Technical Report

UCAM-CL-TR-97
ISSN 1476-2986

Number 97

A study on abstract interpretation and "validating microcode algebraically"

Alan Mycroft

October 1986

ISSN 1476-2986

A study on abstract interpretation and "Validating Microcode algebraically"

Alan Mycroft
Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG
England

This report is a pre-print of a chapter of "Abstract interpretation – theory and practice", Abramsky, S. and Hankin, C. (eds.), Ellis-Horwood, 1987. Any references should refer to the book rather than this report.

Introduction

This chapter attempts to perform two rôles: the first part aims at giving a state-of-the-art introduction to abstract interpretation with as little mathematics as possible – the intention being to bring together ideas from various sources. We leave open the question of what is the 'best' denotational meta-language for abstract interpretation *per se*. The second part gives a fairly tutorial introduction to an *application* of abstract interpretation based on the relational style of [16] by considering an embedding of Foster's work on 'validating microcode algebraically' into the the framework of abstract interpretation. It is intended that the parts may be read independently. No claim of novel techniques is made, but rather this work attempts to make the literature more understandable to a wider audience. I will draw mainly on the works of Nielson and myself but the dependence on Foster's [6] and the Cousots' [4] contributions should also be clear.

This work is seen as complementary to Foster's and provides an alternative (and semantically based) way of understanding its theory. In particular we will not attempt to achieve the pragmatic success of Foster's work. One might compare his work with discovering regular expressions corresponding to the regular languages we work with. Another parallel is that between Milner's algorithm for type checking using type expressions containing variables and unification with the development in [16] using sets of monotypes and abstract interpretation.

1 Abstract interpretation

Let me start this first part by making several statements on abstract interpretation in the form of 'slogans'. These will then be developed in more detail.

- The process of abstract interpretation is merely that of proving the equivalence of two denotational semantic definitions of a given language with respect to a correctness relation.

- Reynolds [23] developed a technique (using directed complete relations) to show a special case of this in 1974. Similar techniques are used by Plotkin [22] (logical relations) and also discussed in Stoy [26] (inclusive predicates).

- Mosses [13,14] introduces the notion of "abstract semantic algebra" which can be used to describe the semantics of programming languages *via* interpretations in the algebraic manner. The interpretations give meaning to symbols in the conventional denotational meta-language of the typed λ-calculus.

- (F.) Nielson [18] and in an application with (H. R.) Nielson [21] demonstrates that a rather more complicated two-level meta-language can be pragmatically useful in separating 'compile-time' and 'run-time' notions.

- In a sense, one can see the following sequence of works (in chronological order) providing the principal theory on which this exposition is based: Cousot and Cousot [4], Donzeau-Gouge [5], Mycroft [15], Nielson [18], Burn, Hankin and Abramsky [3] and Mycroft and Jones [16].

The above choice of works are intended to reflect my view on principal contributions to the ***theory*** of abstract interpretion. There are also many other excellent works on its applications which I do not touch on here.

The view we shall take in this paper — the details follow — is that the process of abstract interpretation is the setting up of standard and non-standard semantics of a given language. The framework we consider here is when they are both given denotationally. This means that they may both be given as interpretations in the algebraic sense (of domains and values for uninterpreted symbols) for a single denotational definition of the language into an uninterpreted meta-language. The differing domain(s) are related by ***abstraction relation(s)***. Moreover, these abstraction relations are required to be preserved by operations used in the meta-language and so we may deduce that the relevant abstraction relation holds between standard and non-standard meanings of any given program [1].

One can see abstract interpretation as thus including certain forms of equivalence proof between denotational semantics. On the other hand, there is a conventional view that the abstracting interpretation 'ought' to be computable. Abstract interpretation is not concerned at first sight with studies such as that in chapter 13 of Stoy [26] in which denotational and operational definitions for a given language were shown to be equivalent.

In section 3 we will set up a denotational semantics for microcode programs. In the meanwhile we will, for simplicity and concreteness, discuss ideas in a framework

[1]This restriction would seem to be the way in which proving the equivalence of two denotational semantics is in general richer (harder) than abstract interpretation as it is generally understood. For example, in proving semantic equivalence, we may merely desire the weaker condition that a given equivalence only holds in certain contexts – see later for more detail.

of a simple language of commands and expressions given by

$$Cmd\colon c \;\; ::= \;\; x := e \;\mid\; c; c' \;\mid\; \mathbf{if}\, e\, \mathbf{then}\, c\, \mathbf{else}\, c' \;\mid\; \mathbf{while}\, e\, \mathbf{do}\, c$$

where x is assumed to range over a set *Var* of variables and e over a set *Exp* of expressions.

We will now consider various possible semantics for such a language. The only requirement is the "denotational assumption" [26] — that there be a domain M_{Cmd} of meanings for commands and that there be a function $C : Cmd \to M_{Cmd}$ which gives the meaning of commands and which is defined in terms of the meanings of their constituent parts. Since *Cmd* is generally infinite, we view C as giving a 'translation' of *Cmd* into meta-language[2] terms whose type requires them to yield values in M_{Cmd}. This is often described by saying that C is 'homomorphic'[3]. It is more precise to state that C is to be defined over *Cmd* by structural induction. Similar requirements would be placed on expressions if they were recursively defined.

Accordingly the only possible form of $C[\![c;c']\!]$ is $C[\![c;c']\!] = E[C[\![c]\!], C[\![c']\!]]$ where $E[m, m']$ represents a (meta-language) expression which (possibly) contains m and m'.

Of course, since the meta-language is the λ-calculus this merely rewrites to a requirement that there be a function $seq : M_{Cmd} \times M_{Cmd} \to M_{Cmd}$. Each clause can thus be written in the form (*e.g.*) $C[\![c;c']\!] = seq(C[\![c]\!], C[\![c']\!])$

Due to our interest in giving several alternative semantics for a given program we will give a *schematic* semantics in which several domains and functions are left uninterpreted. Formalisation of this idea can be traced to Nielson's thesis [18] although the identification of interpreted and uninterpreted symbols with "compile-time" and "run-time" notions only appears as a statement in his later work [19] on two-level meta-languages.

The above arguments essentially re-create Mosses' earlier and more theoretic work on "abstract semantic algebras" ([14] is the definitive reference but [13] sets out the points in a somewhat simpler manner). The difference is however that Mosses uses this structure to then modularise semantic definitions (see the discussion in the "Aside" below) in order to investigate which interpretations of the combinators such as *seq* occur naturally. We are interested in varying the interpretations of combinators and investigating their relationships.

[2]Here, as usual, the typed λ-calculus.

[3]This is a terribly weak form of the word since syntactic structures are generally assumed to be a free algebra (*i.e.* programs are syntactically equal if and only if they have the same parse trees).

To put this on a little more formal basis we state that thus *any* denotational semantics for *Cmd* can be written:

$$\left.\begin{array}{l} M_{Cmd} \\ M_{Exp} \\ M_{Var} \end{array}\right\}\text{uninterpreted spaces of denotations}$$

$$\left.\begin{array}{l} ass : M_{Var} \times M_{Exp} \rightarrow M_{Cmd} \\ seq : M_{Cmd} \times M_{Cmd} \rightarrow M_{Cmd} \\ cond : M_{Exp} \times M_{Cmd} \times M_{Cmd} \rightarrow M_{Cmd} \\ rpt : M_{Exp} \times M_{Cmd} \rightarrow M_{Cmd} \end{array}\right\}\begin{array}{l}\text{uninterpreted function} \\ \text{symbols } (\textit{combinators})\end{array}$$

$$\left.\begin{array}{l} \mathcal{C} : Cmd \rightarrow M_{Cmd} \\ \mathcal{E} : Exp \rightarrow M_{Exp} \\ \mathcal{V} : Var \rightarrow M_{Var} \end{array}\right\}\text{semantic scheme functionalities}$$

$$\left.\begin{array}{l} \mathcal{C}[\![x := e]\!] = ass(\mathcal{V}[\![x]\!], \mathcal{E}[\![e]\!]) \\ \mathcal{C}[\![c; c']\!] = seq(\mathcal{C}[\![c]\!], \mathcal{C}[\![c']\!]) \\ \mathcal{C}[\![\textbf{if } e \textbf{ then } c \textbf{ else } c']\!] = cond(\mathcal{E}[\![e]\!], \mathcal{C}[\![c]\!], \mathcal{C}[\![c']\!]) \\ \mathcal{C}[\![\textbf{while } e \textbf{ do } c]\!] = rpt(\mathcal{E}[\![e]\!], \mathcal{C}[\![c]\!]) \end{array}\right\}\text{semantic equation schemes}$$

Note that such semantics are really very weak — for more discussion see [7] where the options of presenting detail in the algebra or in the semantics are explored. However, for our purposes of re-interpreting meta-language symbols the choice is ideal.

Now, we define an *interpretation*, I say, to be a tuple of spaces[4] interpreting $M_{Cmd}, M_{Exp}, M_{Var}$ and a tuple of functions $ass, seq, cond, rpt$ with the above functionalities (together with possibly other functions determining the meaning of expressions and variables).

Such an interpretation induces a meaning to the semantic functions (and thence a semantics) in the traditional algebraic manner. We refer to the semantic functions induced from the semantic functions schemes $\mathcal{C}, \mathcal{E}, \mathcal{V}$ and interpretation I as $\mathcal{C}_I, \mathcal{E}_I, \mathcal{V}_I$.

As an example, letting *Val* be a set of values including a value 0 interpreted as 'false' and $Env = Var \rightarrow Val$ be the set of states, the standard (direct) semantics can be given by an interpretation

$$Std = \left(\begin{array}{c} Env \rightarrow Env_\perp, \\ Env \rightarrow Val, \\ Var; \\ \lambda(v, m_e).\lambda\rho.\rho[m_e/v], \\ \lambda(m_c, m_{c'}).m_{c'} \circ m_c, \\ \lambda(m_e, m_c, m_{c'}).(\lambda\rho.m_e(\rho) \neq 0 \rightarrow m_c(\rho), m_{c'}(\rho)), \\ \lambda(m_e, m_c).fix\lambda\Phi.(\lambda\rho.m_e(\rho) \neq 0 \rightarrow \Phi(m_c(\rho)), \rho), \\ \ldots \end{array}\right) \leftrightarrow \left(\begin{array}{c} M_{Cmd}, \\ M_{Exp}, \\ M_{Var}; \\ ass, \\ seq, \\ cond, \\ rpt, \\ \ldots \end{array}\right)$$

[4]Here domains, although we use the unbiased word 'space' to allow for interpretation over other mathematical structures.

Note that to avoid complicating the equations certain coercions have been omitted such as that from $Env \to Env_\perp$ to $Env_\perp \to Env_\perp$ in the interpretation of *seq*.

For the relation with Nielson's two-level meta-language see section 2.1.

Note on "homomorphic"

Earlier, the use of the word "homomorphic" to describe semantic equations was criticised on the grounds that it merely meant "inductive". On the other hand, we may wish to assert (following ideas such as transformational semantics [2]) that in all contexts the programs

$$[\![\textbf{while } e \textbf{ do c}]\!] \quad \text{and} \quad [\![\textbf{if } e \textbf{ then } (c; \textbf{while } e \textbf{ do } c) \textbf{ else skip}]\!]$$

are equivalent. Then this would translate to a *real* homomorphic constraint on interpretations that

$$rpt(m_e, m_c) = cond(m_e, seq(m_c, rpt(m_e, m_c)), skip)$$

provided *skip* gave the interpretation of **skip** which could itself be further restricted to be the identity. Such restrictions naturally encode 'traditional' program construct equivalences.

Aside

The reader familiar with denotational semantics [8,26] will recognise the concepts of 'direct' and 'continuation' styles of semantics as special cases of the above.

- Direct semantics has state transformers (*i.e.* functions from $Env \to Env$) as the meanings of commands.
- Continuation semantics has continuation transformers (*i.e.* functions from $(Env \to Ans) \to (Env \to Ans)$ for some space *Ans* of answers, probably *Env* here) as the meanings of commands.

For *Cmd* these could lead to semantics equations (for a *store* semantics) of the respective forms:

$$\begin{array}{ll} C[\![c;c']\!]\rho = C[\![c']\!](C[\![c]\!]\rho) & \text{(direct)} \\ C[\![c;c']\!]\kappa\rho = C[\![c]\!](C[\![c']\!]\kappa)\rho & \text{(continuation).} \end{array}$$

See Gordon [8] or Stoy [26] for more detail.

Mosses rightly decries the convention that all semantic domains are 'concretely' given in terms of domain constructors $+, \times, \to$ from given 'base' domains. This leads to the proliferation of arguments to semantic equations as we see in the above which make them hard to understand for more complicated examples. In Mosses' works referenced above the argument is that abstract data types, instead of these concrete data types, greatly facilitate modularisation of large semantic descriptions — this directly parallels the corresponding argument for the use of abstract data types for programming large systems. Note that we are considering the abstract data types to be ultimately given in terms of certain concrete data types whose exact description is irrelevant to us (see Reynold's parable [24]) rather than an axiomatic specification. This ensures that logical relation ideas can be used straight-forwardly.

Mosses' theory also corresponds to a view that there is a pyramid-shaped spectrum of possible denotational definitions of a language — at its apex are possible interpretations of our semantic scheme given above and lower points correspond to those semantic forms where certain choices of domains have already been made. One example would be the instantiation that meanings of commands are functions accepting function arguments and yielding function results.

2 Abstraction of interpretations

Now let us turn to the *raison d'être* of factoring our semantics into schematic semantics and interpretations — the fact that we wish to relate two different semantics for the same language.

Suppose we have two different denotational semantics for our programming language *Cmd* — one 'real' and one 'abstract' whose results specify properties about the 'real' computation. We can see both of these as being given by interpretations (in the sense above) of the semantic archetype above. Let us suppose that these are called $\mathcal{C}_R[\![\cdot]\!]$ and $\mathcal{C}_A[\![\cdot]\!]$.

Now what we wish to show is that some form of correctness property holds *i.e.* $(\forall c)\mathcal{C}_R[\![c]\!] \rhd \mathcal{C}_A[\![c]\!]$ for some correctness relation $\rhd$. One way of proving that such a relation holds is to construct relations at appropriate meta-language types and show (by structural induction on meta-language terms and thereby for any given form of c) that this holds.

This idea can be traced to Plotkin, Reynolds and Milne and Strachey and leads to the notion of a *logical relation.* (The term *inclusive predicate* is often used for a very similar idea in spite of the fact that this is at some variance with the definition in Milne and Strachey [11].) To be more precise, suppose we label the relation above as $\rhd_{Cmd}$ to indicate that it relates the 'R' interpretation of M_{Cmd} to its 'A' interpretation. We will write these domains as R_{Cmd} and A_{Cmd} respectively, thus $\rhd_{Cmd} \in \mathcal{R}(R_{Cmd}, A_{Cmd})$ – the space of relations. Now let σ, τ be meta-language types formed from $M_{Cmd}, \times, \rightarrow$ and let R_σ, A_σ be the spaces obtained as the interpretation of the type σ when M_{Cmd} is interpreted as R_{Cmd}, A_{Cmd} respectively and $\times, \rightarrow$ are conventionally interpreted as product and function space respectively.

Suppose now that we have a type-indexed family $\rhd_\sigma \in \mathcal{R}(R_\sigma, A_\sigma)$ of relations.

We define ($\rhd_\sigma$) to be a *logical relation* [22] if and only if for all types σ, τ we have

$$\begin{array}{l}(\forall r \in R_\sigma, r' \in R_\tau)(\forall a \in A_\sigma, a' \in A_\tau) \\ \qquad (r \rhd_\sigma a \wedge r' \rhd_\tau a') \Leftrightarrow (r, r') \rhd_{\sigma\times\tau} (a, a') \\ (\forall r \in R_\sigma, \phi \in R_{\sigma\to\tau})(\forall a \in A_\sigma, \psi \in A_{\sigma\to\tau}) \\ \qquad\quad (r \rhd_\sigma a \Rightarrow \phi r \rhd_\tau \psi a) \Leftrightarrow \phi \rhd_{\sigma\to\tau} \psi\end{array}$$

This definition generalises trivially to allow more than one base type (*Cmd* above).

Although the above is a *definition* of when an indexed relation is logical it is clear that is can also be used to give an inductive definition of a logical relation from a specification of its effect at base types. In the above this means that, for all σ, $\rhd_\sigma$ is determined from a given $\rhd_{Cmd}$.

We now return to the problem of proving $(\forall c) \mathcal{C}_R[\![c]\!] \rhd_{Cmd} \mathcal{C}_A[\![c]\!]$. Let R_{seq} and A_{seq} give the interpretations of *seq* in R and A respectively. Now, showing that

$$\begin{array}{l}(\forall r, r' \in R_{Cmd})(\forall a, a' \in A_{Cmd}) \\ \qquad (r \rhd_{Cmd} a \wedge r' \rhd_{Cmd} a') \Rightarrow R_{seq}(r, r') \rhd_{Cmd} A_{seq}(a, a')\end{array} \tag{1}$$

would establish one case of a proof by structural induction (over meta-language terms).

Now, if $\rhd_\sigma$ is a logical relation we could simply require that

$$R_{seq} \rhd_{Cmd\times Cmd\to Cmd} A_{seq}$$

which is is equivalent to

$$\begin{array}{l}(\forall r, r' \in R_{Cmd})(\forall a, a' \in A_{Cmd}) \\ \qquad (r \rhd_{Cmd} a \wedge r' \rhd_{Cmd} a') \Leftrightarrow R_{seq}(r, r') \rhd_{Cmd} A_{seq}(a, a')\end{array}$$

which is indeed a stronger form of equation 1 above.

The above explains how to relate corresponding *constants* of the two differing interpretations R and A. We now say that the interpretation A *abstracts* R, written $R \rhd A$ or more formally $R \rhd_{Cmd,Exp,\ldots} A$, if we have a relation $\rhd_\iota \in \mathcal{R}(R_\iota, A_\iota)$ for each base type $\iota \in \{Cmd, Exp, \ldots\}$ of the uninterpreted type symbols and $R_f \rhd_\tau A_f$ for each uninterpreted constant f (presumed to be of type τ).

Mycroft and Jones [16] discuss these ideas in greater depth (including the possible definitions for +) and also showed how this form of argument could be used to verify a form of the Hindley/Milner polmorphic type system. However, the idea of relating two interpretations goes back to the Cousot' original work [4] although they used adjoined functions instead of relations. Similar ideas were introduced by Nielson [18] in his thesis (the sim_{ct} relation) which also discusses their definition on certain forms of recursive types; however, this source is rather inaccessible due to that fact that sim_{ct} is rather tied up with the definition of the collecting semantics.

In my opinion this style of proof seems to characterise much of abstract interpretation. On the other hand it seems equally clear that not all proofs of equivalence of denotational definitions can be performed by such techniques (consider proofs involving 'context'). However, I would shy away from attempting to characterise abstract interpretation in this manner — even though it seems natural and would distinguish it from general denotational equivalence proof theory. Another aspect is that Nielson's treatment of abstract interpretation with *TML* essentially provides the ability to insist that various constant symbols may only take on restricted meanings as interpretations vary — see the next section.

I would like at this point to put the, possibly controversial, view that abstract interpretation is essentially a special case of denotational equivalence proof theory. In particular we assert that "abstract interpretion" is nothing more than proving the equivalence (relative to some relation) of two different denotational semantics of a given programming language and that Reynolds [23] as long ago as 1974 described a special case. On the other hand many applications of abstract interpretation do not feel very much like classical denotational equivalence theory. I believe that this effect is due to the fact that previous work on semantic equivalence proof has always been based on an "equality-like" relation setting up a mutual simulation of the two denotational definitions. Abstract interpretation applications have by and large been concerned with a "partial order-like" relation due to the fact that we are interested mainly in proving 'safe' properties of programs using a decidable interpretation.

This is, at first sight, a rather depressing thought to add to a book devoted to abstract interpretion. Solace may be gained from the view that abstract interpretation devotees now understand proof of equivalence of denotational definitions!

2.1 Comparison with Nielson's two-level meta-language

In this section we compare the special case development above with Nielson's *TML*. Nielson introduced in his thesis [18] the idea of a two-level meta-language (*TML*) for denotational definitions. One of the most readable introductions I have seen is given in [21] but note that Nielson has a contribution in this volume. To enable direct comparison and also for the sake of completeness we accordingly give a short introduction to what I feel are the essential elements of *TML*.

The idea is that we give meanings to some of the (meta-language) symbols occurring in the denotational translation of a program once and for all, but leave others uninterpreted to be specified by interpretations.

The above works introduce a meta-language *TML* which is a form of the typed λ-calculus with a two-level structure of types. These are called ct and rt and can be seen as representing notions of "compile-time" and "run-time" in that they formalise the somewhat intuitive notions of Tennent[27]. The above references make this more precise. Letting A and $\underline{B}$ be sets of type symbols and X and $\underline{Y}$ be sets of type variables then the type structure of *TML* can be written

$$
\begin{aligned}
ct &::= A \mid X \mid ct_1 + \cdots + ct_n \mid ct_1 \times \cdots \times ct_n \mid ct_1 \to ct_2 \mid recX.ct \mid rt \\
rt &::= \underline{B} \mid \underline{Y} \mid rt_1 \underline{+} \cdots \underline{+} rt_n \mid rt_1 \underline{\times} \cdots \underline{\times} rt_n \mid rt_1 \underline{\to} rt_2 \mid \underline{recY}.rt
\end{aligned}
$$

where underscoring is used to distinguish type symbols in *rt*. The idea is that the elements, A_i of A are interpreted by domains corresponding to conventional base types (*e.g.* the integers) and $+, \times, \rightarrow$ and *rec* $X.ct$ represent the standard domain constructions of sum, product, function space and recursive domain definition. These are fixed over all interpretations. On the other hand, the $\underline{B_i}$ are interpreted individually for each interpretation as are the "run-time" constructors $\underline{+}, \underline{\times}, \underline{\rightarrow}$ and $\underline{rec\,Y}.ct$. The link between the two levels is made by $ct ::= rt$.

Together with these types come (two levels of) appropriate operations on them (essentially introduction and elimination rules for each type) together with constants of given types. The meanings of constants are allowed to vary between interpretations (but see the discussion of the term structure of TML_b below).

Due to technical problems with power domains required for the notion of collecting interpretation Nielson's thesis actually considers a sub-language TML_s with the clause $rt ::= rt_1 \underline{\rightarrow} rt_2$ omitted and with some restrictions on the types of constants.

This is quite a rich language (perhaps this accounts for the fact that such a powerful framework has not been more taken to heart by the abstract interpretation community[5]) and somewhat redundant. One aspect of its redundancy is that the requirement for interpretations to specify the lower level domain constructors may be simply removed by adding suitable extra elements of $\underline{B_i}$. For example, the type $(\underline{B}_1 \times \underline{B}_2) \rightarrow (\underline{B}_1 \underline{\times} \underline{B}_2)$ may be replaced by the type $(\underline{B}_1 \times \underline{B}_2) \rightarrow \underline{B}_3$ and the interpretation now specifies $\underline{B}_1, \underline{B}_2, \underline{B}_3$ instead of $\underline{B}_1, \underline{B}_2, \underline{\times}$. This is a simple expansion of a higher-order (in the sense that type constructors as well as types are allowed to vary) type system into a first-order one which can simplify the exposition of *TML*[6]. By way of mitigating this criticism of Nielson let us recall that his aims were to develop abstract interpretation for other than 'toy' languages and such higher-order notions enhance the ease of expressions. This is presumably of great use in Nielson and Nielson's PSI project [21] which involves the development of a powerful program analysis and translation system based on *TML*.

Nielson himself [20] considers a different sub-language of *TML* in a paper aimed at showing how Mycroft and Jones' [16] work on abstraction *relations* and Abramsky's development therefrom [1] could be subsumed in a relatively small development from *TML*. This sub-language is called TML_b and essentially contracts the *rt* type structure to a single point. Its type structure is given by

$$ct ::= A \mid ct_1 + ct_2 \mid ct_1 \times ct_2 \mid ct_1 \rightarrow ct_2 \mid \underline{B}$$

In some sense this indicates some problem in our understanding of the real essence

[5]This is also possibly due (*mea culpa*) to the fact that at the time of its introduction the mixed strands of category theory, logical relations, collecting interpretation via power domains and two-level meta-language were rather intertwined and not easily seen apart.

[6]In particular, I think that the higher order type system of Nielson's meta-language (one parameterised on operations on spaces (*e.g.* the cartesian product) as well as on spaces themselves) is an elegant extension to the base theory which unfortunately seems to clutter the theoretical development.

of "compile-time" and "run-time" notions in that it is not intuitively clear that it should be *rt* to collapse. Investigation as to why this is so seems to be required. In order to clarify the following comparision it is convenient to give Nielson's TML_b meta-language terms here:

$$e ::= \begin{cases} f_i & \text{constants of type } ct \\ x_i & \text{variables of type } ct \\ in_i\, e \mid is_i\, e \mid out_i\, e & + \text{ introduction/elimination} \\ (e, e') \mid e \downarrow i & \times \text{ introduction/elimination} \\ \lambda x : ct.e \mid e(e') & \rightarrow \text{ introduction/elimination} \\ fix_{ct}\, e \mid & \text{fixpoint} \\ e \rightarrow e', e'' & \text{conditional} \end{cases}$$

Note that only the constants f_i are allowed to vary between interpretations — in particular the fixpoint and conditional terms are always conventionally interpreted (which indeed requires that the spaces $\underline{B}_i$ are always interpreted as *cpo's*). However, looking at this term structure a little closer we can reduce its complexity further by considering syntactic elements (except for λ) as being meta-language constants (for example we can eliminate $(\cdot,\cdot)$ by introducing a constant symbol *pair* and then replacing (e, e') with $pair(e)(e')$ everywhere). This then brings *TML* much closer to the framework we consider here. So perhaps an appropriate slogan is that Nielson's two-level *meta-language* is really a two-level *interpretation* of the uninterpreted typed λ-calculus! See the end of this section for a more detailed proposal.

The development given in section 1 above has a system of types given by

$$t ::= t_1 \times \cdots \times t_n \rightarrow t_0 \mid \underline{B}$$

and terms of the form

$$e ::= f_i \mid x_i \mid \lambda(x_1, \ldots, x_n).e \mid e_0(e_1, \ldots, e_n)$$

Moreover the λ-term is only required in restricted ways to express notions of binding and is employed in such places as the treatment of input in section 4. Mosses [13] discusses this in more detail.

Now it is clear that this system is a greatly simplified version of Nielson's and corresponds to a further honing towards minimality and rational reconstruction following the movement in this direction in Mycroft and Jones [16]. Moreover, we are left to interpret the spaces and functions over any[7] category. Our approach is essentially to treat an interpretation as specifying the meaning of *all* the sorts and operations from the signature of the denotational semantics schema (as in the initial algebra approach to data types [7]). Of course, several symbols would probably have the same interpretation relative to other symbols in all interpretations of interest.

[7]If we wish to explain binding constructs by λ-abstractions (see [13]) then this must be cartesian closed.

Moreover, it is not at all clear that this technique loses any expressive power over the full *TML*, for example we can achieve the effect of $\mathit{fix}_{ct}.e$ by $f_i e$ where we will chose to interpret f_i as the fixpoint-taking constant of type $(ct \rightarrow ct) \rightarrow ct$. It is clear that similar replacements can be done for all the other terms (except λ-abstraction and application) in TML_b. As a slogan this would be "replace pre-interpreted combinators with application of appropriate constants". This example tends to reinforce the view that the latter has some duplication of concepts in the two levels. (This is intuitively clear as (*e.g.*) a constant expression in a programming language can often be interchangeably evaluated at compile-time or run-time.)

On the other hand, we ought to lose some facility. This would seem to be (considering the fixpoint example above) the externalisation of the restriction of f_i to mean *fix*. This movement may have the disadvantage of making the proof that a particular interpretation satisfies a certain correctness (abstraction) relation harder, in that less has been done schematically. However, it would seem that this question has not yet been properly investigated (nor, as far as I know, even discussed) and this should be done with some urgency.

At this point I would like to propose an alternative proposal of a formulation for *TML* as hinted at above. Let us take the liberty of calling this TML_i. TML_i is a meta-language with a type structure of the following form:

$$t ::= A \mid t_1 \rightarrow t_2 \mid \underline{B}$$

i.e. the typed λ-calculus with two levels of type symbols. Its term structure is given by

$$e ::= g_i \mid f_i \mid x_i \mid \lambda x.e \mid e(e')$$

The intention is that this this specifies the typed λ-calculus with two levels of interpretation. The first, fixed, level is that for (A, g_i) of type symbols and constants of type t together with variables x_i of type t for λ-abstraction. The second, variable level, varies over interpretations and specifies $(\underline{B}, f_i)$ as in Nielson's framework. I must emphasis that this is only a tentative proposal, but it seems to come closer to our intuitive views on abstract interpretation. Certainly this notion of two-level interpretation allows us the freedom to work over any algebra — its structure is specified by the first level interpretation and its constants (which vary over interpretations) by the second. Moreover, TML_i allows any types and values for interpretations and thus would appear to sidestep Nielson's restriction requiring constants to possess "contravariently-pure" types [20]. This is connected with the fact that this development works purely with relations and not, as Nielson does, also with (abstraction) functions. A theory of abstract interpetation based on TML_i may also be easier to work with due to the smaller number of expression and type forms. Again research is needed.

2.2 The collecting interpretation and adjoined functions

Nielson's thesis framework required the notion of a *collecting* semantics which was formally given by the power-interpretation of the standard interpretation. The collecting interpretation was then related to abstract interpretations by means of adjoined *abstraction* and *concretisation* functions. This followed previous work on the collecting semantics by Mycroft and Nielson [15,17]. Basically, Nielson's collecting interpretation (which induced the so-called collecting semantics) had the domains which interpreted the *rt* level of meta-language types replaced by their power domains and functions lifted from elements to sets of elements[8]. The problem which arises is that power domains (with all the natural requirements) do not seem to exist at function types, thus Nielson was forced to restrict *TML* by restricting the valid meta-language types to forbid these situations from occurring. Mycroft and Jones [16] gave an alternative development in which the use of adjoined functions with a collecting interpretation were replaced by relations with the standard interpretation.

It may be helpful for me to recant on one point now: I now believe that my development [15] of the collecting interpretation to have been a mistake. It seemed at the time to generalise the Cousots' work [4] on flowcharts in a natural way to functional languages. Certainly it enabled proof about the $\sharp$ function in strictness analysis. However, it seems to present a blind alley and attempts to solve this merely led to unreadably complicated papers. My view now is that the *minimal function graph* semantics of Jones and Mycroft [10] is much closer to the Cousots' collecting semantics.

The view from [16] is that moving away from the viewpoint of abstraction and concretisation functions in *general* (although particular uses of abstract interpretation will no doubt continue to find them a useful special case) will obviate the need for the collecting interpretation and the problems of suitable power domains with it. Nielson's latest work [20] also follows this train of thought.

3 Microcode

In this second part we consider applying some of the above ideas to provide an alternative exposition of aspects of Foster's [6] work on "verifying microcode algbraically". The model of microcode used below is loosely based on Foster's. We suppose our (micro-)machine has a set *Reg* of microcode registers which each may contain values from a set *Val* of values which we assume to contain a distinguished value 0 representing a false condition. Instructions *Inst* may be an (atomic) action (from *Act*) which updates an environment – see later, a conditional branch (based on some expression *Exp*) or a read from or write to a port. Instructions carry a (possibly implicit) indication of which instruction is to be executed next. The letters r, v, i, c, e will be used to range over these sets.

[8]This is one conventional category theoretic answer to describing a relation by a function.

Traditionally, we would say that such a program is given by a graph[9] together with a consistent labelling of its nodes and arcs with instructions. This can be a little messy formally and so here we define:

Microcode programs:

A microcode program is a triple $(N, n_0, code)$ where N is a set of *nodes (labels)* ranged over by n,l. n_0 is a distinguished (start) node and *code* is a function from N to

$$Inst \;=\; Act \times N \;+\; Exp \times (Act \times N)^2 \;+\; Exp \times N \;+\; Reg \times N$$

in which the summands represent command, conditional (possibly with commands executed 'on the fly'), output and input respectively. We will write elements of these summands using the suggestive notation:

$$\begin{array}{l} [\![c; \mathbf{goto}\; l]\!] \\ [\![\mathbf{if}\; e\; \mathbf{then}\; c; \mathbf{goto}\; l; \mathbf{else}\; c'; \mathbf{goto}\; l']\!] \\ [\![\mathbf{write}\; e; \mathbf{goto}\; l]\!] \\ [\![\mathbf{read}\; r; \mathbf{goto}\; l]\!] \end{array}$$

Note the slightly unusual binding of $[\![\mathbf{if}\; e\; \mathbf{then}\; c; \mathbf{goto}\; l; \mathbf{else}\; c'; \mathbf{goto}\; l']\!]$. We have adopted a single (assumed multiplexed) input and output channel for simplicity and it should be clear how to extend the semantics to handle more than this. Otherwise we have provided most of the features of a typical microcode controller expect for micro-procedures and instruction despatch (the latter may be modeled as iterated conditional branch). Note that we do not include a 'halt' instruction – it makes very little sense at the microcode level and its absence simplifies things by ensuring that all execution sequences are infinite.

In our simple model we will treat the computer's main memory as an I/O device as far as microcode is concerned. Now one might argue that this is unreasonable given that we fully expect that writing to a storage location and then reading it twice should yield the same answer and this should be built into our semantics. We will counter by arguing that microcode should be reliable against failures in system components and that no such assumptions should therefore be made. Moreover, such assumptions would be wrong with (*e.g.*) dual-ported memory. We take the view that such arguments concerning the behaviour of *systems* should be made separately in a language designed for this (such as Milner's CCS [12]). Microcode programs should be verified under the assumption that their environment is malicious.

[9]Compare the definition of a graph as a pair $(N, f : N \to \mathcal{P}(N))$ where N is a set of nodes and f gives the set of successor nodes (and hence arcs) from a node.

4 Microcode Semantics

From now one we will assume that we have a fixed program u given as the triple $(N, n_0, code)$.

As is usual in abstract interpretation, we will wish to give several meanings in different universes of discourse to our microcode program whose syntax was introduced above.

To allow us to readily change the set of values manipulated we will suppose the set V (archetype *Val*) gives the values manipulated in a given interpretation. An environment is a (the current) association between register and values, *i.e.* a function (ranged over by ρ) from the set $Env_V = Reg \to V$. We will often drop the subscript on *Env* when it is clear from context or when the discussion is generic.

We assume that the meaning of atomic commands and expressions are given by two functions

$$\mathcal{A} : Act \to Env_V \to Env_V, \qquad \mathcal{E} : Exp \to Env_V \to V$$

Formally, both $\mathcal{A}$ and $\mathcal{E}$ should be specified by interpretations or at least should be subscripted by V *and* the way in which new environments or values are calculated. However, as in the first part the internal details are not of interest. Here we will only use this standard interpretation specified by the hardware (which we write $\mathcal{A}_{Val}, \mathcal{E}_{Val}$ when we wish to stress this) and a one-point interpretation $\mathcal{A}_{\{*\}}, \mathcal{E}_{\{*\}}$ with $V = \{*\}$ which can only have the one definition *viz*

$$\mathcal{A}_{\{*\}}[\![c]\!](\lambda r.*) = (\lambda r.*), \qquad \mathcal{E}_{\{*\}}[\![e]\!](\lambda r.*) = *$$

and so this abuse of notation will not be harmful. The above is a restriction on the possible interpretations we may consider, for example it disallows the "relational method" of [9]. However, this is not important here.

Due to our interest in giving several alternative semantics for a given microcode program (as we discussed in the introduction) we will give a *schematic* semantics in which several domains and functions are left uninterpreted.

An interpretation will be required to specify three domains and five functions:

V	as discussed above — specifies $\mathcal{A}, \mathcal{E}$ too.
D	a domain of denotations — see interpretations
Ans	a domain of answers
$atom : Inst \times D \to D$	atomic actions
$cond : Inst \times V \times D \times D \to D$	conditional actions
$output : Inst \times V \times D \to V$	output actions
$input : Inst \times (V \to D) \to D$	input actions
$init : (Node \times Env_V \to D) \to Ans$	gives starting conditions and answer extraction

As in the first part, this induces a semantics by:

$$\begin{aligned}
&\mathcal{U} : \mathit{Program} \to \mathit{Ans} \\
&\mathcal{U}[\![u]\!] = \mathit{init}(\phi) \textbf{ whererec} \\
&\phi(n,\rho) \;=\; \textbf{case } \mathit{code}(n) \textbf{ of} \\
&\qquad [\![c; \textbf{goto } l]\!].\mathit{atom}(\mathit{code}(n), \phi(l, \mathcal{A}[\![c]\!]\rho)) \\
&\qquad [\![\textbf{if } e \textbf{ then } c; \textbf{goto } l; \textbf{else } c'; \textbf{goto } l']\!].\mathit{cond}(\mathit{code}(n), \mathcal{E}[\![e]\!]\rho, \phi(l, \mathcal{A}[\![c]\!]\rho), \phi(l', \mathcal{A}[\![c']\!]\rho)) \\
&\qquad [\![\textbf{write } e; \textbf{goto } l]\!].\mathit{output}(\mathit{code}(n), \mathcal{E}[\![e]\!]\rho, \phi(l,\rho)) \\
&\qquad [\![\textbf{read } r; \textbf{goto } l]\!].\mathit{input}(\mathit{code}(n), \lambda v.\phi(l, \rho[v/r]))
\end{aligned}$$

Of course, the use of **whererec** implies the use of a *fix* constant which must also be specified in our interpretations, but we will always interpret this as least fixpoint and accordingly omit it.

In some sense this definition does not 'feel' denotational, especially after considering the simple while-language *Cmd* in the first part. The cause of this ill-feeling is that *Cmd* had simple rules for constructing bigger programs from their constituent parts which involved at most three components. For flowcharts, we have one construction rule for every graph – and it can only be applied once. This is also the source of the criticism that flowcharts are much less 'structured' than corresponding while-programs.

Having supplied such a semantic scheme, we must give it a *standard interpretation* which specifies its standard semantics[10]. The attitude taken is that the space of answers is an interactive input/output stream. This is modelled denotationally by the domain equation for *Ans* in the type part of the interpretation:

$$\begin{aligned}
&\mathit{Ans} = (\mathit{Val} \times \mathit{Ans}) \;+\; (\mathit{Val} \to \mathit{Ans}) \\
&V = \mathit{Val} \\
&D = \mathit{Ans}
\end{aligned}$$

where $\times$ is the cartesian product and $+$ is the separated sum. Intuitively this means that the program computes for a while until it possibly either offers output and computes further (the left summand) or waits for input by providing a function which when applied to a value yields the answer corresponding to rest of the computation (the right summand). Note that interactive I/O is of the essence, since microcode typically sends an address to an output port and then reads an input port to read the associated data. The expression part of the interpretation is given by:

$$\begin{aligned}
&\mathit{atom}(i,d) = d \\
&\mathit{cond}(i,v,d,d') = v \neq 0 \to d, d' \\
&\mathit{output}(i,v,d) = \mathit{in}_1(v,d) \\
&\mathit{input}(i,f) = \mathit{in}_2(f) \\
&\mathit{init}(\phi) = \phi(n_0, \lambda r.0)
\end{aligned}$$

where in_1 and in_2 are the injection functions associated with the sum type for *Ans* above.

[10]But see section 4.1 for complications timing constraints may place on such a semantics.

Following the discussion in the introduction about continuation and direct semantics the above semantics we have given is in some senses direct (in that its first clause applies meaning of the 'rest of the program' to the state resulting from its first action). However, it has the continuation-style ability to yield output before termination. This shows the rather artificial nature of the pedagogic distinction between the two extremes.

Instruction trace interpretations

In many ways the semantics we have just constructed is 'too abstract'. For example the meaning of a program is just its input/output relationship with no details preserved as to (*e.g.*) timing – one of the points we particularly wish to discuss. Accordingly we need to add *operational detail* to distinguish microcode programs with a given input/output behaviour and which respect timing constraints from others with the same behaviour but which fail to meet timing constraints. Accordingly we consider the semantics given by the interpretation:

$$IT_1 = \begin{cases} Ans = Inst \times ((Val \times Ans) + (Val \to Ans) + Ans) \\ V = Val \\ D = Ans \\ atom(i,d) = (i, in_3(d)) \\ cond(i,v,a,a') = (i, in_3(v \neq 0 \to a, a')) \\ output(i,v,a) = (i, in_1(v,a)) \\ input(f) = (i, in_2(f)) \\ init(\phi) = \phi(n_0, \lambda r.0) \end{cases}$$

This interpretation gives a semantics in which a trace of all instructions executed is recorded as part of the meaning of the program. We do not keep a trace of the values of the registers during execution even though it would be quite easy to do so. This corresponds to the fact that we wish to model Foster's framework in which values of registers are not considered in calculating the regular expression path algebra.

Note: there were many ways in which this augmentation could have been done, and not all would be correct. It is *not* part of the framework of abstract interpretation to show that the augmentation is correct. Certainly we can define an appropriate abstraction relation (here function) from the augmented semantics to the standard semantics and thereby show that the standard semantics is some form of projection:

$$\begin{aligned} &\theta : Ans_{IT_1} \to Ans_{standard} \\ &\theta(i, in_3(d)) = \theta(d) \\ &\theta(i, in_1(v,d)) = in_1(v, \theta(d)) \\ &\theta(i, in_2(f)) = in_2(\lambda v.\theta(fv)) \end{aligned}$$

However this is far from showing that the augmented semantics is correct! I feel that insufficient attention has been given to this problem in the literature.

Following the discussion in the introduction we now start to consider instruction sequences which may be executed subject the the assumption of a malicious world in which any **read** statement may read any datum. This can be specified by an interpretation with:

$$IT_2 = \begin{cases} Ans = \mathcal{P}(Inst^+) \\ V = Val \\ D = Ans \\ atom(i,d) = add(i,d) \\ cond(i,v,d,d') = add(i, v \neq 0 \rightarrow d, d') \\ output(i,v,d) = add(i,d) \\ input(i,f) = add(i, \bigcup_{v \in V} f(v)) \\ init(\phi) = \phi(n_0, \lambda r.0) \end{cases}$$
$$where\ add(i,d) = \{i :: x \mid x \in d\}$$

where $Inst^+ = (Inst \times Inst^+)_\perp$ is the set of (possibly infinite) sequences of instructions with constructor function ::. This interpretation collects a trace of all instruction sequences assuming that each read operation may read any possible datum, but still assuming calculation within the interpretation is precise.

Of course, this is still in general difficult to compute (it would generally be undecidable if *Val* were infinite) and one much closer to Foster's is given by:

$$IT_3 = \begin{cases} Ans = \mathcal{P}(Inst^+) \\ V = \{*\} \\ D = Ans \\ atom(i,d) = add(i,d) \\ cond(i,v,d,d') = add(i, d \cup d') \\ output(i,v,d) = add(i,d) \\ input(i,f) = add(i, f(*)) \\ init(\phi) = \phi(n_0, \lambda r.*) \end{cases}$$

and where *add* is as in IT_2. Now we have identified all data items (some form of terminal interpretation – at least with respect to the definition of *Ans*) and the consequence is that the program graph determines its semantics and that conditional branches are all assumed to 'go both ways'. As a remark we note that this naturally identifies programs such as

n_0 : **if** r **then** c_0; **goto** n_1; **else** c_1; **goto** n_1
n_1 :

and

n_0 : **if** r **then** c_1; **goto** n_1; **else** c_0; **goto** n_1
n_1 :

and that therefore not all program analyses (nor exact execution) can be modelled in this scheme.

Now what we have developed is an interpretation which defines a semantics for microcode programs in which meanings are sets of strings of instructions which include all those which may actually be executed in the standard interpretation. What we have is essentially a theoretical understanding of the formal language of execution sequences and Foster's implementation associates them with one regular expression describing that language.

In the denotational semantics sense our interpretation is more abstract whereas Foster's is far more implementable.

Foster uses regular expressions to represent the execution paths through microcode programs and then uses homomorphisms on these to determine program properties. Because he uses regular expressions, microcode programs (despatch loops) interpreting higher level instruction sets are commonly represented by

$$program = init.(ins_0 + ins_1 + \cdots + ins_n)^*$$

Therefore, because the code $[\![l : c;\ \mathbf{goto}\ l]\!]$ is not representable as a regular expression Foster's algorithm for constructing the regular expression from a program detects such loops before applying homomorphisms. (Note that simply ignoring such a construct would lead to errors if one wanted to know (*e.g.*) the maximal time between two interrupt polling instructions.) One may be tempted to suggest that a more natural form for a microcode 'regular' expression, bearing in mind that microcode programs do not reasonably "halt", is in fact

$$program = init.(ins_0 + ins_1 + \cdots + ins_n)^\omega$$

where *init* and the ins_i are regular expressions. This would accord closer with our framework where we have to consider infinite computation sequences. However it is not clear how to fit such an idea into Foster's framework and moreover it is not clear that it bestows any benefit in that his homomorphisms would detect (for example) any ins_i which had a net change on the microcode stack register with $(\cdot)^*$ equally well as with $(\cdot)^\omega$. Certainly we could model this by changing our interpretation so that such a despatch loop could produce finite as well and infinite execution sequences. However, we will not do so as in our case the natural choice seems to be only to consider infinite computation sequences and their abstractions – an interesting difference between theory and practice!

4.1 Discussion on semantic correctness

We noted that the standard interpretation and semantics are so abstract that they take no account of the states through which the microcode interpreting mechanism passes. This is generally a good thing in that we do not wish to clutter up the semantics of a high level language with details of its possible inplementation techniques. We then added details of instruction traces in successive interpretations. However, in microcode it is quite conceivable for (*e.g.*) the effect of a subroutine to depend on its (dynamic) nesting — certainly microcode sequencers often include some limit on subroutine nesting. Moreover, because of the relationship between microcode instruction counts and hardware timing for interrupts and the like, we will in general require a much less abstract semantics — another example would be the real possibility that a read operation may not follow a write operation for at least n cycles. One formal way of doing this is to define a 'fully checking' interpretation in which the state contains a component specifying machine details (such as clock ticks since startup, subroutine nesting and the like). However, these tend to be extremely complicated reflecting the complexity of the underlying hardware. Another, which is very much in the spirit of Foster's work is to assume a 'perfect' semantics but also to consider the exact execution instruction trace semantic interpretation IT_1. If the trace component of Ans_{IT_1} contains an invalid instruction sequence or violation of timing constraint then we treat the answer as if a fully checking semantics had produced an error value. Now we can characterise several of Foster's homomorphisms as exhibiting images of Ans_{IT_1} in which absence of such errors there implies that the standard semantics is valid.

5 Abstraction relations corresponding to Foster's homomorphisms

Above we derived an interpretation IT_3 for microcode programs which produced traces of all possible executions by assuming that **read** actions produced arbitrary values and by identifying all data values.

Let us now consider one of Foster's microcode validation homomorphisms which determines the maximum number of instructions executed between two "interrupt poll" instructions. Given an instruction trace set $S \in \mathcal{P}(Inst^+)$ and a set $Poll \subset Inst$ we can derive this quantity by a definition of the form (provided we use the order $\sqsubseteq = \leq$ on $\mathbb{N}$ – this will be written $\mathbb{N}^{\leq}$):

$$\begin{aligned}
&\delta : \mathcal{P}(Inst^+) \to \mathbb{N}^{\leq} \\
&\delta(S) = \bigsqcup\{x \in S \mid \delta'(x)\} \\
&\delta' : Inst^+ \to \mathbb{N}^{\leq} \\
&\delta'(x) = \bigsqcup\{n \in \mathbb{N} \mid (\exists i \in \mathbb{N}) x_i \in Poll \wedge x_{i+n} \in Poll \wedge OK(x, i, i+n)\} \\
&OK(x, i, j) \Leftrightarrow (\forall k)(i < k < j \Rightarrow x_k \notin Poll)
\end{aligned}$$

where $\delta'(x)$ gives the maximum distance apart of any two members of $Poll$ in the sequence x.

It is not immediately clear that the above definition of δ is continuous with respect to these orderings. In fact, writing a fixpoint definition for δ' leads to considering pairs of the form (n, m) where n is the time from the last interrupt poll and m is the maximum separation so far (of course $n \leq m$). Accordingly δ' may be defined in terms of

$$try(i, (n, m)) = \begin{cases} (0, m) & \mathit{if}\ i \in Poll \\ (n+1, m+1) & \mathit{if}\ i \notin Poll \wedge n = m \\ (n+1, m) & \mathit{if}\ i \notin Poll \wedge n < m \end{cases}$$

The problem is that this characterisation is somewhat *ad hoc* and indirect. We would rather define an interpretation *POLL* to express the value directly as

$\mathcal{U}_{POLL}[\![u]\!]$ instead of $\delta(\mathcal{U}_{IT_3}[\![u]\!])$. This can be done by:

$$POLL = \begin{cases} Ans = I\!N^{\leq} \\ D = I\!N^{\leq} \times I\!N^{\leq} \\ V = \{*\} \\ atom(i,d) = try(i,d) \\ cond(i,v,d,d') = try(i,d) \sqcup try(i,d') \\ output(i,v,d) = try(i,d) \\ input(i,f) = try(i,f(*)) \\ init(\phi) = \pi_2(\phi(n_0, \lambda r.*)) \end{cases}$$

where π_2 is the projection $\sigma \times \tau \to \tau$ and *try* as above.

One final thing remains, to set up a relation between $POLL$ and IT_3 and to compare it to Foster's homomorphism. It turns out that all the base type cases of the abstraction relations can be defined by functions:

$$\begin{aligned} &abs_V : \{*\} \to \{*\} \\ &abs_D : D_{IT_3} \to D_{POLL} \\ &abs_{Ans} : Ans_{IT_3} \to Ans_{POLL} \\ &abs_V(*) = * \\ &abs_D(S) = (\textstyle\bigsqcup_{x \in S} \sqcap\{i \mid x_i \in Poll\}, \delta(S)) \\ &abs_{Ans}(S) = \delta(S) \end{aligned}$$

So writing, for space $\alpha \in \{V, D, Ans\}$,

$$x \rhd_\alpha y \Leftrightarrow y = abs_\alpha(x)$$

we have that $IT_3 \rhd_{V,D,Ans} POLL$ and thus Foster's homomorphism has become an abstraction relation.

As a final comment it seems necessary to remark that in a sense this homomorphism did not convert to an abstraction relation in anything like such a natural manner as I had anticipated when I started this work. This may simply represent my missing some obvious short-cut. However, the above use of the domain $I\!N^{\leq} \times I\!N^{\leq}$ seems essential for (continuously) capturing the notion of maximum polling interval. On the other hand it is clear that one only wants one component of this as an answer in order to satisfy the correctness relation

$$\delta(\mathcal{U}_{IT_3}[\![u]\!]) \leq \mathcal{U}_{POLL}[\![u]\!]$$

The most likely explanation is that Foster has found a framework which sidesteps the domain theory used here by his use of regular languages.

Acknowledgments

Thanks to Mike Gordon, Mogens Nielsen and Glynn Winskel here at Cambridge for discussions which aided my understanding. Also thanks to the editors for perceptive and useful comments and for tolerating my inspiration's inability to meet their deadlines.

References

[1] Abramsky, S. Abstract interpretation, logical relations and Kan extensions. Unpublished manuscript, 1985. (An earlier version was entitled "Strictness analysis based on logical relations".)

[2] Bauer, F.L., Berghammer, R., Broy, M., Dosch, W., Gnatz, R, Hangel, E., Möller, B., Partsch, H., Pepper, P., Samelson, K. and Wössner, H. "The Munich project CIP – volume 1: the wide spectrum language CIP-L", vol. 183, Springer-Verlag, 1985.

[3] Burn, G., Hankin, C. and Abramsky, S. The theory and practice of strictness analysis for higher order functions. Imperial college report DoC 85/6, 1985. To appear in *Science of Computer Programming.*

[4] Cousot, P. and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction and approximation of fixpoints. Proc. ACM symp. on Principles of Programming Languages, 1977.

[5] Donzeau-Gouge, V. Utilisation de la sémantique dénotationelle pour l'étude d'interprétations non-standard. INRIA rapport 273, 1978.

[6] Foster, J.M. Validating microcode algebraically. Research report, Royal Signals and Radar Establishment, Malvern, Worcs., UK, 1985. To appear in Computer Journal??

[7] Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. "Initial algebra semantics and continuous algebras", JACM vol. 24, no. 1, 1977.

[8] Gordon, M.J.C. The denotational description of programming languages. Springer-Verlag 1979.

[9] Jones, N.D. and Muchnick, S.S. Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra. Proc. 20th Conf. on Foundations of Computer Science, 1979.

[10] Jones, N.D. and Mycroft, A. Dataflow of applicative programs using minimal function graphs. Proc. ACM symp. on Principles of Programming Languages, 1986.

[11] Milne, R.E. and Strachey, C. A theory of programming language semantics. Chapman and Hall, 1976.

[12] Milner, R. A calculus for communicating systems. Lecture notes in computer science, vol. 92, Springer-Verlag 1980.

[13] Mosses, P.D. A semantic algebra for binding constructs. DIAMI report PB-132, Computer science dept., Aarhus University, 1981.

[14] Mosses, P.D. Abstract semantic algebras. Proc. IFIP TC2 Working conf. on formal description of programming concepts II, Garmisch, North-Holland, 1982. Also available as DIAMI report PB-145, Computer science dept., Aarhus University.

[15] Mycroft, A. Abstract Interpretation and Optimising Transformations of Applicative Programs. Ph.D. thesis, Edinburgh University, 1981. Available as computer science report CST-15-81.

[16] Mycroft, A. and Jones, N.D. A relational framework for abstract interpretation. Lecture Notes in Computer Science: Proc. Copenhagen workshop on programs as data objects, vol. 215, Springer-Verlag, 1985.

[17] Mycroft, A. and Nielson, F. Strong abstract interpretation using power domains. Lecture Notes in Computer Science: Proc. 10th ICALP, vol. 154, Springer-Verlag, 1983.

[18] Nielson, F. Abstract interpretation using domain theory. Ph.D. thesis, Edinburgh University, 1984. Available as computer science report CST-31-84.

[19] Nielson F. Abstract interpretation of denotational definitions. Research report R-85-5, Institut for electroniske systemer, Aalborg Universitetscenter, Aalborg, Denmark, 1985.

[20] Nielson F. Strictness analysis and abstract interpretation of denotational definitions. Unpublished manuscript, 1986.

[21] Nielson, H.R. and Nielson F. Pragmatic aspects of two-level denotational meta-languages. Research report R-85-13, Institut for electroniske systemer, Aalborg Universitetscenter, Aalborg, Denmark, 1985.

[22] Plotkin, G. Lambda definability in the full type hierarchy. In [25].

[23] Reynolds, J.C. On the relation of direct and continuation semantics. Lecture Notes in Computer Science: Proc. 2nd ICALP, vol. 14, Springer-Verlag, 1974.

[24] Reynolds, J.C. Types, abstraction and parametric polymorphism. IFIP 83, (ed) R.E.A. Mason, North-Holland, 1983.

[25] Seldin, J.P., and Hindley, J.R. To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism. Academic Press, 1980.

[26] Stoy, J. Denotational semantics: the Scott-Strachey approach to programming language theory. MIT press, 1977.

[27] Tennent, R.D. Principles of programming languages. Prentice-Hall, 1981.