Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

210

STACS 86

3rd Annual Symposium on Theoretical Aspects
of Computer Science
Orsay, France, January 16–18, 1986

Edited by B. Monien and G. Vidal-Naquet

Springer-Verlag
Berlin Heidelberg New York Tokyo

Editors

B. Monien
Universität-Gesamthochschule Paderborn
Fachbereich Mathematik/Informatik
Warburgerstr. 100, 4790 Paderborn, FRG

G. Vidal-Naquet
Laboratoire de Recherche en Informatique
Bâtiment 490, Université de Paris-Sud, 91405 Orsay Cedex, France

CR Subject Classifications (1985): B.7.2, C.2, C.1, F.1, F.2, F.3, F.4, H.2.1

ISBN 3-540-16078-7 Springer-Verlag Berlin Heidelberg New York Tokyo
ISBN 0-387-16078-7 Springer-Verlag New York Heidelberg Berlin Tokyo

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

FOREWORD

This volume contains the papers presented at the Third Symposium on Theoretical Aspects of Computer Science (STACS 86) held at the University Paris-Sud January 16-18, 1986.

This Symposium is organized jointly by the Special Interest Group for Theoretical Computer Science of the "Gesellschaft für Informatik" (G.I.) and the special interest group for applied mathematic of the "Association Française des Sciences et Techniques de l'Information, de l'Organisation et des Systèmes" (AFCET). It is held alternatively in France and Germany.

In response to the call for papers 122 papers were submitted.

The program committee met on September 27th and selected 29 papers chosen on the basis of their scientific qualities and relevance to the Symposium. Three invited talks were given.

On behalf of the program committee the symposium chairmen would like to thank all those who submitted papers and the referees who helped for the task of selecting papers.

Our thanks also to the sponsoring organizations, the secretariat of AFCET and Sylvie Congnard for the preparation of the symposium.

B. MONIEN - University of Paderborn

G. VIDAL-NAQUET - University of Paris-Sud, Ecole Supérieure d'Electricité

LIST OF REFEREES

E. AESTESIANO
A. ALBANO
J. ALBERT
V. AMBRIOLA
A. ARNOLD
J. AUTEBERT
R. BACK
R. BADE
R. BARBUTI
G. BAUDET
J. BEAUQUIER
M. BELLIA
J. BERMOND
J. BERSTEL
E. BERTINO
A. BERTOSSI
M. BIDOIT
G. BILLAUD
A. BLIKLE
N. BLUM
L. BOASSON
L. BOUGE
A. BRANDSTÄDT
A. BRAUGGEMANN-KLEIN
G. BREBNER
M. BROY
P. CAMION
R. CAPOCELLI
H. CARSTENSEN
I. CASTELLANI
M. CASTERAN
P. CHARPIN
C. CHOPPI
H. COHEN
R. CORI
G. COSTA
P. COUSOT
R. COUSOT
M. CROCHEMORE
P. DEGANO
P. DEMBINSKI
R. DE NICOLAS
W. DIEKERT
T. DOEPPNER
P. DOLLAND
A. DOUCET
J. EBERT
G. EDELSBRUNNER
J. FALLOT
E. FEHR
J. FERBUS
A. FILE
A. FINKEL
A. FISCHER
U. FISSGUS
M. FLE
P. FRAISSE
L. FRIBOURG
C. FROUGNY
M. GAUDEL
E. GHELLI
S. GNESI
D. GOUYOU-BEAUCHAMPS
A. HEINZ
N. HOLSTI
P. INVERARDI
M. JANTZEN
M. JERRUM
M. KANELLAKIS
S. KAPLAN
V. KERÄNEN
R. KLEIN
J. KORTELAINEN
K. KOSKIMIES
T. KRETSCHMER
M. KUDLEK
G. LALLEMENT
J. LAMBERT
K. LANGE
M. LATTEUX
D. LAZARD
C. LERMEN
T. LETTMANN
A. MAGGIOLO
V. MANCA
H. MANNILA
K. MELHORN
R. MIGNOTTE
P. MOLITOR
E. NETT
F. NICKL
M. NIELSON
O. NURMI
E. OLDEROG
P. ORPONEN
R. ORSINI
F. OTTO
J. PANSIOT
E. PENAUD
M. PENTONEN
P. PEPPER
M. PROTASI
L. PUEL
F. RADEMACHER
K. RAIHA
J. RAOULT
W. RULLING
A. SALIBRA
A. SALOMAA
D. SANNELLA
J. SCHMIDT
D. SCHOETT
T. SCHRÖDER
P. SCHUSTET
S. SCHWER
G. SENIZERGUES
K. SIEBERD
M. SIMI
S. SIPPU
E. SOISALON SOININEN
G. SONTACHI
J. STERN
C. STIRLING
H. STORK
A. STOUGHTON
T. STROHOTTE
J. TARHIO
P. TURAKAINEN
F. TURINI
R. VALK
M. VANNESCHI
R. VAUQUELIN
J. VAUTHERIN
R. VERBEEK
K. WAGNER
L. WALLEN
R. WANKMÜLLER
P. WEGNER
P. WIDMAYER

CONTRIBUTED PAPERS

* Contributions not submitted in time for publication

ABSTRACT INTERPRETATION OF DENOTATIONAL DEFINITIONS

(A survey)

Flemming Nielson
Institute of Electronic Systems
Aalborg University Centre
Strandvejen 19, 4
9000 Aalborg C.
Denmark

ABSTRACT

Abstract interpretation is a framework for describing data flow analyses and for proving their correctness. Traditionally the framework is developed for flow chart languages, but this paper extends the applicability of the idea to a wide class of languages that have a denotational semantics. The main idea is to study a denotational metalanguage with two kinds of types: one kind describes compile-time entities and another describes run-time entities. The run-time entities will be interpreted differently so as to obtain different semantics from the same denotational definition: the standard semantics is the ordinary semantics, an approximating semantics describes a data flow analysis and the collecting semantics is a convenient tool in relating the previous two semantics.

1. INTRODUCTION

Often the acceptability of using some high-level programming languages depends critically on the quality of the code produced by the compiler: sometimes the code must execute fast and sometimes the code must be short. Consider the fragment

if -9 × -3 + 5 > 0 then...else...

and the problem of generating suitable code for it. For this it would help to know at compile-time that -9 × -3 + 5 is positive because then no code needs to be generated for the test and the else branch. To obtain such information a compiler performs some data flow analyses [AhUl 78, Hec 77]. For example the properties

negative, positive, zero, integer, true, false, boolean

may be used in a calculation showing that

negative × negative + positive > zero

yields positive > zero and hence true.

The formulation of data flow analyses depends on the way programs are modelled. When they are modelled as flow charts the monotone framework of [KaUl 77] provides a very general setting for the formulation. The work by Patrick and Radhia Cousot on abstract interpretation [CoCo 77, CoCo 79] ex-

tends this by incorporating a correctness relation between properties and values. In this way many data flow analyses can be specified and proved correct. However, it is often more convenient to have a more structural model of programs. In this setting the high-level data flow analyses of [Ros 77] provides a framework for formulating data flow analyses, but no theory of correctness has been developed.

Denotational semantics is a structural method for expressing the meaning of programs. This paper surveys a framework of abstract interpretation that is applicable to all programming languages that have a denotational definition using the metalanguage of section 3. Previously such developments have only been performed for particular languages (e.g. [Don 78, Don 81, Nie 82, MyNi 83, BuHA 85]). The detailed development is given in [Nie 84]. Here no details of particular data flow analyses will be given as such examples can be found elsewhere (e.g. [CoCo 78]). Such examples include constant propagation [AhUl 78], array bound checking, type checking in a typed language and type inference in an untyped language [Ten 74].

2. INTRODUCTORY EXAMPLE

For the benefit of the reader unacquainted with abstract interpretation we shall begin with an example. The framework to be developed should include the flow charts so consider the following program:

```
a: if x<y then goto b
   x:=x-y
   goto a
b: skip
```

For positive integers x and y the program calculates the remainder of dividing x by y. The semantics of the program may be viewed as a partial function

$$f: Z\times Z \hookrightarrow Z\times Z$$

over pairs of integers. In the terminology of [MiSt 76] this function is called the standard semantics of the program. It is formally defined by

$$f = FIX(\lambda(\phi_a,\phi_b).((\lambda s.f_1(s) \rightarrow \phi_b(s),(\phi_a\cdot f_2)(s)),f_3))\downarrow 1$$

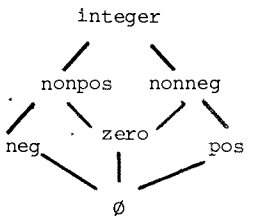
where FIX is the least fixed point operator, $f_1(x,y)=x<y$, $f_2(x,y)=(x-y,y)$, $f_3(x,y)=(x,y)$ and $\ldots\rightarrow\ldots,\ldots$ is conditional.

A data flow analysis amounts to giving the program a non-standard semantics h which will be called an approximating semantics. In a data flow analysis for detecting the signs of integers one may have

$$h: L\times L \rightarrow L\times L$$

where L is a set of properties of the integers. A central notion is that of a property safely approximating another, e.g. "integer" safely approximates "positive". This is modelled by equipping L with a partial order $\sqsubseteq$ such that $l_1 \sqsubseteq l_2$ whenever l_2 safely approximates l_1. In the detection of signs example from section 1 one may use:

L =



The definition of h is given by

$$h = FIX(\lambda(\psi_a,\psi_b).((\lambda\sigma.\psi_b(h_{1t}(\sigma)))\sqcup(\psi_a\cdot h_2)(h_{1f}(\sigma))),h_3))\downarrow 1$$

The intention of $h_{1t}(\sigma)$ is that it should be a safe approximation to the part of the argument for which x<y may be true. It is common to use $h_{1t}(\sigma)=\sigma$ but one could be more precise and specify e.g. $h_{1t}(pos,neg)=(\emptyset,\emptyset)$ etc. The definition of h_{1f} is similar and h_2 and h_3 should be straightforward. It is common to require L to be a complete lattice so that the effect of combining the results along the then and else branches is modelled by the binary least upper bound operator $\sqcup$.

When x and y are initially both positive the resulting value of x will be non-negative and y will remain positive. Therefore one might hope that h(pos,pos)=(nonneg,pos) but the definition of h gives the less precise h(pos,pos)=(integer,pos) because $h_2(pos,pos)=(integer,pos)$. The problem is that h_{1f} is unable to record the connection between the values of x and y. This means that the data flow analysis is an independent attribute method [JoMu 81]. It does not suffice to replace L with a more extensive set of properties such as the powerset $P(Z)$ ordered by subset inclusion $\subseteq$. What is required is to use an adequate description of pairs of integers and $P(Z\times Z)$ is a good candidate. The collecting semantics (static semantics [CoCo 77])

$$g: P(Z\times Z) \to P(Z\times Z)$$

may then be defined by

$$g = FIX(\lambda(\phi_a,\phi_b).((\lambda\underline{\sigma}.\ \phi_b(g_{1t}(\underline{\sigma}))\cup(\phi_a\cdot g_2)(g_{1f}(\underline{\sigma}))),g_3))\downarrow 1$$

where

$$g_{1t/f}(\underline{\sigma}) = \{(x,y)\in\underline{\sigma} \mid f_1(x,y) = true/false\}$$

$$g_{2/3}(\underline{\sigma}) = \{(u,v) \mid \exists(x,y)\in\underline{\sigma}:\ f_{2/3}(x,y)=(u,v)\}.$$

It is a relational method [JoMu 81] and gives

$$g(\{x|x \text{ is positive}\}\times\{y|y \text{ is positive}\})$$

$$\underline{\subseteq} \{x|x \text{ is non-negative}\}\times\{y|y \text{ is positive}\}$$

because g_{1f} does record the connection between the values of x and y.

The collecting semantics is of interest because it intuitively is the most precise data flow analysis consistent with the standard semantics, i.e.

$$g(\underline{\sigma}) = \{(u,v)\,|\,\exists(x,y)\in\underline{\sigma}: (u,v)=f(x,y)\}.$$

This is the case whenever g_{1t}, g_{1f}, g_2 and g_3 are as in the previous paragraph. The connection between g and h is established using a pair (α,γ) of adjoined functions [CoCo 79]. The upper adjoint

$$\gamma: L\times L \to P(Z\times Z)$$

gives the meaning of pairs of properties and is defined as $\gamma(l_1,l_2)=\overline{\gamma}(l_1)\times\overline{\gamma}(l_2)$ where $\overline{\gamma}:L\to P(Z)$ sends pos to $\{z|z \text{ is positive}\}$ etc. The lower adjoint

$$\alpha: P(Z\times Z) \to L\times L$$

is uniquely determined by the adjoinedness condition

$$\alpha(\underline{\sigma})\underline{\sqsubseteq}(l_1,l_2) \Leftrightarrow \underline{\sigma} \underline{\subseteq} \gamma(l_1,l_2).$$

This implies that α and γ are both monotonic, i.e. that they preserve the relation "safe approximation". Furthermore $\gamma(\alpha(\underline{\sigma}))\underline{\supseteq}\underline{\sigma}$, which means that $\alpha(\underline{\sigma})$ is a safe representation of $\underline{\sigma}$, and $\alpha(\gamma(l_1,l_2))\underline{\sqsubseteq}(l_1,l_2)$ which means that $\alpha(\underline{\sigma})$ is as precise a description as possible. The desired relation between g and h then is

$$g(\underline{\sigma})\underline{\subseteq}\gamma(h(\alpha(\underline{\sigma})))$$

which is abbreviated to $g\underline{\subseteq}\gamma\cdot h\cdot\alpha$. This relation is equivalent to

$$g\cdot\gamma\underline{\sqsubseteq}\gamma\cdot h$$

because (α,γ) is a pair of adjoined functions. The relation holds whenever there are similar relations between g_2 and h_2, g_3 and h_3, g_{1t} and h_{1t} as well as g_{1f} and h_{1f}.

3. METALANGUAGE

The denotational semantics of a programming language may be viewed as consisting of:

- a structurally defined mapping from programs to terms in a semantic metalanguage, and
- an interpretation of the primitives of the metalanguage.

Usually only one interpretation is considered and the standard semantics results.

We shall aim at obtaining the approximating and collecting semantics as well by changing the interpretation. This places certain demands upon the metalanguage.

Usually the metalanguage is a typed λ-calculus so let us begin with consideration of the types. As an aid we shall concentrate upon the functionality of the arguments to FIX in the definitions of f, h and g:

$$(Z\times Z\to Z\times Z)\times(Z\times Z\to Z\times Z) \to (Z\times Z\to Z\times Z)\times(Z\times Z\to Z\times Z)$$

$$(L\times L\to L\times L)\times(L\times L\to L\times L) \to (L\times L\to L\times L)\times(L\times L\to L\times L)$$

$$(P(Z\times Z)\to P(Z\times Z))\times(P(Z\times Z)\to P(Z\times Z)) \to (P(Z\times Z)\to P(Z\times Z))\times(P(Z\times Z)\to P(Z\times Z))$$

To obtain these as different interpretations of a formal type we shall use the type:

$$\underline{Z}\times\underline{Z}\to\underline{Z}\times\underline{Z}\times\underline{Z}\to\underline{Z}\times\underline{Z}\to\underline{Z}\times\underline{Z}\to\underline{Z}\times\underline{Z}\to\underline{Z}\times\underline{Z}.$$

It should be clear that the first two functionalities can be obtained by suitably interpreting the underlined symbols, e.g. $\underline{Z}$ as Z and L respectively. The same holds for the third functionality if $\underline{\times}$ is interpreted as the operator ⊗ temporarily defined by $P(A)\otimes P(B)=P(A\times B)$. It should also be clear from this example that it is necessary to distinguish between $\underline{\times}$ and × etc.

Based on this experience one may consider the system TMLs of types [+]:

$$ct ::= A_i \mid ct_1+\ldots+ct_k \mid ct_1\times\ldots\times ct_k \mid ct_1\to ct_2 \mid \mathrm{rec}X.ct \mid X \mid ft$$

$$ft ::= rt_1\underline{\to}rt_2$$

$$rt ::= \underline{A_i} \mid rt_1\underline{+}\ldots\underline{+}rt_k \mid rt_1\underline{\times}\ldots\underline{\times}rt_k \mid \underline{\mathrm{rec}}\ \underline{X}.rt \mid \underline{X}.$$

Imagine for a moment that ct::=ft is omitted. The types ct then are those usually found in a semantic metalanguage: A_i are the base types, + gives disjoint sum, × gives cartesian product, → gives function space and rec X.ct stands for the type of the solution to the equation X=ct. The types rt resemble ct except that rt::=ft has been omitted for simplicity. The best way to view the types ct is as being the type of compile-time (or static) entities. Similarly the types rt correspond to runtime (or dynamic) entities. The functions of type ft then are the state transformations which are of prime interest in abstract interpretation. It therefore makes sense to connect the two systems with ct::=ft. Also it becomes clear that the absense or rt::=ft means that "storable procedures" cannot be accomodated.

+ With respect to [Nie 84] it appears that * and $\underline{*}$ have been omitted. However, from section 4 it will emerge that $\underline{\times}$ here corresponds to $\underline{*}$ in [Nie 84] so that it is really * and $\underline{\times}$ of [Nie 84] that have been omitted. The use of $\underline{X}$ rather than X is a notational improvement.

Remark The development of the present paper constitutes one example of a non-standard semantics (in the sense of [MiSt 76]) where the distinction between compile-time and run-time types is indispensable. Another example where this is the case is when specifying code generation [NiNi 85a], where rt→rt essentially is a domain of code for state transformations over the state rt. Perhaps it is more interesting that the distinction may also prove useful when defining the standard semantics (in the sense of [MiSt 76]) for actual programming languages. One example is the distinction in [Ten 81] between expression procedures and static expression procedures. Ignoring side effects and free variables some typical types would be Z→Z and Z→Z respectively, so that the metalanguage does allow to distinguish between the two concepts. Another example is the programming language type array [...] of integer where ... denotes a static expression determining the size. Here one might allow a run-time type construction Π(e)rt to denote the product of e copies of rt assuming that e denotes a compile-time integer. In general one would expect one notion of type for each binding time [JoMu 78]. □

Turning to the expressions of the metalanguage the goal is to obtain the definitions of f, h and g by suitably interpreting a formal expression. For this we shall use the expression

$$FIX(\lambda(\delta_a,\delta_b).(cond(lt,\delta_b,\delta_a \square [\![x\leftarrow x-y]\!], id))\downarrow 1.$$

It is straightforward to verify that the definitions of f, h and g can be achieved by suitably interpreting lt, id, □, cond and [[x←x-y]]. As an example $cond(\delta_1,\delta_2,\delta_3)$ will be $\lambda s.\delta_1(s)\rightarrow\delta_2(s),\delta_3(s)$ in the standard semantics. In the remainder of this section some of the expressions will be surveyed but it is impractical to cover all aspects.

All expressions have a formal type ct. It makes sense to require such types to be closed, i.e. not to contain any free domain variables. Since ct includes rt→rt' it is possible to have expressions that denote state transformations. The constants lt (for less than) and id (for identity) are two such examples. The only restriction imposed upon constants is that their type must be contravariantly pure [Nie 84]: essentially this means that each ct'→ct" occurring in the type must satisfy that ct' does not contain a type ft. The need for this restriction will become clear in section 4 in the context of transforming constants from the standard semantics and to the collecting semantics.

The constructs whose types are not contravariantly pure are FIX and a few functionals: these are functions whose type is of the form $ft^n\rightarrow ft$. One functional is composition

$$\square: (rt'\rightarrow rt")\times(rt\rightarrow rt')\rightarrow(rt\rightarrow rt").$$

Another is conditional

cond: $(rt \underline{\rightarrow} \underline{T}) \underline{\times} (rt \underline{\rightarrow} rt') \underline{\times} (rt \underline{\rightarrow} rt') \underline{\rightarrow} (rt \underline{\rightarrow} rt')$

where $\underline{T}$ is the type of the run-time truth values. This differs from the conditional ...→...,... of type T×ct×ct→ct that was used in section 2. However, both conditionals will be used in the metalanguage: the former conditional expresses a run-time test whereas the latter will be used to express a compile-time test.

The distinction between syntax for compile-time operations and run-time operations permeate all of the metalanguage. Related to the compile-time cartesian product we have notation $(e_1,\ldots,e_k)$ and e↓i for forming tuples and selecting components. Related to the run-time product $\underline{\times}$ the analogous notations are

tuple: $(rt \underline{\rightarrow} rt_1) \underline{\times} \ldots \underline{\times} (rt \underline{\rightarrow} rt_k) \underline{\rightarrow} (rt \underline{\rightarrow} rt_1 \underline{\times} \ldots \underline{\times} rt_k)$

$take_i$: $rt_1 \underline{\times} \ldots \underline{\times} rt_k \underline{\rightarrow} rt_i$.

In the standard semantics one has $tuple(f_1,\ldots,f_k)(v)=(f_1(v),\ldots,f_k(v))$ and $take_i(v)=v\downarrow i$. The use of the functional tuple and constant $take_i$ is illustrated by the definition

$[\![x \leftarrow x-y]\!] = tuple(sub,take_2)$

where sub: $\underline{Z \times Z \rightarrow Z}$ is a constant subtraction function. Turning to compile-time sum + we have injection in_i, projection out_i and test is_i. For run-time sum $\underline{+}$ we have

case: $(rt_1 \underline{\rightarrow} rt) \underline{\times} \ldots \underline{\times} (rt_k \underline{\rightarrow} rt) \underline{\rightarrow} (rt_1 \underline{+} \ldots \underline{+} rt_k \underline{\rightarrow} rt)$

in_i: $rt_i \underline{\rightarrow} rt_1 \underline{+} \ldots \underline{+} rt_k$.

In the standard semantics we will have $case(f_1,\ldots,f_k)(in_i(v))=f_i(v)$. We refer to [NiNi 85b] for a discussion of the pragmatic aspects of the notation.

4. STANDARD AND COLLECTING SEMANTICS

An interpretation of the primitives of the metalanguage amounts to defining the meaning of types and expressions. We begin with defining the standard semantics $\underline{S}$ upon the run-time types rt. The programme is to follow the categorical approach of [SmPl 82] rather than the universal domain approach. The presentation needs to be rather terse so consult [ArMa 75] for general categorical concepts and [SmPl 82] for special notions. (A detailed development is given in [Nie 84].) The main idea is to define the semantics as a locally continuous covariant functor

$\underline{S}[\![rt]\!] : \underline{\underline{ACCs}}^N \underline{\rightarrow} \underline{\underline{ACCs}}$

where N is the number of free domain variables in rt. The category $\underline{\underline{ACCs}}$ has as objects the (ω-) algebraic consistently complete (ω-) cpo's and as morphisms the strict (ω-) continuous functions.

The definition of $\underline{S}[\![rt]\!]$ is by induction on the structure of rt. So $\underline{S}[\![\underline{A}_i]\!]$ is defined to be a constant functor and $\underline{S}[\![\underline{X}_i]\!]$ is a projection functor. If $rt_1,\ldots,rt_k$ all have the same free domain variables we get

$$\underline{S}[\![rt_1 \underline{+} \ldots \underline{+} rt_k]\!] = +\cdot(\underline{S}[\![rt_1]\!],\ldots,\underline{S}[\![rt_k]\!])$$

where + is the coalesced sum functor, $\cdot$ is composition of functors and $(\ldots,\ldots,\ldots)$ denotes tupling of functors. We shall write $\underline{S}(\underline{+})=+$ as a record of this. The use of coalesced sum is motivated by the isomorphism $S_\perp+S_\perp \tilde{=} (S \uplus S)_\perp$ where $\uplus$ is disjoint union and $(\ )_\perp$ constructs a flat cpo. Similarly $rt_1 \underline{\times} \ldots \underline{\times} rt_k$ is handled by using $\underline{S}(\underline{\times})=*$ which is the smash product functor. If differs from cartesian product in that all tuples with a $\perp$ component are identified and is motivated by $S_\perp * S_\perp \tilde{=} (S \times S)_\perp$. When interpreting $\underline{S}(\underline{\rightarrow})$ as the strict continuous function space constructor $\rightarrow_s$ one then obtains e.g.

$$S_\perp+S_\perp \rightarrow_s S_\perp * S_\perp \tilde{=} S \uplus S \rightarrow S \times S$$

so that the formal definition of the standard semantics $\underline{S}$ is close to the "naive" definition in section 2.

Finally we consider the recursive domain $\underline{\text{rec}}\ \underline{X}_1.rt$ and suppose that $\underline{X}_1$ and $\underline{X}_2$ are the only free domain variables or rt. Let $\underline{S}[\![rt]\!]^0$ be $\underline{S}[\![\underline{X}_1]\!]$ and let $\underline{S}[\![rt]\!]^{n+1}$ be $\underline{S}[\![rt]\!]^n \cdot (S[\![rt]\!]^n, \underline{S}[\![\underline{X}_2]\!])$. We shall feel free to write $\underline{S}[\![rt^n(\underline{X}_1)]\!]$ for $\underline{S}[\![rt]\!]^n$ where $rt^0(\underline{X}_1)=\underline{X}_1$ and $rt^{n+1}(\underline{X}_1)=rt[rt^n(\underline{X}_1)/\underline{X}_1]$. To define $\underline{S}[\![\underline{\text{rec}}\ \underline{X}_1.rt]\!]$ we must define its effect upon an object D and a morphism $f: D \rightarrow D'$. Let (r_n,E) be the limiting cone for the chain $(\underline{S}[\![rt]\!]^n(U,D), \underline{S}[\![rt]\!]^n(\perp,id))$ and similarly (r_n',E'). As usual the r_n and r_n' are embeddings with upper adjoints r_n^u and $r_n'^u$, i.e. they are morphisms of the subcategory $\underline{\underline{\text{ACCe}}}$. Then

$$\underline{S}[\![\underline{\text{rec}}\ \underline{X}_1.rt]\!](D)=E$$

$$S[\![\underline{\text{rec}}\ \underline{X}_1.rt]\!](f)=\bigsqcup_n r_n' \cdot \underline{S}[\![rt]\!]^n(\perp,f) \cdot r_n^u$$

completes the definition.

Before proceeding with the definition of $\underline{S}$ let us consider the definition of the collecting semantics $\underline{C}$ upon the run-time types rt. The example in section 2 used powersets and the corresponding analogue within domain theory is the so-called relational powerdomain P_R. This is a locally continuous and covariant functor from $\underline{\underline{\text{ACCs}}}$ to the category $\underline{\underline{\text{ACLs}}}$ of algebraic complete lattices and strict continuous functions. It is defined by

$$P_R(D)=(\{Y \underline{\subseteq} B_D \mid Y \neq \emptyset \wedge Y=LC(Y)\}, \underline{\subseteq})$$

$$P_R(f)=\lambda Y.\{e \mid \exists d \in Y:\ e \underline{\sqsubseteq} f(d)\}$$

where B_D is the set of finite elements of D and

$$LC(Y) = \{d' \in B_D \mid \exists d \in Y:\ d' \underline{\sqsubseteq} d\}.$$

The motivation for using $\mathcal{P}_R$ rather than e.g. the Smyth or Plotkin powerdomains is that $\mathcal{P}_R(S_\perp)\simeq\mathcal{P}(S)$ so that the development is still close to section 2. It is then natural to define $\underline{C}[\![\underline{A}_i]\!]$ as $\mathcal{P}_R\cdot\underline{S}[\![\underline{A}_i]\!]$ and to let $\underline{C}[\![\underline{X}]\!]$ be a projection functor. For $\underline{C}(+)$ we shall use cartesian product $\times$ because $\mathcal{P}_R(D_1)\times\ldots\times\mathcal{P}_R(D_k)\simeq\mathcal{P}_R(D_1+\ldots+D_k)$. For $\underline{C}(\times)$ we shall use the tensor product $\circledast$. For k=2 it is defined by

$$L\circledast M = (L\to_{as} M^{op})^{op}$$

$$f\circledast g = \lambda h.\lambda l'.\bigsqcup\{g(h(l))\,|\,l'\sqsubseteq f(l)\}$$

where "as" means strict, additive and continuous and $(I\sqsubseteq)^{op}=(I\sqsupseteq)$. The motivation for this choice is that $\mathcal{P}_R(D)\circledast\mathcal{P}_R(E)\simeq\mathcal{P}_R(D*E)$. Unfortunately $\circledast$ is not a functor from $\underline{\underline{ACLs}}^k$ to $\underline{\underline{ACLs}}$ because the composition law

$$\circledast(f\cdot f',g\cdot g')=\circledast(f,g)\cdot\circledast(f',g')$$

may fail. In general we have only

$$\circledast(f\cdot f',g\cdot g')\sqsubseteq\circledast(f,g)\cdot\circledast(f',g').$$

However, equality holds whenever f and g are additionally additive and then $\circledast(f,g)$ is also additive. We shall use the term semifunctor for such a mapping. A semi-functor specializes to a covariant functor over the category $\underline{\underline{ACLe}}$. This means that $\underline{C}[\![\underline{\text{rec}}\ \underline{X}.rt]\!]$ can be defined analogously to $\underline{S}[\![\underline{\text{rec}}\ \underline{X}.rt]\!]$. In summary we obtain a locally continuous semi-functor $\underline{C}[\![rt]\!] : \underline{\underline{ACLs}}^N\to\underline{\underline{ACLs}}$.

The motivational remarks in the definition of $\underline{C}[\![rt]\!]$ have already hinted at the following connection between the functionalities in the standard and collecting semantics.

Lemma 1. For a closed type rt the object $\mathcal{P}_R(\underline{S}[\![rt]\!])$ is isomorphic to the object $\underline{C}[\![rt]\!]$. □

Proof. The proof proceeds by structural induction and this means that it will be necessary to consider types that are not closed. The inductive hypothesis therefore considers a type rt with free domain variables $\underline{X}_1,\ldots,\underline{X}_n$ and asserts that:

there is a natural equivalence nat_{rt}

from $\mathcal{P}_R\cdot\underline{S}[\![rt]\!]$

to $\underline{C}[\![rt]\!]\cdot(\mathcal{P}_R\cdot\underline{S}[\![\underline{X}_1]\!],\ldots,\mathcal{P}_R\cdot\underline{S}[\![\underline{X}_N]\!])$.

The proof also gives a structural definition of nat_{rt}. The details may be found in [Nie 84, Theorem 3.3:4]. □

When rt is closed we shall write nat_{rt} for the isomorphism from the object $\mathcal{P}_R(\underline{S}[\![rt]\!])$ to the object $\underline{C}[\![rt]\!]$.

We now continue the definition of the standard semantics $\underline{S}$ upon types. It would be natural to define $\underline{S}[\![ct]\!]$ as a locally continuous and covariant functor over the category CPOs of cpo's and strict continuous functions. However, the contravariance of the function space construction motivates considering so-called symmetric functors over another category CPO2s. In this category an object is a cpo, a morphism $f:A\to B$ is a pair $(f' :A\to B, f'':B\to A)$ of strict continuous functions and composition is given by $(f',f'')\cdot(g',g'') = (f'\cdot g', g''\cdot f'')$. A symmetric functor (domain functor [Rey 74]) $F:\text{CPO2s}^N\to\text{CPO2s}$ is a covariant functor that satisfies

$$F(f_1,\ldots,f_N)^R = F(f_1^R,\ldots,f_N^R)$$

where $(f',f'')^R = (f'',f')$.

The definition of $\underline{S}[\![ct]\!] : \text{CPO2s}^N\to\text{CPO2s}$ as a locally continuous symmetric functor (abbreviated l.c.s. functor) then has $\underline{S}[\![A_i]\!]$ to be the constant functor over the appropriate cpo and $\underline{S}[\![X]\!]$ to be a projection functor. All $rt\to rt'$ considered will be closed so $\underline{S}[\![rt\to rt']\!]$ may be defined to be the constant functor over the cpo of strict continuous functions from $\underline{S}[\![rt]\!]$ to $\underline{S}[\![rt']\!]$. For $\underline{S}(+)$ we use the symmetric variant of coalesced sum +. It is defined as + upon objects but as

$$\underline{S}(+)((f_1',f_1''),\ldots,(f_N',f_N'')) = (f_1'+\ldots+f_N', f_1''+\ldots+f_N'')$$

upon morphisms. Similarly $\underline{S}(\times)$ is the symmetric variant of cartesian product $\times$. The symmetric functor $\underline{S}(\to)$ is defined as $\to$ upon objects but as

$$S(\to)((f_1',f_1''),(f_2',f_2'')) = (f_1''\to f_2', f_1'\to f_2'')$$

upon morphisms. (In a sense the contravariance has been hidden in the composition of CPO2s.) Since embeddings have unique upper adjoints it is still possible to specialize a l.c.s. functor to a suitable functor over CPOe. This means that recursive domains may be solved much as before and furthermore the solution may be turned into a l.c.s. functor.

The l.c.s. functor $\underline{C}[\![ct]\!]$ is defined analogously to $\underline{S}[\![ct]\!]$. The difference between $\underline{S}[\![rt]\!]$ and $\underline{C}[\![rt]\!]$ was motivated by the different perceptions of run-time entities in the standard and collecting semantics. There seems to be no reason for why compile-time entities should be perceived as differently and therefore the only difference in the definition is that $\underline{C}[\![rt\to rt']\!]$ corresponds to the cpo of strict continuous functions from $\underline{C}[\![rt]\!]$ to $\underline{C}[\![rt']\!]$.

To relate $\underline{S}[\![ct]\!]$ and $\underline{C}[\![ct]\!]$ we define a "relation" sim_{ct}. It has the form

$$sim_{ct}(Q_1,\ldots,Q_N): \underline{S}[\![ct]\!](D_1,\ldots,D_N)\times\underline{C}[\![ct]\!](L_1,\ldots,L_N)\to\{true,false\}$$

where $Q_i: D_i\times L_i\to\{true,false\}$ is a relation corresponding to the free domain variable X_i of ct. When ct is $rt\to rt'$ the relation is true upon (d,l) iff

$$l = nat_{rt'}\cdot P_R(d)\cdot nat_{rt}^{-1}$$

as should be expected given Lemma 1 and the development in section 2. When $ct=A_i$ we require that $l=d$ and when $ct=X_i$ we require that $Q_i(l,d)$. In the remaining cases the definition is "componentwise". So when $ct=ct_1\times\ldots\times ct_k$ it is

$$\forall i:\ sim_{ct_i}(Q_1,\ldots,Q_N)(d\downarrow i,l\downarrow i)$$

and when $ct=ct'\to ct''$ it is the so-called logical relation

$$\forall(u,v):\ sim_{ct'}(Q_1,\ldots,Q_N)(u,v)\Rightarrow sim_{ct''}(Q_1,\ldots,Q_N)(d(u),l(v)).$$

When $ct = rec\ X_{N+1}.ct'$ the definition roughly is

$$\forall n:\ sim_{ct'^{n}(X_{N+1})}(Q_1,\ldots,Q_N,\lambda(d',l').true)(r_n^u(d),s_n^u(l))$$

for appropriate embeddings r_n and s_n.

The definition of sim_{ct} fits well with the definition of the functors $\underline{S}[\![\ ct]\!]$ and $\underline{C}[\![\ ct]\!]$. To make this precise define a category $\underline{\underline{SIM}}$ as follows. An object is a triple (D,Q,L) where D and L are cpo's and $Q: D\times L\to\{true,false\}$ is an admissible relation (i.e. is true of $(\bot,\bot)$ and truth is preserved when taking least upper bounds of chains). A morphism from (D,Q,L) to (D',Q',L') is a pair of $\underline{\underline{CPO2s}}$ morphisms

$$((f',f''):D\to D',(g',g''):L\to L')$$

fulfilling that

$$Q(d,l)\Rightarrow Q'(f'(d),g'(l))$$
$$Q(f''(d'),g''(l''))\Leftarrow Q'(d',l').$$

Composition is defined componentwise. Next define a functor-like mapping

$$\underline{SC}[\![\ ct]\!]:\ \underline{\underline{SIM}}^N\to\underline{\underline{SIM}}\ \text{by}$$

$$\underline{SC}[\![\ ct]\!]\ ((D_1,Q_1,L_1),\ldots)=(\underline{S}[\![\ ct]\!]\ (D_1,\ldots),sim_{ct}(Q_1,\ldots),\underline{C}[\![\ ct]\!]\ (L_1,\ldots))$$

$$\underline{SC}[\![\ ct]\!]\ ((f_1,g_1),\ldots)=(\underline{S}[\![\ ct]\!]\ (f_1,\ldots),\underline{C}[\![\ ct]\!]\ (g_1,\ldots)).$$

We then have (in analogy with the relational functors of [Rey 74]).

Lemma 2. $\underline{SC}[\![\ ct]\!]$ is a locally continuous covariant functor over $\underline{\underline{SIM}}$. □

Proof. The proof is by structural induction on ct. Full details are given in [Nie 84, Theorem 3.3:13] and the discussion after that theorem. □

We are now ready to consider the semantics of expressions. The definition of $\underline{S}[\![\ e]\!]$ is by structural induction on e. The notation pertaining to compile-time types ct is rather standard and is interpreted as usual, so for example FIX gives the least fixed point. For the notation pertaining to run-time types rt the explanatory remarks in section 3 indicate what their standard semantics is.

The definition of $\underline{C}[\![e]\!]$ corresponds to that of $\underline{S}[\![e]\!]$ as long as no run-time types are involved. Furthermore, FIX is interpreted so as to give the least fixed point. For a constant f of contravariantly pure type ct define the type ct' to be ct with all $rt_i \underline{\rightarrow} rt_i'$ replaced by X_i. The contravariantly purity of ct enforces that $\underline{S}[\![ct']\!]$ $(=\underline{C}[\![ct']\!])$ is the symmetric variant of a covariant functor $F_{ct}: \underline{\underline{CPOs}}^{N} \rightarrow \underline{\underline{CPOs}}$. It is then natural to define

$$\underline{C}[\![f]\!] = F_{ct}(\lambda d_i.nat_{rt_i'} \cdot P_R(d_i) \cdot nat_{rt_i}^{-1})(\underline{S}[\![f]\!]).$$

When ct=rt$\underline{\rightarrow}$rt' this gives the expected $\underline{C}[\![f]\!] = nat_{rt'} \cdot P_R(\underline{S}[\![f]\!]) \cdot nat_{rt}^{-1}$. This method is not applicable to functionals.

When considering the functionals it is instructive to assume that the isomorphisms in Lemma 1 are the identities (but see [Nie 84] for the general case). Composition is straightforward

$$\underline{C}[\![e \square e']\!] = \underline{C}[\![e]\!] \cdot \underline{C}[\![e']\!] .$$

Building on the development of section 2 it is natural to put +

$$\underline{C}[\![cond(e_1,e_2,e_3)]\!] = \underline{C}[\![e_2]\!] \cdot true(\underline{C}[\![e_1]\!]) \cup \underline{C}[\![e_3]\!] \cdot false(\underline{C}[\![e_1]\!]).$$

Here $\cup$ means "set-union pointwise",

$$true(g) = \lambda Y.LC(\{d \in Y \mid g(LC(\{d\})) \ni true\})$$

and false(g) similarly. Recalling that $\underline{C}(+) = \times$ we shall use

$$\underline{C}[\![case\ e_1,\ldots,e_k]\!] = \lambda(Y_1,\ldots,Y_k).\underline{C}[\![e_1]\!](Y_1) \cup \ldots \cup \underline{C}[\![e_k]\!](Y_k).$$

Finally,

$$\underline{C}[\![tuple\ e_1,\ldots,e_k]\!] = \lambda Y.\cup\{\underline{C}[\![e_1]\!](LC(\{y\})) * \ldots * \underline{C}[\![e_k]\!](LC(\{y\})) \mid y \in Y\}.$$

We cannot just use $\lambda Y.\underline{C}[\![e_1]\!](Y) * \ldots * \underline{C}[\![e_k]\!](Y)$ because then $\underline{C}[\![tuple\ id,id]\!]$ (LC({2,3})) gives LC({(2,2),(2,3),(3,2),(3,3)}) rather than the desired LC({(2,2),(3,3)}).

The correctness of these definitions is guaranteed by

Theorem 1. For all expressions e of type ct we have $sim_{ct}(\underline{S}[\![e]\!], \underline{C}[\![e]\!])$. □

We shall omit the proof (but see [Nie 84, Theorem 3.3:14]) which is by structural induction on e using Lemma 2. When ct=rt$\underline{\rightarrow}$rt' and nat... are the identities the Theorem asserts that

$$\underline{C}[\![e]\!](Y) = LC(\{\underline{S}[\![e]\!](d) \mid d \in Y\}).$$

This is the analogue within domain theory of the connection expressed in section 2.

+ This agrees with [Nie 84] because here $\underline{C}[\![e_2]\!]$ and $\underline{C}[\![e_3]\!]$ are strict.

3. ABSTRACT INTERPRETATION

The main design principle underlying the metalanguage is that data flow analyses should be obtainable by interpreting the primitives adequately. However, not all interpretations may meaningfully be viewed as specifications of data flow analyses. We shall therefore define an approximating interpretation $\underline{I}$ to be a specification of:

- constant functors $\underline{I}(\underline{A}_1),\ldots$ over ACLs; these correspond to the approximations desired in the particular data flow analysis,
- locally continuous semifunctors $\underline{I}(+)$ and $\underline{I}(\times)$ over ACLs; these correspond to the data flow analysis methods used (e.g. relational method),
- constant functions $\underline{I}(f_1),\ldots$ of contravariantly pure type,
- functionals $\underline{I}(\text{cond})$, $\underline{I}(\text{tuple}),\ldots$

The collecting interpretation $\underline{C}$ is an approximating interpretation but the standard interpretation $\underline{S}$ is not (e.g. $\underline{S}(+)$ does not map into ACLs). The use of ACLs means that the sets of data flow values (like L in section 2) must be complete lattices and this is a common assumption. The additional assumptions of algebraicity of objects and strictness and continuity of morphisms is a technical convenience that hardly limits the applicability of the framework.

Consider now how to relate two data flow analyses described by approximating interpretations $\underline{I}$ and $\underline{J}$. (A natural choice for $\underline{I}$ might be $\underline{C}$.) The first task is to relate the complete lattices $\underline{I}[\![\,\text{rt}\,]\!]$ and $\underline{J}[\![\,\text{rt}\,]\!]$ for closed types rt. Motivated by section 2 we use a morphism

$$\gamma_{\text{rt}}: \underline{J}[\![\,\text{rt}\,]\!] \to \underline{I}[\![\,\text{rt}\,]\!]$$

of ACLs to describe the meaning of elements of $\underline{J}[\![\,\text{rt}\,]\!]$ in terms of those of $\underline{I}[\![\,\text{rt}\,]\!]$. In keeping with section 2 we shall require that γ_{rt} is an upper adjoint (relative to ACLs). This means that there must be a (necessarily unique) morphism (of ACLs)

$$\alpha_{\text{rt}}: \underline{I}[\![\,\text{rt}\,]\!] \to \underline{J}[\![\,\text{rt}\,]\!]$$

called a lower adjoint such that $\alpha_{\text{rt}}(l) \sqsubseteq m$ iff $l \sqsubseteq \gamma_{\text{rt}}(m)$. It now remains to give a structural definition of $(\alpha_{\text{rt}}, \gamma_{\text{rt}})$.

We shall use section 2 as a guide in this. So let $\text{rt}=\underline{A}\times\underline{A}$ with $\underline{I}(\underline{A})=\mathcal{P}(Z)$, $\underline{J}(A)=L$ and $\gamma_{\underline{A}}=\overline{\gamma}$. Suppose first that $\underline{I}(\times)=\underline{J}(\times)=\times$. It is then natural to define γ_{rt} as the componentwise application of $\overline{\gamma}$ so that e.g. $\gamma_{\text{rt}}(\text{pos},\text{neg})$ is $(\{z \mid z>0\},\{z \mid z<0\})$. This suggests the general definition

$$\gamma_{\text{rt}} = \underline{J}(\times)(\overline{\gamma},\overline{\gamma}).$$

Suppose next that $\underline{I}(\times)=\otimes$ (as defined in section 2) whereas $\underline{J}(\times)=\times$.