N(\mathbf{0}) = \{[N] \mid N \text{ has normal form}\}$,
- $\mathscr{K}^N(\mathbf{1}) = \{[N] \mid N$ has normal form and, for all $n > 0$, $NM_1, \ldots, M_n$ has normal form whenever $M_1, \ldots, M_n$ have normal form$\}$.

Take Σ' as the inclusion context built by the following six axioms:

$$\Sigma' = \{\mathbf{1} \leqslant \mathbf{0},\ \mathbf{0} \sim \mathbf{1} \to \mathbf{0},\ \mathbf{1} \sim \mathbf{0} \to \mathbf{1},\ \omega \leqslant \omega \to \omega\}.$$

The meaning of these relations is the following:

(1) If a term preserves normalization then it is normalizable, in other words the property represented by $\mathbf{1}$ is stronger than the one represented by $\mathbf{0}$ and so $\mathbf{1} \leqslant \mathbf{0}$;

(2) A normalizable term, applied to a term that preserves normalization, is still normalizable;

(3) If a term preserves normalization, applied to a normalizable term, gives a term that still preserves normalization;

It is easy to verify that Σ' is a N-inclusion context, even if $\Sigma' \subseteq \Sigma^N$ (see [13], where it is shown that $\Sigma' \vdash \alpha \leqslant \beta \Leftrightarrow \Sigma^N \vdash \alpha \leqslant \beta$ holds only if α and β are types without occurrences of $\wedge$).

However, Σ' is powerful enough to prove interesting facts about normalization properties of terms. For instance, we have $\Sigma^N \vdash \lambda x.xx : (\mathbf{0} \to \mathbf{0}) \to \mathbf{0}$, i.e. if we apply $\lambda x.xx$ to a term M mapping any normalizable term in another normalizable term, we have that $((\lambda x.xx)M)$ has still a normal form. This can easily be proved by observing that, in general, for any type α which does not contain occurrences of ω, we have $\Sigma' \vdash \alpha \leqslant \mathbf{0}$. Thus, we also have $\Sigma' \vdash \mathbf{0} \to \mathbf{0} \leqslant \mathbf{0}$. Now assuming $x : \mathbf{0} \to \mathbf{0}$ we can deduce $x : \mathbf{0}$ and, then, $(xx) : \mathbf{0}$.

$\mathscr{F}^{\Sigma'}$ is an (extensional) model satisfying

$$\mathscr{F}^{\Sigma'} \cong [\mathscr{F}^{\Sigma'} \to \mathscr{F}^{\Sigma'}],$$

which is isomorphic to a D_∞-like model [27] built using nonstandard initial projections (see [13] for more details).

4. Strictness analysis

In this section we will study a particular type system suitable for the study of strictness properties of (higher-order) functional languages.

We first define a model for our language, and we will then build a type system for strictness properties for it. We assume that the language has a lazy evaluation strategy.

We assume to have integer and boolean values as basic domains. For technical reasons we will interpret the language in a lattice instead of a c.p.o., using the top element as an error value. The only reason for this is that this makes the proof of the completeness theorems (Theorems 4.4 and 4.10) easier. A suitable domain to interpret our language is then the domain D satisfying

$$D = N_\perp^\top \oplus B_\perp^\top \oplus [D \to D]_\perp^\top,$$

where $N_\perp^\top$ and $B_\perp^\top$ are flat lattices of integers and booleans and $\oplus$ represents coalesced sum of lattices. $[D \to D]_\perp^\top$ is obtained from the $[D \to D]$ by adding new bottom and top elements. The lifting operator on $[D \to D]$ is introduced to model properly lazy evaluation [3]. In this way the interpretation of functional values, also the function constantly undefined $\lambda v.\perp$, is kept distinct from $\perp$. D is made into a model by choosing F and G as

$$F(d)(e) = \begin{cases} f(e) & \text{if } d = \mathbf{in}_{[D\to D]}(f) \text{ for some } f \in [D \to D], \\ \perp_D & \text{if } d = \perp_D, \\ \top & \text{otherwise,} \end{cases}$$

$$G = \mathbf{in}_{[D\to D]}.$$

We take $K_S = \{\mathbf{bot}, \mathbf{int}, \mathbf{bool}\}$ as the set of basic types. Since we want to study strictness properties of functions, we need to assume that $\perp$ is a potential argument of any function. This leads to include $\perp$ in the interpretation of all types. There are however other reasons for assuming this, mainly the fact to be able to include recursively defined functions in the language (see [21, 10]). **bot** will be interpreted as the subset of D containing only $\perp$. So if we know that a value has type **bot** we know everything about it. It corresponds to a maximum information about an element.

Definition 4.1. (i) The type interpretation $\mathscr{K}_S$ is defined by the following:

(1) $\mathscr{K}_S(\mathbf{bot}) = \{\perp\}$,

(2) $\mathscr{K}_S(\mathbf{int}) = \{\mathbf{in}_N(n) \mid n \in N_\perp^\top,\ n \neq \top\} \cup \{\perp\}$,

(3) $\mathscr{K}_S(\mathbf{bool}) = \{\mathbf{in}_B(n) \mid n \in B_\perp^\top,\ n \neq \top\} \cup \{\perp\}$.

(ii) Let $\mathcal{S} = \langle D, K_S, \mathscr{K}_S, \mathbf{in}_D([D \to D]) \cup \{\perp\}\rangle$.

The type system $\boldsymbol{S}$ then gives information about strictness properties of the language. For instance, if $\boldsymbol{S}, B \models M : \mathbf{bot} \to \mathbf{bot}$ then, by Theorem 2.5, M maps $\perp$ in $\perp$ and is so strict in its first argument. Since the language is essentially type-free, however, we must be a little careful in intending what we mean for a function to be strict in one of its arguments. A type-free term can be applied, in general, to an arbitrary number of arguments. For instance, if

$$\boldsymbol{S}, B \models M : \mathbf{bot} \to \omega \to \mathbf{bot}$$

then M is strict in its first argument when applied to two arguments, while M is not strict in its first argument when seen as a function of only one argument, see [19] for a more precise definition of the notion of strictness for type-free terms.

Using types we can also describe more detailed properties of type-free functions involving strictness. For instance, a term having type $(\mathbf{bot} \to \mathbf{bot}) \to \omega \to \mathbf{bot} \to \mathbf{bot}$ is such that whenever applied to a strict function gives a term that is strict in its second argument.

It is easy to verify, by induction on types, that the interpretations of types are downward-closed, directed complete subsets of D (i.e. ideals in D) that do not contain $\top$.

Lemma 4.2. *Let α be an arbitrary type. Then*

(i) $[\![\alpha]\!]^{\mathscr{K}_S}$ *is a downward closed, directed complete subset of D.*

(ii) *If α is different from ω then $\top \notin [\![\alpha]\!]^{\mathscr{K}_S}$.*

In particular, we have that $\perp$ is contained in the interpretation of all types. The inclusion theory Σ^S can be axiomatized in the following way.

Definition 4.3. Let Σ^L be the inclusion context defined as the union of the following sets:

(1) $\{\mathbf{bot} \leqslant \gamma \mid \gamma \in T^{K_S}\}$,

(2) $\{\mathbf{bot} \geqslant \mathbf{int} \wedge \mathbf{bool}\}$,

(3) $\{\mathbf{bot} \geqslant \mathbf{int} \wedge \alpha \to \beta \mid \alpha, \beta \in T^{K_S}\}$,

(4) $\{\mathbf{bot} \geqslant \mathbf{bool} \wedge \alpha \to \beta \mid \alpha, \beta \in T^{K_S}\}$.

Axiom scheme (1) just says that $\perp$ is included in all interpretation of types. The axiom schemes of point (2) mean that the only value which belongs both to a basic type and to a functional type, or to two different basic types is $\perp$. Σ^L is indeed a complete axiomatization of Σ^S.

Theorem 4.4. $\Sigma^L = \Sigma^S$, *i.e.* (1) *and* (2) *completely axiomatize the inclusion theory of* $\boldsymbol{S}$.

The proof of this theorem will be given in the next section.

Remark 4.5. We have chosen to include types **int** and **bool** in $\boldsymbol{S}$ since in this way $\boldsymbol{S}$ can be seen as a natural extension of the basic intersection types system with **int** and **bool**

as basic types. However, **int** and **bool** are not essential when only strictness properties of terms are considered. In this case axiom (1) of Theorem 4.4 is enough to characterize $\Sigma^{S'}$ where S' is the type system defined by taking $K'_S = \{\mathbf{bot}\}$ instead of K_S. Also the completeness theorems (Theorems 4.10 and 4.11) are valid for S'.

This is, for instance, the approach of [19] where just one basic type φ is considered (having the same meaning of **bot**). Kuo and Mishra study the property of this type system in the term model of β-equality interpreting φ as the set of all unsolvable terms. They consider the simple semantics of types with a normal order evaluation and so have the axiom $\omega \leqslant \omega \to \omega$. There is no distinction, in this way, from an unsolvable term M and $\lambda x.M$, both having only type ω or equivalent types. They also have proved in [20] a completeness theorem similar to Theorem 4.4 but their result holds only for types not containing occurrences of the $\wedge$ type operator.

It is easy to see that **bot**, **int** and **bool** are plain in Σ^L. Then by Theorem 3.5 $\mathscr{F}^S$, the filter domain associated with Σ^L, is a model of Λ_C.

Theorem 4.6. *$\mathscr{F}^S$ is a model.*

Indeed $\mathscr{F}^S$ is a domain which satisfies the equation

$$\mathscr{F}^S \cong \{\mathbf{int}\}_\perp^\top \oplus \{\mathbf{bool}\}_\perp^\top \oplus [\mathscr{F}^S \to \mathscr{F}^S]_\perp^\top,$$

where, as in Section 3, $\{\mathbf{int}\}$, $\{\mathbf{bool}\}$ are two one-element lattices. Note the similarities with $\mathscr{F}^*$ defined in Section 3. Indeed, **bot** can be seen as a shorthand for **int** $\wedge$ **bool** and we have $\uparrow^{\Sigma_S} \{\mathbf{bot}\} = \top$, the top element of $\mathscr{F}^S$. Note also the differences in the function space.

Also S', as defined above, does determine a filter domain $\mathscr{F}^{S'}$ which is a model. In particular, $\mathscr{F}^{S'}$ satisfies $\mathscr{F}^{S'} \cong [\mathscr{F}^{S'} \to \mathscr{F}^{S'}]_\perp^\top$.

Since $\mathscr{F}^S$ is a model we have that β-convertible terms have the same types. So two terms are equal in the theory of $\mathscr{F}^S$ iff they have the same provable strictness properties in the sense that the same types can be assigned to them. In particular, all unsolvable terms of order 0, like $(\lambda x.xx)(\lambda x.xx)$, have only type ω and are then interpreted as the bottom element of $\mathscr{F}^\Sigma$.

However, as the previous example shows, the type assignment system is not complete, in the sense that $\Sigma^L, B \models M : \alpha$ does not imply $\Sigma^L, B \vdash M : \alpha$. In fact, we have $\Sigma^L, B \models (\lambda x.xx)(\lambda x.xx) : \mathbf{bot}$ but this cannot be proved by the inference system. Indeed, to prove $\Sigma^L, B \models M : \mathbf{bot}$ is equivalent to prove that the denotation of M is $\perp$ and this is not an r.e. property. We will give a complete system at the end of this section.

The types of the basic constants are naturally induced by their strictness properties. For instance, a complete set of types for $+$ (and of most binary functions on integers) is

$$\tau_+ : \{(\mathbf{int} \to \mathbf{int} \to \mathbf{int}), (\mathbf{int} \to \mathbf{bot} \to \mathbf{bot}), (\mathbf{bot} \to \mathbf{int} \to \mathbf{bot})\}$$

which expresses the fact that $+$ is strict in both its arguments (in S' simply replace **int** by ω).

The if-then-else operator *if* with the usual semantics, has both type $\mathbf{bot}\to\omega\to\omega\to\mathbf{bot}$ (which expresses strictness in the first argument) and type $\mathbf{bool}\to\alpha\to\alpha\to\alpha$ for all types α. Then its set of types (which determines its abstract interpretations) is

$$\tau_{if}=\{\mathbf{bot}\to\omega\to\omega\to\mathbf{bot}\}\cup\{\mathbf{bool}\to\alpha\to\alpha\to\alpha\mid\alpha\in T^{K_S}\}.$$

Type $\mathbf{bot}\to\omega\to\omega\to\mathbf{bot}$ expresses the fact that *if* is strict in its first argument (note that $\mathbf{bot}\to\omega\to\omega\to\mathbf{bot}\leqslant\mathbf{bot}\to\alpha\to\beta\to\mathbf{bot}$ for all α,β). Observe that, using $\leqslant$, we can infer for *if* the types $\mathbf{bool}\to\mathbf{bot}\to\mathbf{bot}\to\mathbf{bot}$, $\mathbf{bool}\to\omega\to\mathbf{bot}\to\omega$ and $\mathbf{bool}\to\mathbf{bot}\to\omega\to\omega$ which express the other usual strictness properties of *if* (i.e. *if* is not strict in its second and third argument but is undefined only if both of them are undefined).

Example 4.7. As an example consider the function

$$G=\lambda f.\lambda x.\lambda y.(fxy)+(fyx)$$

which has type

$$(\mathbf{int}\to\mathbf{bot}\to\mathbf{bot})\wedge(\mathbf{bot}\to\mathbf{int}\to\mathbf{bot})\to\mathbf{bot}\to\mathbf{int}\to\mathbf{bot}$$

and

$$(\mathbf{int}\to\mathbf{bot}\to\mathbf{bot})\wedge(\mathbf{bot}\to\mathbf{int}\to\mathbf{bot})\to\mathbf{int}\to\mathbf{bot}\to\mathbf{bot},$$

which means that G, if applied to a function H which is strict in *both* its arguments (like $+$, for instance), gives a function (GH) which is strict in its second argument (assuming that its first argument is an integer). Similarly, we can prove that GH is strict in its first argument. Note that there is no way to prove, for instance, that $(G+)$ has type $\mathbf{bot}\to\mathbf{int}\to\mathbf{bot}$ without using intersection types.

As for fixed points we have that the standard interpretation in $\mathscr{F}^S$ of the internal operator of the λ-calculus $Y_\lambda=\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ is given by $[\![Y_\lambda]\!]^{\mathscr{F}^S}\xi=fix$, where

$$fix=\uparrow\{(\omega\to\alpha_1)\wedge(\alpha_1\to\alpha_2)\wedge\cdots\wedge(\alpha_n\to\alpha_{n+1})\to\alpha_{n+1}\mid n\geqslant 0,\ \alpha_1,\ldots,\alpha_{n+1}\in T^{K_S}\},$$

i.e. Y_λ has all the types of the form $(\omega\to\alpha_1)\wedge(\alpha_1\to\alpha_2)\wedge\cdots\wedge(\alpha_n\to\alpha_{n+1})\to\alpha_{n+1}$. To see this, observe that $Y_\lambda\overset{\beta}{=}\lambda f.\Delta_f\Delta_f\overset{\beta}{=}\lambda f.f(f(\ldots(f(\Delta_f\Delta_f))))$, where $\Delta_f=\lambda x.f(xx)$, and apply the invariance of types under β-conversion (we can always assign type ω to $\lambda x.f(xx)$).

This type of Y_λ, however, is not very useful in the inference of strictness properties of recursively defined functions. Take, for example, $Y_\lambda F$ where F is defined as

$$F=\lambda f.\lambda x.\lambda y.\mathit{if}(x=0)y(f(x-1)y)$$

(i.e. $Y_\lambda F$ is the recursive function f defined by $fxy=\mathit{if}(x=0)y(f(x-1)y)$. Using the standard interpretation of Y_λ we can prove

$$\Sigma^L\vdash Y_\lambda F:\mathbf{bot}\to\omega\to\mathbf{bot},$$

which means that $Y_\lambda F$ is strict in x but we cannot prove

$$\Sigma^L \vdash Y_\lambda F : \omega \to \mathbf{bot} \to \mathbf{bot},$$

which proves that f is strict also in y.

To fix this, we must consider an explicit fixpoint operator Y_{FIX} defined as a constant in the basic language with the following interpretation in $\mathscr{F}^\Sigma$:

$$[\![Y_{FIX}]\!]^{K_S}\xi = FIX, \quad \text{where } FIX = \uparrow^{\Sigma^L}\{(\alpha\to\alpha)\to\alpha \mid \alpha \in T^{K_S}\}.$$

This means that we assume for Y_{FIX} all types of the form $(\alpha\to\alpha)\to\alpha$, and corresponds to the usual type inference rule for fixed points:

$$(FIX) \quad \frac{\Sigma; B \cup f:\sigma \vdash M:\sigma}{\Sigma; B \vdash \mu f.M : \sigma}$$

Using this rule (i.e. using Y_{FIX} instead of Y_λ as the abstract version of the recursion operator), we can prove $\Sigma^L \vdash Y_{fix}F : \mathbf{int}\to\mathbf{bot}\to\mathbf{bot}$ as we want.

It is easy to check, in fact, that FIX is a fixed point operator on $\mathscr{F}^S$(i.e. $FIX \cdot d = d \cdot (FIX \cdot d)$ for all $d \in \mathscr{F}^S$), but it is *not* the least one. The least fixed point operator over $\mathscr{F}^S$ is *fix*, and we have only $fix \subseteq FIX$.

As an application of Theorem 4.10, however, the use of Y_{FIX} as the abstract fixed point operator is sound, in the sense that we have

$$\mathbf{abs}([\![Y_\lambda]\!]^D\rho) = FIX,$$

i.e. $S \models Y_\lambda : \alpha$ for all $\alpha \in FIX$.

In most cases, intersection is not necessary to analyse functions, especially on recursively defined functions. Let us consider, for example, the function *hof* defined by

$$hof(g, x, y) = (g(hof(K\,0)x(-y\,1)) + (\underline{\text{if}}(=y\,0)x(hof\,I\,3(-y\,1)))$$

taken from [9], where $K = \lambda x.\lambda y.x$ and $I = \lambda x.x$. We have, for instance,

$$\Sigma^L \vdash hof : (\mathbf{int}\to\mathbf{int})\to\mathbf{int}\to\mathbf{bot}\to\mathbf{bot},$$

thus proving that *hof* gives a strict function when applied to any function from integers to integers and to any integer. All other conclusion from the analysis of [9] can be proved within the present approach using FIX.

As we remarked before, type assignment (with Σ^L as inclusion context) is not complete for strictness properties, not even after the explicit introduction of the Y_{FIX} operator. This incompleteness in not due to Σ^L (which gives a complete inclusion theory), but to the type inference rules. Indeed, the type assignment system is r.e.(Σ_1^0) while a complete system must be at least Π_1^0 (in fact $\Sigma, B \models M : \mathbf{bot}$ iff M is unsolvable, which is a Π_1^0 property [5]). A complete inference system for Σ can be obtained by adding two new rules to the ones of Definition 2.2. One of these rules is an infinitary one, thus making the type assignment system Π_1^0.

The new rules (which are these introduced, in a slightly different context, in [10]) can be defined with the help of a syntactical notion of approximation for λ-terms. The following definition is taken from [5].

The set of λ–$\perp$-terms is defined as $\Lambda_{C\cup\{\perp\}}$, where $\perp$ is a new constant.

The set of approximate normal forms is defined as the least set $\mathcal{N}\subseteq\Lambda_{C\cup\{\perp\}}$ containing variables, constants and $\perp$, such that $\lambda x.A$ and $\phi A_1 \ldots A_n \in \mathcal{N}$ whenever $A, A_1, \ldots, A_n \in \mathcal{N}$ and ϕ is either a variable or a constant.

Definition 4.8. (i) For any $M\in\Lambda_C$, the *direct approximant* of M is the term $\Omega(M)$ defined as follows:

(1) $\Omega(x)=x$,

(2) $\Omega(c)=c$,

(3) $\Omega(\lambda x.M)=\lambda x.\Omega(M)$,

(4) $\Omega((\lambda x.M)M_1 \ldots M_n)=\perp$,

(5) $\Omega(\Phi M_1 \ldots M_n)=\Phi\Omega(M_1)\ldots\Phi\Omega(M_n)$ where Φ is a variable or a constant different from $\perp$.

(ii) The set $\mathscr{A}(M)$ of *approximants* of a term M is defined as

$$\mathscr{A}(M)=\{A \mid \exists N \text{ s.t. } M \overset{\beta}{=} N \text{ and } A=\Omega(N)\}.$$

The interpretation of terms of $\Lambda_{C\cup\{\perp\}}$ in D is defined as in Definition 2.3 with $\perp$ interpreted as $\perp_D$. For any term M and environment ρ the set $\{[\![A]\!]\rho \mid A\in\mathscr{A}(M)\}$ is directed and has $[\![M]\!]\rho$ as least upper bound in D (see [5, Section 19.3]).

The new type assignment rules are a formal counterpart of the properties of types stated in Lemma 4.2.

Definition 4.9. The system $\vdash_{\text{APP}}$ is defined by adding to Definition 2.2 the following two rules:

$$(\perp) \qquad \Sigma^L; B \vdash_{\text{APP}} \perp : \sigma \quad \text{for all types } \sigma.$$

$$(\text{APP}) \qquad \frac{\Sigma^L; B \vdash_{\text{APP}} A:\sigma \qquad \forall A\in\mathscr{A}(M)}{\Sigma^L; B \vdash_{\text{APP}} M:\sigma}$$

The system $\vdash_{\text{APP}}$ is sound and complete for the type system $\boldsymbol{S}$. Soundness is proved by a standard transfinite induction. Completeness is proved only for the functional core of Λ_C, consisting of the pure λ-terms.

Theorem 4.10. *Let M be a term of Λ. Then*

$$\Sigma^L; B \vdash_{\text{APP}} M:\alpha \text{ iff } \boldsymbol{S}; B \models M:\alpha.$$

A simple inspection of the proof of this theorem, moreover, shows that rules (APP) and ($\perp$) are not needed if a term has a normal form. So the basic system $\vdash$ turns out to be complete for the terms that have normal form.

Corollary 4.11. *If M is a term of Λ which has a normal form then*

$$\Sigma^L; B \vdash M:\alpha \text{ iff } \boldsymbol{S}; B \models M:\alpha.$$

The approximants of Y_λ, for instance, are the terms of the shape from

$$Y_\lambda \overset{\beta}{=} \lambda f.\Delta_f\Delta_f \overset{\beta}{=} \lambda f.f(\ldots(f\perp)\ldots).$$

A simple analysis shows, for instance, that $\Sigma^L \vdash_{\mathrm{APP}} Y_\lambda:(\alpha\to\alpha)\to\alpha$ for all types α. This then implies

$$\mathbf{abs}_S([\![Y_\lambda]\!]^D)=FIX$$

and this shows that FIX is a correct choice for the abstract interpretation of the recursion operator.

All the above results hold also if we consider $\boldsymbol{S}'$ instead of $\boldsymbol{S}$, i.e. if we consider only **bot** and ω as basic types.

In the presence of constants, the truth of Theorems 4.4 and 4.10 depends on the adequacy of the set of types assigned to the constants. The addition of Y_{FIX}, for instance, preserves completeness. Completeness fails when *if* is added. This is due to the fact that it is not possible, with the basic types in K_S, to give a complete characterization of its behaviour.

We remark, lastly, that if we define

$$[\![M]\!]^*\xi=\{\alpha \mid B_\xi \vdash_{\mathrm{APP}} M:\alpha\},$$

we do not get an interpretation of Λ_C. In fact $[\![-]\!]^*$ is not compositional, in the sense that $[\![MN]\!]^*\xi$ is not equal, in general, to $([\![M]\!]^*\xi)\cdot([\![N]\!]^*\xi)$ (we only have $[\![MN]\!]^*\xi \supseteq ([\![M]\!]^*\xi)\cdot([\![N]\!]^*\xi)$).

5. Completeness proof

In this section we give in some detail the proofs of Theorems 4.4, 4.10 and Corollary 4.11. The proof will be given using the technique introduced in [10]. The main idea is to define, for each type α, a value $t_\alpha \in [\![\alpha]\!]^S$ which completely characterizes the behaviour of the elements of type α.

Recall that application in D is defined by

$$d\cdot e=F(d)(e)=\begin{cases} f(e) & \text{if } d=\mathbf{in}_{D\to D}(f) \text{ for some } f\in[D\to D],\\ \perp_D & \text{if } d=\perp_D,\\ \top & \text{otherwise.}\end{cases}$$

For all $\alpha\in T^{K_S}$ we now define element $t^\alpha\in D$ and $\Delta^\alpha\in[D\to[D\to D]]$ by induction on types.

Definition 5.1. For all $\alpha \in T^{Ks}$ let t^α, Δ^α be defined by

(1) $t^\omega = \top$,

(2) $t^{\mathbf{int}} = \mathbf{in}_D(\mathbf{0})$,

(3) $t^{\mathbf{bool}} = \mathbf{in}_D(\mathbf{true})$,

(4) $t^{\mathbf{bot}} = \bot$,

(5) $\Delta^\omega = \lambda v.\mathbf{id}_D$,

(6) $\Delta^{\mathbf{int}}(v) = \begin{cases} \mathbf{id}_D \text{ if } v = \mathbf{in}_N(n) \text{ for } n \in N \text{ or } v = \bot, \\ \lambda v.\top \text{ otherwise,} \end{cases}$

(7) $\Delta^{\mathbf{bool}}(v) = \begin{cases} \mathbf{id}_D \text{ if } v = \mathbf{in}_N(t) \text{ for } t \in N \text{ or } v = \bot, \\ \lambda v.\top \text{ otherwise,} \end{cases}$

(8) $\Delta^{\mathbf{bot}} = \lambda v.\text{if } v = \bot \text{ then } \mathbf{id}_D \text{ else } \lambda v.\top$,

(9) $\Delta^{\rightarrow}(v) = \begin{cases} \mathbf{id}_D \text{ if } v = \mathbf{in}_{[D \to D]} v' \text{ or } v = \bot, \\ \lambda v.\top \text{ otherwise,} \end{cases}$

(10) $t^{\alpha \to \beta} = G(\lambda v.(\Delta^\alpha(v))(t^\beta))$,

(11) $\Delta^{\alpha \to \beta} = \lambda v.(\Delta^\beta(v \cdot t^\alpha) \circ (\Delta^{\rightarrow}(v))$,

(12) $t^{\alpha \wedge \beta} = t^\alpha \sqcap t^\beta$,

(13) $\Delta^{\alpha \wedge \beta} = \Delta^\alpha \sqcup \Delta^\beta$,

where $\sqcap$, $\sqcup$ denote, respectively, the l.u.b. and g.l.b. in D. Note that $\Delta^\alpha(v)$ is either $\mathbf{id}_D$ or $\lambda v.\top$.

$\Delta^{\mathbf{int}}$, $\Delta^{\mathbf{bool}}$ and $\Delta^{\rightarrow}$ characterize, respectively, N, B and $[D \to D]$ in the sense that they yield $\top$ whenever applied to an element of D which does not belong to N, B or $[D \to D]$. Note that $\Delta^{\mathbf{int}}$, $\Delta^{\mathbf{bool}}$ and $\Delta^{\rightarrow}$ are continuous and internally representable.

To simplify notations in the rest of this section we will identify α and $[\![\alpha]\!]^S$. Moreover, we will write simply $\alpha \leqslant \beta$ for $\Sigma^L \vdash \alpha \leqslant \beta$.

The following lemma can easily be proved by induction on types.

Lemma 5.2. (i) *If* $\alpha \leqslant \beta$ *then* $t^\alpha \sqsubseteq t^\beta$ *and* $\Delta^\beta \sqsubseteq \Delta^\alpha$,

(ii) *if* $\alpha \sim \beta$ *then* $t^\alpha = t^\beta$ *and* $\Delta^\alpha = \Delta^\beta$.

In the following lemmas it is proved that t^α belongs to the interpretation of α, and Δ^α is a function (representable in D) that, in some sense, characterizes type α in the same way as Δ^κ do for κ.

Lemma 5.3. (i) $t^\alpha \in \alpha$,

(ii) $v \in \alpha \Rightarrow \Delta^\alpha(v) = \mathbf{id}_D$.

Proof. (i) and (ii) are proved simultaneously by induction on α. If α is a basic type the proof is immediate. The induction step is by cases on α.

The case $\alpha \wedge \beta$ where either α or β is a basic type and $\alpha \not\sim \beta$ are immediate from the preceding lemma.

Case 1: $\alpha=\beta\to\gamma$.

(i) Let $d\in\beta\cdot t^{\beta\to\gamma}(d)=(\Delta^{\beta}(d)\cdot t^{\gamma})=t^{\gamma}$ by induction hypothesis (i) and (ii).

(ii) Let $d\in\beta\to\gamma$. Note that in this case $\Delta^{\to}(d)=\mathbf{id}_D$ and $d\cdot t^{\beta}\in\gamma$. Then $\Delta^{\beta\to\gamma}(d)=\Delta^{\gamma}(d\cdot t^{\beta})\circ\Delta^{\to}(d)=\Delta^{\gamma}(d\cdot t^{\beta})=\mathbf{id}_D$ by induction hypothesis (i) and (ii).

Case 2: $\alpha=\bigwedge_{i\in I}(\gamma_i\to\delta_i)$.

(i) $t^{\bigwedge_{i\in I}(\gamma_i\to\delta_i)}\sqsubseteq t^{\gamma_i\to\delta_i}\in\gamma_i\to\delta_i$ for all $i\in I$ (by Definition 4.2 and induction hypothesis). Now recall that the interpretation of types is downward-closed.

(ii) Let $v\in\bigwedge_{i\in I}(\gamma_i\to\delta_i)$ then $v\in\gamma_i\to\delta_i$ for all $i\in I$. Then $\Delta^{\bigwedge_{i\in I}(\gamma_i\to\delta_i)}(v)=\bigsqcup\{\Delta^{\gamma_i\to\delta_i}(v)\mid i\in I\}=\mathbf{id}_D$ by induction hypothesis.

In the proof of the following lemma we often use the following facts about type interpretations:

- The interpretation of a type $\alpha\to\beta$ is contained entirely in the component $[D\to D]$ of the domain plus $\bot$ (which belongs to the interpretation of all types),
- t^{κ}, where κ is a basic type and cannot belong to the interpretation of any type of the shape $\alpha\to\beta$.

Lemma 5.4. (i) $t^{\alpha}\in\beta\Rightarrow\alpha\leqslant\beta$,

(ii) $\Delta^{\beta}(t^{\alpha})\neq\lambda v.\top\Rightarrow\alpha\leqslant\beta$ and $\Delta^{\beta}(t^{\alpha})=\mathbf{id}_D$,

(iii) $t^{\beta\to\tau}\cdot t^{\alpha}\neq\top\Rightarrow\alpha\leqslant\beta$ and $t^{\beta\to\tau}\cdot t^{\alpha}=t^{\tau}$.

Proof. By induction on the total number of "arrows" in α, β and τ. If α is a basic type or a type of the form $\bigwedge_{i\in I}(\beta_i'\to\beta_i'')\wedge k$ where k is a basic type different from ω, (i)–(iii) are obvious.

The induction step is by cases on α. Observe that $t^{\beta\to\tau}\cdot t^{\alpha}=\Delta^{\beta}(t^{\alpha})(t^{\tau})$.

Case 1: $\alpha=\gamma\to\delta$.

(i) $t^{\gamma\to\delta}\in\beta$ implies that either $\beta\equiv\omega$ or $\beta=\bigwedge_{i\in I}(\beta_i'\to\beta_i'')$ for some finite set I of indices. The former case is trivial. In the latter case we must have $t^{\gamma\to\delta}\in(\beta_i'\to\beta_i'')$ for all $i\in I$. We omit the subscript i writing $(\beta'\to\beta'')$ for $(\beta_i'\to\beta_i'')$. It is enough to prove that $\gamma\to\delta\leqslant(\beta'\to\beta'')$. Now we have $t^{\gamma\to\delta}\cdot t^{\beta'}=\Delta^{\gamma}(t^{\beta'})(t^{\delta})$ which is equal to ($if\ \Delta^{\gamma}(t^{\beta'})\neq\lambda v.\top\ then\ t^{\delta}\ else\ \top$)$\in\beta''$ (by induction hypothesis). Then either $\beta''\equiv\omega$ or, by induction hypothesis, we must have $\beta'\leqslant\gamma$ and $\delta\leqslant\beta''$. In either case $\gamma\to\delta\leqslant\beta'\to\beta''$.

(ii) Also in this case we have either $\beta\equiv\omega$ or $\beta=\bigwedge_{i\in I}(\beta_i'\to\beta_i'')$ for some finite set I of indices. In fact, if $\beta=\bigwedge_{i\in I}(\beta_i'\to\beta_i'')\wedge k$ for any basic type k, we have immediately $\Delta^{\beta}(t^{\gamma\to\delta})=\lambda v.\top$. If $\beta\equiv\omega$, the case is trivial. Otherwise, arguing as before, it is enough to prove $\gamma\to\delta\leqslant(\beta'\to\beta'')$ where $(\beta'\to\beta'')$ is $(\beta_i'\to\beta_i'')$ for any $i\in I$. Now $\Delta^{\beta'\to\beta''}(t^{\gamma\to\delta})=\Delta^{\beta''}(t^{\gamma\to\delta}\cdot t^{\beta'})\neq\lambda v.\top$ implies either $\beta''\equiv\omega$ or $t^{\gamma\to\delta}\cdot t^{\beta'}\neq\top$. In the former case, $\gamma\to\delta\leqslant\beta'\to\omega$ always holds. In the other one we have, by induction hypothesis (iii), $\beta'\leqslant\gamma$ and $t^{\gamma\to\delta}\cdot t^{\beta}=t^{\delta}$ from which $\delta\leqslant\beta''$ follows by induction hypothesis. The thesis follows from rule $(\to)$ of Definition 2.1.

(iii) $t^{\beta\to\tau}\cdot t^{\alpha}=\Delta^{\beta}(t^{\alpha})(t^{\tau})$. We must have $\Delta^{\beta}(t^{\alpha})\neq\lambda v.\top$, and the proof follows easily by induction hypothesis.

Case 2: $\alpha = \bigwedge_{i\in I}(\gamma_i \to \delta_i)$. For points (i) and (ii), arguing as in the previous case, we can reduce to the case in which $\beta \equiv \beta' \to \beta''$.

(i) $t^{\bigwedge_{i\in I}(\gamma_i\to\delta_i)} \in \beta' \to \beta''$ implies that $e \overset{\text{def}}{=} t^{\bigwedge_{i\in I}(\gamma_i\to\delta_i)} \cdot t^{\beta'} \in \beta''$. The case $\beta'' \equiv \omega$ is trivial. Else we have $e = \bigsqcap_{i\in I}\{\Delta^{\gamma_i}(t^{\beta'})(t^{\delta_i})\} = \bigsqcap\{t^{\delta_i} \mid \beta' \leqslant \gamma_i\}$ (by induction hypothesis) $= t^{\bigwedge_{i\in J}\delta_i}$ where $J = \{i \mid \beta' \leqslant \gamma_i\}$. Moreover, $e = t^{\bigwedge_{i\in J}\delta_i} \in \beta''$ implies, by induction hypothesis, $\bigwedge_{i\in J}\delta_i \leqslant \beta''$. Then we have $\bigwedge_{i\in I}(\gamma_i\to\delta_i) \leqslant \bigwedge_{i\in J}(\gamma_i\to\delta_i) \leqslant \bigwedge_{i\in J}(\gamma_i) \to \bigwedge_{i\in J}(\delta_i) \leqslant \beta' \to \beta''$.

(ii) $\Delta^{\beta'\to\beta''}(t^\alpha) = \Delta^{\beta''}(t^\alpha \cdot t^{\beta'})$ and proof proceeds as before using induction hypothesis.

(iii) $\Delta^\beta(t^\alpha)(t^\tau) \neq \top$ implies $\Delta^\beta(t^\alpha) \neq \lambda v.\top$ and the proof is similar.

Proof of Theorem 4.4. We have to prove that $\alpha \subseteq \beta$ implies $\alpha \leqslant \beta$. Now $\alpha \subseteq \beta$ implies $t^\alpha \in \beta$, which implies $\alpha \leqslant \beta$ by the preceding lemma.

We need one more lemma to prove the completeness theorem (Theorems 4.10 and 4.11). Let B denote a typing context. Let ρ_B be the environment defined by

$$\rho_B(x) = \begin{cases} t^\alpha & \text{if } x:\alpha \in B, \\ \top & \text{otherwise.} \end{cases}$$

Let $\vdash_\perp$ be the system obtained by eliminating rule (APP) from $\vdash_{\text{APP}}$.

Lemma 5.5. *Let A be an approximate normal form without occurrences of constants. Then*

(i) $[\![A]\!]\rho_B \in \sigma$ *implies* $B \vdash_\perp A:\sigma$,

(ii) $\lambda v.\top \neq \Delta^\sigma([\![A]\!]\rho_B)$ *implies* $B \vdash_\perp A:\sigma$.

Proof. By simultaneous induction on A. We have four cases.

Case 1: $A \equiv \perp$.

Both (i) and (ii) are trivial by rule ($\perp$).

Case 2: $A \equiv x$.

(i) We must have $x:\tau \in B$, so $\rho_B(x) = t^\tau$. By Lemma 5.4 $t^\tau \in \sigma$ implies $\tau \leqslant \sigma$. We can then prove $B \vdash_\perp x:\sigma$ using rule ($\leqslant$).

(ii) Similar, using Lemma 5.4(ii).

Case 3: $A \equiv xA_1 \ldots xA_n$ $(n > 0)$.

(i) We must have $x:\tau \in B$ for some type τ.

If $\tau \equiv \bigwedge_{i\in I}(\tau_i' \to \tau_i'') \wedge k$ where k is a basic type and I is any finite set of indices, then $\tau \leqslant \omega \to \cdots \to \omega \to \sigma$ (n occurrences of ω) and, since $B \vdash_\perp A_k:\omega$ $(1 \leqslant k \leqslant n)$, we have $B \vdash_\perp xA_1 \ldots A_n:\sigma$. Else let $\tau \sim \bigwedge_{i\in I}(\tau_i' \to \tau_i'')$ we have

$$\begin{aligned}
[\![xA_1 \ldots A_n]\!]\rho_B &= [\![x]\!]\rho_B \cdot [\![A_1]\!]\rho_B \ldots [\![A_n]\!]\rho_B \\
&= \tau^{\bigwedge_{i\in I}(\tau_i'\to\tau_i'')} \cdot [\![A_1]\!]\rho_B \ldots [\![A_n]\!]\rho_B \\
&= \bigsqcap_{i\in I}\{\Delta^{\tau_i'}([\![A_1]\!]\rho_B)(t^{\tau_i''}) \cdot [\![A_2]\!]\rho_B \ldots [\![A_n]\!]\rho_B \\
&= t^{\bigwedge_{i\in J}(\tau_i'')} \cdot [\![A_2]\!]\rho_B \ldots [\![A_n]\!]\rho_B,
\end{aligned}$$

where $J=\{i \mid (\Delta^{\tau_i}([\![A_1]\!]\rho_B) \neq \lambda v.\top\}$. Then, for all $i \in J$, we have $B \vdash_\perp A_1 : \tau_i'$ by induction hypothesis, i.e. $B \vdash_\perp A_i : \bigwedge_{i\in J} \tau_i$ by $(\wedge I)$.
Moreover, observe that $t^{\bigwedge_{i\in J}(\tau_i'')} \cdot [\![A_2]\!]\rho_B \ldots [\![A_n]\!]\rho_B = [\![yA_2 \ldots A_n]\!]\rho_{B'}$, where $B' = B \cup \{y : \bigwedge_{i\in J}\tau_i''\}$ where y is any variable not free in B. Let $\xi = \bigwedge_{i\in J}\tau_i''$. We then have $[\![yA_2 \ldots A_n]\!]\rho_{B\cup\{y:\xi\}} \in \sigma$ and this, by induction hypothesis (i), implies $B \cup \{y:\xi\} \vdash_\perp (yA_2 \ldots A_n) : \sigma$. Then we get a proof of $B \vdash_\perp (yA_2 \ldots A_n) : \sigma$ by replacing the assumption $y:\xi$ with the deduction of $B \vdash_\perp (xA_1) : \xi$ obtained using $B \vdash_\perp A_1 : \bigwedge_{i\in J}\tau_i'\}$ and the fact that $\bigwedge_{i\in J}(\tau_i' \to \tau_i'') \leqslant \bigwedge_{i\in J}\tau_i' \to \bigwedge_{i\in J}\tau_i''$.

(ii) The argument is similar to the previous one. Now we must have $\lambda v.\top \neq \Delta^\sigma([\![yA_2 \ldots A_n]\!]\rho_{B\cup\{y:\xi\}})$ and we can argue as before by using induction hypothesis.

Case 4: $A \equiv \lambda x.A'$.

This case is simple and is left as an exercise.

Proof of Theorems 4.10 and 4.11. We prove the "if" direction. Assume $S; B \models M : \sigma$. Since types are ideals (i.e. downward-closed), we have $S; B \models A : \sigma$ for all $A \in \mathscr{A}(M)$. By Lemma 5.5 and the fact that ρ_B satisfies B we have $B \vdash_\perp A : \sigma$ and $B \vdash_{\mathrm{APP}} M : \sigma$ using rule APP.

As for the corollary observe that if M is in normal form we need neither rule $(\perp)$ nor APP. Then we can use the fact that types are preserved by β-equality.

Since $\vdash_{\mathrm{APP}}$ on approximate normal forms is a decidable relation, we have that $\vdash_{\mathrm{APP}}$ is Π_1^0. Indeed, it is complete Π_1^0.

We conjecture that, with a slight complication in the proof, these theorems hold also if we consider c.p.o.s instead of lattices.

Acknowledgment

We thank both referees for their useful comments on an earlier draft of this paper.

Appendix

In [11, Proposition 2.13(ii)] the following characterization of filter domains in which all continuous functions are representable, is given.

Theorem A.1. *In a filter domain $\mathscr{F}^\Sigma$ all continuous functions are representable iff $\Sigma \vdash \bigwedge_{i\in I}(\alpha_i \to \beta_i) \leqslant \gamma \to \delta$ (where $\delta \neq \omega$ and I is a finite set of indices) implies that there exists $J \subseteq I$ such that $\Sigma \vdash \gamma \leqslant \bigwedge_{i\in I}\alpha_i$ and $\Sigma \vdash \bigwedge_{i\in I}\beta_i \leqslant \delta$.*

The proof of [12] is given for context Σ containing $\omega \leqslant \omega \to \omega$; however, this axiom is not used in the proof, which holds unchanged assuming the weaker axiom. The basic difference is that assuming $\omega \leqslant \omega \to \omega$ the least "constantly bottom" function is identified with the bottom element of $\mathscr{F}$, while in our case it is not (see also [15]).

Note that if $\alpha \in T^K$ then $\alpha = \bigwedge_{i\in I}(\alpha_i \to \beta_i) \wedge \bigwedge_{i\in J}(\kappa_i)$ where I and J are finite sets of indices.

In order to obtain the relation $F \circ G = \mathbf{id}$ we still need to prove the following theorem.

Theorem A.2. *In a filter domain, if all continuous functions are representable then* $F_{\to} \circ G_{\to} = \mathbf{id}_{[\mathscr{F}^{\Sigma} \to \mathscr{F}^{\Sigma}]}$.

Proof. Let $f \in [\mathscr{F} \to \mathscr{F}]$. Let $d = G_{\to}(f) = \uparrow^{\Sigma}\{\alpha \to \beta \mid \beta \in f(\uparrow^{\Sigma}\alpha)\}$. Now let $g = F_{\to}(d)$ and assume $g \neq f$. Then there exists a point γ such that $g(\uparrow^{\Sigma}\gamma) \neq f(\uparrow^{\Sigma}\gamma)$. Then there must be a type δ such that $\delta \in g(\uparrow^{\Sigma}\gamma)$ but $\delta \notin f(\uparrow^{\Sigma}\gamma)$. Now $\delta \in g(\uparrow^{\Sigma}\gamma) = F_{\to}(d)(\uparrow^{\Sigma}\gamma)$ if $\gamma \to \delta \in d$ for some type γ. But $\gamma \to \delta \in d$ implies $\bigwedge_{i\in I}(\alpha_i \to \beta_i) \leqslant \gamma \to \delta$ for some finite I, where $\beta_i \in f(\uparrow^{\Sigma}\alpha_i)$ for all $i \in I$. By Theorem A.1 then there exist $J \subseteq I$ such that $\gamma \subseteq \bigwedge_{i\in J}\alpha_i$ and $\bigwedge_{i\in J}\beta_i \subseteq \delta$. Now $\uparrow^{\Sigma}\alpha_i \subseteq \uparrow^{\Sigma}\gamma$ and, by monotonicity, $f(\uparrow^{\Sigma}\alpha_i) \subseteq f(\uparrow^{\Sigma}\gamma)$ and, since $\beta_i \in f(\uparrow^{\Sigma}\alpha_i)$, then $\beta_j \in f(\uparrow^{\Sigma}\gamma)$ for all $j \in J$. Clearly, $\bigwedge_{i\in I}(\beta_i) \in f(\uparrow^{\Sigma}\gamma)$ and then $\delta \in f(\uparrow^{\Sigma}\gamma)$ against the hypothesis. □

Now we want to define a notion of normal form of a type.

Definition A.3. Let Σ be a safe context. The *function normal form* of a type $\bigwedge_{i\in I}(\alpha_i \to \beta_i) \wedge \bigwedge_{i\in J}(\kappa_i)$ is obtained by replacing all the occurrences of functional atoms κ_i in $\bigwedge_{i\in J}(\kappa_i)$ by their equivalent $\alpha_i \to \beta_i$. Let *fnf*(α) denote the functional normal form of α. The *fnf* of a type α is clearly equivalent to α.

Lemma A.4. *Let Σ be a safe inclusion context over K. If $\Sigma \vdash \alpha \leqslant \beta$ and* $fnf(\alpha) = \bigwedge_{i\in I}(\alpha_i^1 \to \alpha_i^2)(\alpha_i^2 \neq \omega)$ *then* $fnf(b) = \bigwedge_{j\in J}(\beta_j^1 \to \beta_j^2)$ *and we have that, for all $j \in J$ there exists $I_J \subseteq I$ such that*

(1) $\Sigma \vdash \beta_j^1 \leqslant \bigwedge_{i\in I_J}\alpha_i^1$,

(2) $\Sigma \vdash \bigwedge_{i\in I_J}\alpha_i^2 \leqslant \beta_j^2$.

Proof. The proof is by induction on the derivation of $\Sigma \vdash \alpha \leqslant \beta$. If $\Sigma \vdash \alpha \leqslant \beta$ is an axiom the proof is immediate from the definition of safe context. In the induction step the only nontrivial case is rule (trans):

$$\text{(trans)} \quad \frac{\Sigma \vdash \alpha \leqslant \gamma \quad \Sigma \vdash \gamma \leqslant \beta}{\Sigma \vdash \alpha \leqslant \beta}$$

by the induction hypothesis $fnf(\gamma) = \bigwedge_{\kappa\in K}(\gamma_\kappa^1 \to \gamma_\kappa^2)$ and so, again by the induction hypothesis, $fnf(\beta) = \bigwedge_{j\in J}(\beta_j^1 \to \beta_j^2)$.

We have now to show that conditions (1) and (2) hold. We give the proof for (1) ((2) is very similar).

By the induction hypothesis for all β_j^1 there exists $K_j \subseteq K$ such that $\Sigma \vdash \beta_j^1 \leqslant \bigwedge_{\kappa\in K_J}\gamma_\kappa^1$. Again for induction hypothesis, for each γ_κ^1 there exists $I_\kappa \subseteq I$ such that $\Sigma \vdash \gamma_\kappa^1 \leqslant \bigwedge_{i\in I_\kappa}\alpha_i^1$. Then, by transitivity $\Sigma \vdash \beta_j^1 \leqslant \bigwedge_{i\in H}\alpha_i^1$ where $H = \bigcup_{\kappa\in K_j} I_\kappa$. □

Corollary A.5. *If Σ is a safe inclusion context over K, then all continuous functions are representable in $\mathscr{F}^{\Sigma}$ and $\mathscr{F}^{\Sigma}$ is a model of Λ_C.*

The proof is a trivial application of the preceding lemma.

References

[1] S. Abramsky, Strictness analysis and polymoprhic invariance, in: *Proc. Workshop on Programs as Data Objects*, Lecture Notes in Computer Science, Vol. 217 (Springer, Berlin, 1985) 1–23.

[2] S. Abramsky, Domain theory in logical form, *Ann. Pure Appl. Logic* **51** (1991) 1–77.

[3] S. Abramsky, Research topics in functional programming, in: D. Turner, ed., *The Lazy Lambda-Calculus* (Addison-Wesley, Reading, MA, 1990) 65–116.

[4] S. Abramsky and C. Hankin, An introduction to abstract interpretation, in: S. Abramsky and C. Hankin, eds., *Abstract Interpretation of Declarative Languages* (Ellis Horwood, Chichester, 1987) 9–31.

[5] H. Barendregt, *The Lambda Calculus: its Syntax and Semantics* (North-Holland, Amsterdam, 1984).

[6] H.P. Barendregt, M. Coppo and M. Dezani Ciancaglini, A filter lambda model and the completeness of type assignment, *J. Symbolic Logic* **48** (1983) 931–940.

[7] G. Burn, L. Hankin and S. Abramsky, Strictness analysis of higher order functions, *Sci. Comput. Programming* **7** (1986) 249–278.

[8] F. Cardone and M. Coppo, Two extensions of Curry's type inference system, *Logic and Comput. Sci.* (1990).

[9] C. Clak and S. Peyton Jones, Strictness analysis: a practical approach, in: *Proc. IFIP Symp. on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol. 201 (Springer, Berlin, 1985).

[10] M. Coppo, Completeness of type assignment in continuous lambda models, *Theoret. Comput. Sci.* **29** (1984) 309–324.

[11] M. Coppo, M. Dezani Ciancaglini, F. Honsell and G. Longo, Extended type structures and filter lambda models, in: G. Lolli and G. Longo, eds., *Proc. Logic Colloq. '82* (North-Holland, Amsterdam, 1984).

[12] M. Coppo, M. Dezani Ciancaglini and B. Venneri, Principal type schemes and lambda calculus semantics, in: J. Seldin and R. Hindley, eds., *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism* (Academic Press, New York, 1980) 536–560.

[13] M. Coppo, M. Dezani-Ciancaglini and M. Zacchi, Type theories, normal forms and D_∞ lambda-models. *Inform. and Comput.* **72** (1987) 85–116.

[14] M. Coppo and P. Giannini, A complete type inference algorithm for simple intersection types, in: *Proc. 17th Colloq. on Trees in Algebra and Programming*, Lecture Notes in Computer Science, Vol. 581 (Springer, Berlin, 1992).

[15] M. Dezani-Ciancaglini and I. Margaria, A characterization of f-complete type assignments, *Theoret. Comput. Sci.* **45** (1986) 121–157.

[16] A. Ferrari, Un sistema di tipi per analisi di strictness, Master's Thesis, Università degli Studi di Torino, 1992.

[17] R. Hindley, The completeness theorem for typing λ-terms, *Theoret. Comput. Sci.* **22** (1983) 1–17.

[18] T. Jensen, Strictness analysis in logical form, in: J. Huges, ed., *Proc. Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol. 532 (Springer, Berlin, 1991).

[19] T. Kuo and P. Mishra, On strictness and its analysis, in: *Proc. ACM POPL '87* (1987) 144–155.

[20] T. Kuo and P. Mishra, Inferring strictness properties of the pure λ-calculus: completeness and incompleteness theorems, Technical Report, 1990.

[21] D. MacQueen, G.D. Plotkin and R. Sethi, An ideal model for recursive polymorphic types, *Inform. and Control* **71** (1986) 95–130.

[22] A.R. Meyer, What is a model of the λ-calculus? *Inform. and Control* **52** (1982) 87–122.

[23] J.C. Mitchell, Type inference and type containment, *Inform. and Comput.* **76** (1988) 211–249.
[24] A. Mycroft and F. Nielsen, Strong abstract interpretation using power domains, in: *Proc. ICALP '83*, Lecture Notes in Computer Science, Vol. 154 (Springer, Berlin, 1983).
[25] F. Nielsen, Abstract interpretation of denotational definitions, in: *Proc. STACS* '86, Lecture Notes in Computer Science, Vol. 210 (Springer, Berlin, 1986) 1–20.
[26] S. Ronchi della Rocca, Principal type scheme and unification for intersection type discipline, *Theoret. Comput. Sci.* **59** (1988) 181–209.
[27] D. Scott, Continuous lattices, in: F.W. Lawvere, ed., *Toposes, Algebraic Geometry and Logic*, Lecture Notes in Math., Vol. 274 (Springer, Berlin, 1972) 97–136.
[28] D. Scott, *Lectures on a Mathematical Theory of Computation*, Merton College, Oxford, Michaelmas term, 1981.
[29] D. Scott, Domains for denotational semantics, in: E.M. Schmidt and M. Nielsen, eds., *Proc. 9th Internat. Colloq. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 140 (Springer, Berlin, 1982) 577–613.