PROCEEDINGS

Southern African Computer Symposium

6th de

1991

Rekendaarsimposium van Suider Afrika

1991

DE OVERBERGER HOTEL,
CALEDON

2 - 3 JULY 1991

SPONSORED BY

ISM
FRD
GENMIN

EDITED BY
M H Linck

DEPARTMENT OF COMPUTER SCIENCE • UNIVERSITY OF CAPE TOWN
PROCEEDINGS / KONGRESOPSPOMMINGS

6th
SOUTHERN AFRICAN COMPUTER SYMPOSIUM

6de
SUIDELIKE-AFRIKAANSE REKENAARSSIMPOSIUM

De Overberger Hotel, Caledon
2 - 3 JULY 1991

SPONSORED by
ISM
FRD
GENMIN

EDITED by
M H LINCK
Department of Computer Science
University of Cape Town
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>1</td>
</tr>
<tr>
<td>Organising Committee</td>
<td>2</td>
</tr>
<tr>
<td>Referees</td>
<td>3</td>
</tr>
<tr>
<td>Program</td>
<td>5</td>
</tr>
<tr>
<td>Papers (In order of presentation)</td>
<td>9</td>
</tr>
<tr>
<td>&quot;A value can belong to many types&quot;</td>
<td></td>
</tr>
<tr>
<td>B H Venter, University of Fort Hare</td>
<td>10</td>
</tr>
<tr>
<td>&quot;A Transputer Based Embedded Controller Development System&quot;</td>
<td></td>
</tr>
<tr>
<td>M R Webster, R G Harley, D C Levy &amp; D R Woodward, University of Natal</td>
<td>16</td>
</tr>
<tr>
<td>&quot;Improving a Control and Sequencing Language&quot;</td>
<td></td>
</tr>
<tr>
<td>G Smit &amp; C Fair, University of Cape Town</td>
<td>25</td>
</tr>
<tr>
<td>&quot;Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers&quot;</td>
<td>40</td>
</tr>
<tr>
<td>P Machanick, University of Witwatersrand</td>
<td></td>
</tr>
<tr>
<td>&quot;Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface&quot;</td>
<td>51</td>
</tr>
<tr>
<td>L van Zijl &amp; D Mitton, University of Stellenbosch</td>
<td></td>
</tr>
<tr>
<td>&quot;Product Form Solutions for Multiserver Centres with Hierarchical Classes of Customers&quot;</td>
<td>69</td>
</tr>
<tr>
<td>A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig</td>
<td></td>
</tr>
<tr>
<td>&quot;A Reusable Kernel for the Development of Control Software&quot;</td>
<td>83</td>
</tr>
<tr>
<td>W Fouché and P de Villiers, University of Stellenbosch</td>
<td></td>
</tr>
<tr>
<td>&quot;An Implementation of Linda Tuple Space under the Helios Operating System&quot;</td>
<td>95</td>
</tr>
<tr>
<td>P G Clayton, E P Wentworth, G C Wells and F de Heer-Menlah, Rhodes University</td>
<td></td>
</tr>
<tr>
<td>&quot;The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System&quot;</td>
<td>107</td>
</tr>
<tr>
<td>K Macgregor, University of Cape Town &amp; R Campbell University of Illinois at Urbana-Champaign</td>
<td></td>
</tr>
</tbody>
</table>
"Concurrency Control Mechanisms for Multidatabase Systems"
A Deacon, University of Stellenbosch 118

"Extending Local Recovery Techniques for Distributed Databases"
H L Victor & M H Rennhackkamp, University of Stellenbosch 135

"Analysing Routing Strategies in Sporadic Networks"
S Melville, University of Natal 148

The Design of a Speech Synthesis System for Afrikaans"
M J Wagener, University of Port Elizabeth 167

"Expert Systems for Management Control: A Multiexpert Architecture"
V Ram, University of Natal 177

"Integrating Similarity-Based and Explanation-Based Learning"
G D Oosthuizen and C Avenant, University of Pretoria 187

"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town 201

"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch 211

"Building a secure database using self-protecting objects"
M Olivier and S H von Solms, Rand Afrikaans University 228

"Modelling the Algebra of Weakest Preconditions"
C Brink and I Rewitsky, University of Cape Town 242

"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch. 262

"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal 276
FOREWORD

The 6th Computer Symposium, organised under the auspices of SAICS, carries on the tradition of providing an opportunity for the South African scientific computing community to present research material to their peers.

It was heartening that 31 papers were offered for consideration. As before all these papers were refereed. Thereafter a selection committee chose 21 for presentation at the Symposium.

Several new dimensions are present in the 1991 symposium:

* The Symposium has been arranged for the day immediately after the SACLA conference.

* It is being run over only 1 day in contrast to the 2-3 days of previous symposia.

* I believe that it is first time that a Symposium has been held outside of the Transvaal.

* Over 85 people will be attending. Nearly all will have attended both events.

* A Sponsorship package for both SACLA and the Research Symposium was obtained. (This led to reduced hotel costs compared to previous symposia)

A major expense is the production of the Proceedings of the Symposium. To ensure financial soundness authors have had to pay the page charge of R20 per page.

A thought for the future would be consideration of a poster session at the Symposium. This could provide an alternative approach to presenting ideas or work.

I would sincerely hope that the twinning of SACLA and the Research Symposium is considered successful enough for this combination survive. As to whether a Research Symposium should be run each year after SACLA, or only every second year, is a matter of need and taste.

A challenge for the future is to encourage an even greater number of MSc & PhD students to attend the Symposium. Unlike this year, I would recommend that they be accommodated at the same cost as everyone else. Only if it is financially necessary should the sponsored number of students be limited.

I would like to thank the other members of the organising committee and my colleagues at UCT for all the help that they have given me. A special word of thanks goes to Prof. Pieter Kritzinger who has provided me with invaluable help and ideas throughout the organisation of this 6th Research Symposium.

M H Linck
Symposium Chairman
SYMPOSIUM CHAIRMAN

M H Linck, University of Cape Town

ORGANISING COMMITTEE

D Kourie, Pretoria University.
P S Kritzinger, University of Cape Town.
M H Linck, University of Cape Town.

SPONSORS

ISM
GENMIN
FRD
<table>
<thead>
<tr>
<th>NAME</th>
<th>INSTITUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnard, E</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Becker, Ronnie</td>
<td>UCT</td>
</tr>
<tr>
<td>Berman S</td>
<td>UCT</td>
</tr>
<tr>
<td>Bishop, Judy</td>
<td>Wits</td>
</tr>
<tr>
<td>Berman, Sonia</td>
<td>UCT</td>
</tr>
<tr>
<td>Brink, Chris</td>
<td>UCT</td>
</tr>
<tr>
<td>Bodde, Ryn</td>
<td>Networks Systems</td>
</tr>
<tr>
<td>Bornman, Chris</td>
<td>UNISA</td>
</tr>
<tr>
<td>Bruwer, Piet</td>
<td>UOFS</td>
</tr>
<tr>
<td>Cherenack, Paul</td>
<td>UCT</td>
</tr>
<tr>
<td>Cook Donald</td>
<td>UCT</td>
</tr>
<tr>
<td>de Jaeger, Gerhard</td>
<td>UCT</td>
</tr>
<tr>
<td>de Villiers, Pieter</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Ehlers, Elize</td>
<td>RAU</td>
</tr>
<tr>
<td>Eloff, Jan</td>
<td>RAU</td>
</tr>
<tr>
<td>Finnie, Gavin</td>
<td>Natal</td>
</tr>
<tr>
<td>Gaynor, N</td>
<td>AECI</td>
</tr>
<tr>
<td>Hutchinson, Andrew</td>
<td>UCT</td>
</tr>
<tr>
<td>Jourdan, D</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Kourie Derrick</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Kritzinger, Pieter</td>
<td>UCT</td>
</tr>
<tr>
<td>Krzesinski, Tony</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Laing, Doug</td>
<td>ISM</td>
</tr>
<tr>
<td>Labuschagne, Willem</td>
<td>UNISA</td>
</tr>
<tr>
<td>Levy, Dave</td>
<td>Natal</td>
</tr>
</tbody>
</table>
MacGregor, Ken UCT
Machanick, Philip Wits
Mattison Keith UCT
Messerschmidt, Hans UOFS
Mutch, Laurie Shell
Neishlos, N Wits
Oosthuizen, Deon Pretoria
Peters Joseph Simon Fraser
Ram, V Natal, Pmb.
Postma, Stef Natal, Pmb
Rennhackkamp, Martin Stellenbòsch
Shochot, John Wits
Silverberg, Roger Council for Mineral Technology
Smit, Riel UCT
Smith, Dereck UCT
Terry, Pat Rhodes
van den Heever, Roelf UP
van Zijl, Lynette Stellenbosch
Venter, Herman Fort Hare
Victor, Hema Stellenbosch
von Solms, Basie RAU
Wagenaar, M UPE
Wentworth, Peter Rhodes
Wheeler, Graham UCT
Wood, Peter UCT
6TH RESEARCH SYMPOSIUM - 1991

FINAL PROGRAM

TUESDAY 2nd July 1991

10h00 - 13h00  Registration
13h00 - 13h50  PUB LUNCH

14h00 - 15h30  SESSION 1A
Venue: Hassner
Chairman: Prof Basie von Solms
14h00 - 14h30  "A value can belong to many types."  B H Venter, University of Fort Hare
14h30 - 15h00  "A Transputer Based Embedded Controller Development System"  M R Webster, R G Harley, D C Levy & D R Woodward, University of Natal
15h00 - 15h30  "Improving a Control and Sequencing Language"  G Smit and C Fair, University of Cape Town

SESSION 1B
Venue: Hassner C
Chairman: Prof Roelf v d Heever
14h00 - 14h30  "Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers"  P Machanick, University of Witwatersrand
14h30 - 15h00  "Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface"  L van Zijl & D Mitton, University of Stellenbosch
15h00 - 15h30  "Product Form Solutions for Multiserver Centres with Hierarchical Classes of Customers"  A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig

15h30 - 16h00  TEA
16h00 - 17h30 SESSION 2A

Venue: Hassner

Chairman: Prof Derrick Kourie

16h00 - 16h30
"A Reusable Kernel for the Development of Control Software" W Fouché and P de Villiers, University of Stellenbosch

16h30 - 17h00
"An Implementation of Linda Tuple Space under the Helios Operating System" P G Clayton, E P Wentworth, G C Wells and F de-Heer-Menlah, Rhodes University

17h00 - 17h30
"The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System" K MacGregor, University of Cape Town & R Campbell, University of Illinois at Urbana-Champaign

19h30 PRE-DINNER DRINKS

20h00 GALA CAPE DINNER
(Men: Jackets & ties)
WEDNESDAY 3rd July 1991

7h00 - 8h15  BREAKFAST

8h15 - 9h45  SESSION 3A
Venue: Hassner
Chairman: Assoc Prof P Wood

8h15 - 8h45
"Concurrency Control Mechanisms for Multidatabase Systems" A Deacon,
University of Stellenbosch

8h45 - 9h15
"Extending Local Recovery Techniques for Distributed Databases" H L Victor
& M H Rennhackkamp, University of Stellenbosch

9h15 - 9h45
"Analysing Routing Strategies in Sporadic Networks" S Melville,
University of Natal

9h45 - 10h15  TEA

10h15 - 11h00  SESSION 4
Venue: Hassner
Chairman: Prof P S Kritzinger
Invited paper: E Coffman

11h00 - 11h10  BREAK
11h10 - 12h40 SESSION 5A

Venue: Hassner
Chairman: Prof C Bornman

11h10 - 11h40
"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town

11h40 - 12h10
"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch

12h10 - 12h40
"Building a secure database using self-protecting objects" M Olivier and S H von Solms, Rand Afrikaans University

SESSION 5B

Venue: Hassner C
Chairman: Prof A Krzesinski

11h10 - 11h40
"Modelling the Algebra of Weakest Preconditions"
C Brink & I Rewitsky, University of Cape Town

11h40 - 12h10
"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch

12h10 - 12h40
"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal

12h45-12h55 GENERAL MEETING of RESEARCH SYMPOSIUM ATTENDEES

Venue: Hassner
Chairman: Dr M H Linck

13h00 - 14h00 LUNCH

FINIS 6th COMPUTER SYMPOSIUM
PAPERS

of the

6TH RESEARCH SYMPOSIUM
A value can belong to many types

B H Venter
Department of Computer Science, University of Fort Hare
Private Bag X1314, Alice, Republic of Ciskei, Southern Africa
e-mail: cssbv1@uhzl.ufh.ac.za

Abstract

Most statically typed Algol-style languages were designed to conform with the view that a value belongs to one and only one type. The adoption of this view bought compiler simplicity at the expense of language expressivity. This trade off has been partially offset by the introduction of subtypes. The adoption of the view that types may intersect in arbitrary ways, on the other hand, requires complicated compilers or costly run-time type checks. However, language expressivity is enhanced, language semantics are simplified and object-oriented and database concepts can be integrated smoothly with the Algol style. Keywords: Data abstraction, object-oriented programming, polymorphism, type checking

Computing Reviews Categories: D.3.3

1 Introduction

In his influential early paper on data types[4], Hoare states:

Every value belongs to one and only one type.

It is evident from Algol-style languages like Ada and Modula that many programming language designs conform with this view.

The adoption of this view bought compiler simplicity at the expense of language expressivity — a reasonable trade off then, but not now. In recent languages, such as Oberon, some of the lost expressivity is regained by the introduction of subtypes[13].

It is shown in section 10 that, in real life, types intersect in ways that are difficult to model using only subtyping. A closer fit between reality and model can be achieved if types are allowed to intersect in arbitrary ways. The aim of the paper is to examine some of the consequences and to suggest how arbitrary intersections can be constructed.

2 Types as sets

If one views a type as merely a set of values, type intersection is straightforward to define and understand (it is merely set intersection). If one adopts the classical view that a type is an algebra[8], however, it is much more difficult to define the meaning of intersection.

The author contends that it is not essential to view a type as an algebra. Types are useful because they provide structure and allow type checking. This can be done even if one views types merely as sets of values. In the rest of this paper, therefore, types are sets of values and \( \mathcal{U} \) will denote the set of all the values that can belong to a type.

The author contends further that in programming languages types play only one role: they serve as constraints on the domains and codomains of functions. That is, if \( \mathcal{D} \) and \( \mathcal{C} \) are types, and \( f : \mathcal{U} \rightarrow \mathcal{U} \) is a function with domain constraint \( \mathcal{D} \) and codomain constraint \( \mathcal{C} \), then all programming languages require the definition of \( f \) to conform to the following:

\[
\begin{align*}
f(z) &= \text{Undefined}, & \text{if } z \notin \mathcal{D} \\
f(z) &\in \mathcal{C} \cup \{\text{Undefined}\} & \text{if } z \in \mathcal{D}
\end{align*}
\]

Typically, an error is signaled when \( f(z) \) returns Undefined (not be confused with non-termination or Bottom). In statically typed languages such errors are sometimes predictable by the compiler.

As a syntactic convenience, we will write \( f : \mathcal{D} \rightarrow \mathcal{C} \) to show that \( f : \mathcal{U} \rightarrow \mathcal{U} \) is a function with domain constraint \( \mathcal{D} \) and codomain constraint \( \mathcal{C} \).

The reader may be forgiven for being sceptical about the contention that types only serve to constrain functions. After all, types are used to declare variables and check assignments. What's that got to do with functions?

Everything, as long as we regard variables as functions whose definitions are provided and updated by assignment statements. In other words, a variable declaration such as
V : Some-type
must be regarded as syntactic sugar for

\( v : \{ \} \rightarrow \text{Some-type} \) is a function whose
evaluation rule will be supplied later on
by an assignment statement and may
be changed (as time progresses) by fur­
ther assignment statements.

We will furthermore regard V to stand for \( v() \)
when used in an expression (\( v \) can be denoted
as Address(V)).

3 Related work

Value sharing (type intersection) has received
scant attention from researchers working on type
systems. For example, Cardelli(2] mentions it in
passing as a special case of parametric polymor­
phism but gives no explicit consideration to it.
Language designers either dispense altogether
with the notion of typed variables (for exam­
ple Icon[5] and SETL[9]) or make every effort to
restrict value sharing.

As far as the author knows, the work of Bailes
on G[1] has been the sole exception to this
trend. Bailes, however, arrives at value sharing
as a necessary consequence of his well-motivated
overall approach and gives little specific atten­
tion to it.

4 Type equivalence

The question of when type A is equivalent to
type B has been difficult to resolve using other
views of what a type is. If, however, types are
viewed as sets that merely serve as domain and
codomain constraints on functions, there is only
one sensible answer: type A is equivalent to type
B, if and only if the same set of values make up
both type A and type B.

Thus, if we declare

\[
\text{type } A = \text{array} 1..10 \text{ of Int} \\
\text{type } B = \text{array} 0..9 \text{ of Int}
\]

we can determine whether types A and B are
equivalent merely by determining whether set A
is equal to set B.

Of course, the difficult question now becomes
the interpretation to be given to type construc­
tors. Most type constructors have straightfor­
ward interpretations, as can be seen from the
following examples:

\[
\text{set of } T = \{ S \mid S \subseteq T \} \\
\text{array } R \text{ of } T = \{ f \mid f : R \rightarrow T \} \\
\text{pointer to } T = \{ f \mid f : \{ \} \rightarrow T \}
\]

We thus can readily conclude that types A
and B above are not equivalent since \( f \neq g \) for
all \( f : (1\ldots10) \rightarrow \text{Int} \) and \( g : (0\ldots9) \rightarrow \text{Int} \).
But what about the following types?

\[
\text{type } C = \text{record } x : \text{Int}; y : \text{Char} \text{ end} \\
\text{type } D = \text{record } x : \text{Int}; y : \text{Char} \text{ end}
\]

The question should be familiar to partici­
pants in the name equivalence versus structural
equivalence debate. By interpreting types as
sets, the question is transformed, however, from
"Are types C and D equivalent?" into "How
does one characterize the set of values repre­
sented by a record constructor?". Neither ques­
tion has an answer that will satisfy everyone.

Given our convention of interpreting a vari­
able name, such as \( R \), as syntactic sugar for \( r() \),
it seems reasonable to regard \( R.x \) as syntactic
sugar for \( r()() \) and \( S.x \) as syntactic sugar for
\( s()() \), if \( R \) is of type C and \( S \) of type D. This
requires the following interpretation of type C
as a set:

\[
\{ f \mid f : (x_C, y_C) \rightarrow \text{Int} \cup \text{Char} \}
\text{and } f(x_C) \in \text{Int and } f(y_C) \in \text{Char}
\]

Thus, if \( x_C = x_D \) and \( y_C = y_D \) then types C
and D are equivalent. We now have two sensible
choices:

1. \( x_C = x_D = 'x' \) and \( y_C = y_D = 'y' \).
2. \( x_C, y_C, x_D \) and \( y_D \) are distinct values.

Selecting the first choice amounts to a form of
structural equivalence that allows the conclusion
that type E, below, is also equivalent to types C
and D.

\[
\text{type } E = \text{record } y : \text{Char}; x : \text{Int} \text{ end}
\]

Selecting the second choice amounts to name
equivalence.

5 Assignment compatibility

By our earlier definition, a variable is a function
and the value returned by this function must be
a member of the set \( T \cup \{ \text{Undefined} \} \) where \( T \)
is the type of the variable. When an expression
is assigned to a variable, it is thus necessary to
ensure that the expression results in a member
of $T \cup \{\text{Undefined}\}$ before allowing the assignment to take effect\(^1\).

Since operators are also functions, expressions can be regarded as function applications. The type of an expression is thus the range set of the function that is being applied. This is guaranteed to be a subset of $C \cup \{\text{Undefined}\}$ where $C$ is the codomain of the function.

Now, if $C \subseteq T$ or $C \cap T = \emptyset$, the validity of the assignment can be decided without evaluating the expression. Thus, a language designer can ensure that all type checks can be done by a compiler, simply by insisting that assignments are illegal unless $C \subseteq T$.

Compile-time (or static) type checking is generally regarded as highly desirable and hence many language designers claim that their languages allow static type checking. In practice, however, static type checking is so restrictive that no realistic language enforces it. Virtually all languages allow assignments to be carried out if $C \supseteq T$ and generate run-time checks to ensure that the assignments are legal. The main difference from language to language is whether the language forces the programmer to make the type check explicit or not.

If types are to intersect \textit{freely}, however, it is not reasonable to insist that $C \supseteq T$. Assignments should be allowed, subject to run-time checks if necessary, whenever $C \cap T \neq \emptyset$.\(^2\)

Of course, this means that many type errors that are easily detected by the compilers of languages like Ada and Oberon, become very difficult for a compiler to detect, thus requiring the generation of expensive run-time checks. The author contends, however, that it is no longer appropriate to restrict the expressivity of a language merely to limit the complexity of compilers and run-time systems. \textit{Such a strategy does not get rid of the complexity}, it merely dumps it over the fence in the application programmer's yard. Furthermore, much progress is being made on the optimization of "dynamic" languages (for example [3]) and it seems reasonable to expect that such compilers will convert most run-time errors into compile-time errors.

That is, while static languages may be safer than dynamic languages at present, they are not inherently safer.

6 Problems with records

Although the second interpretation for record constructors (amounting to name-equivalence) has been dubbed sensible in section 4, it does not seem sensible to adopt this convention in a language where types are regarded merely as sets, since the obvious interpretations of the sets of values generated by the other type constructors all amount to structural equivalence.

Structural equivalence, however, creates problems when records are used as the "concrete" representations of abstract types. For example, if

\begin{verbatim}
  type Stack =
    record
      Top : range 0..100;
      Elems : array 1..100 of Int;
    end record
\end{verbatim}

and

\begin{verbatim}
  type Funny-rec =
    record
      Top : range 0..100;
      Elems : array 1..100 of Int;
    end record
\end{verbatim}

then types Stack and Funny-rec represent the \textit{same} set of values if structural equivalence is used.

Given assignment compatibility as outlined in the previous section, this means that a client programmer can always make the "hidden" representation of abstract types visible by providing alternative type definitions that happen to describe the same sets of values as the abstract types. Thus, abstract types have to be handled in non-traditional ways if types are to intersect freely.

7 Types as values

One intuitively expects to be able to declare variables of type "set of something". If one allows such types, it implies that one allows sets to be elements of sets, which implies that one ought to allow types to be values\(^3\).

Of course, if a programming language allows very large sets, like Int, to be regarded as values, one usually assigns \text{Undefined} to a variable will also result in a run-time error.

\(^2\)This is already the case for subrange types in most languages. The argument here is that it should apply to other types as well.

\(^3\)If $\mathcal{U}$ is treated as a value, it becomes possible to specify paradoxical sets, for example $\{S \subseteq \mathcal{U} \mid S \notin S\}$. To avoid paradoxes we decree that $\mathcal{U} \in \mathcal{U}$ and $\mathcal{U} \subseteq \mathcal{U}$ by definition and that all paradoxical set formers must result in \text{Undefined}.

---

\(^1\)Usually, assigning \text{Undefined} to a variable will also result in a run-time error.

---

12
the language implementations must resort to a form of lazy evaluation: one would not wish the compiler to generate code that fully evaluates a set like \( \{ x \in \text{Int} \mid \text{odd}(x) \} \) if the set is only used as the target of a members test.

8 Dynamic Types

Most programming languages require all the values making up a type to be determinable during compilation, since this is a prerequisite for static type checking. If one drops the insistence that type checks should be static where possible and very simple otherwise, it becomes possible to consider whether sets of values, that vary at run-time, should be allowed to serve as types.

Such “dynamic” types would be very useful in database-oriented languages. For example, one would be able to declare:

```
Parts : set of \( \mathcal{U} \)
Suppliers : set of \( \mathcal{U} \)
Parts-supplied : Suppliers - set of Parts
```

Since Suppliers is a variable, values of type Suppliers can be created and destroyed as time progresses, simply by including and excluding values from set Suppliers. Moreover, the domain constraint of function Parts-supplied dynamically adjusts to the creation and destruction of supplier values. (In other words, the type system takes care of referential integrity.)

Although any sets of values can make up types Parts and Suppliers, it is better not to press numeric, string, array or record values into service as part or supplier values. Parts and Suppliers are abstract data types and a means must be provided to populate them with abstract values. In other words, it must be made possible to get hold of values that belong to \( \mathcal{U} \), but to no other set. Providing a built-in function called new comes to mind.

By making it possible to create new values at run-time and to include these values into one or more run-time sets that can serve as types, one makes it possible for types to intersect in arbitrary ways. With classical sub typing, the set of types to which a newly created value can belong can be determined by the compiler and types can only intersect if the one is a subtype of the other or if both types share a common subtype.

9 Object Classes

Dynamic typing subsumes sub typing and provide for the integration of object-oriented concepts and Algol-style languages in a natural way. For example

```
Suppliers : set of \( \mathcal{U} \)
```

can be seen as the declaration of an object class and

```
Parts-supplied : Suppliers - set of Parts
```

can be seen as the declaration of a member variable for class Suppliers. Dynamically creating new abstract values and including them in \( \text{set} \) Suppliers amounts to dynamically creating objects of class Suppliers.

Derived classes correspond to subtypes, and inheritance follows naturally. For example, given type (class) \( \mathcal{A} \), an operator (method) \( f : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A} \), and a subtype (derived class) \( \mathcal{B} \subset \mathcal{A} \), then \( f \) may obviously be applied to arguments of type \( \mathcal{B} \). Moreover, the value resulting from the application of \( f \) to arguments of type \( \mathcal{B} \) may well be a member of \( \mathcal{B} \) and thus may be assigned to a variable of type \( \mathcal{B} \), subject to a “run-time” type check. In other words, subtype (derived class) \( \mathcal{B} \) effectively inherits operator (method) \( f \) from supertype (super class) \( \mathcal{A} \).

Inherited methods can be overridden by allowing operators (methods) to be overloaded. However, operator/function names cannot be as freely overloaded as in a language like Ada. For example, if

```
f : \mathcal{A} \rightarrow \mathcal{C}, f : \mathcal{B} \rightarrow \mathcal{C}, \text{and } \mathcal{A} \cap \mathcal{B} = \{ ab \}
```

then it is impossible to determine which \( f \) must be used when one encounters \( f(ab) \).

Similar difficulties arise in object-oriented languages that allow multiple inheritance and similar solutions can be adopted.

10 Beyond Classes

Object-oriented programming languages are currently advocated by many as the long awaited solution to the software crises. For the most part, the author agrees that these high expectations are justified. However, it should not be imagined that these languages have advanced programming language expressivity to the natural limits.
For example, consider an object-oriented design for a university database. It seems likely that the designer will have introduced a class Person with sub classes Student and Staff-member. The usual interpretation of classes as types require Student and Staff-member to be disjoint. In real life, however, these sets are not disjoint and the designer thus will have to go further.

In simple cases, multiple inheritance seems the answer. If the designer adds a Student-staff-member class, which inherits from both Student and Staff-member, to the database schema the problem seems solved: An instance of Student-staff-member ISA Student and ISA Staff-member, the latter two classes are thus no longer disjoint.

Consider, however, the complications that arise when using this technique to model the real life intersections between classes Student, Staff-member, Parent and Sports-club-member. Must the database designer anticipate the need for the class Sporting-parent-student-staff-member? Surely not!

Consider further, the complications that arise if the designer decides to introduce sub classes Academic-staff-member, Administrative-staff-member, Technical-staff-member, and so on. In this case, creating an instance of Student-staff-member creates a staff member object that belongs to none of the sub classes of Staff-member. The designer thus is forced to remove class Student-staff-member from the schema and replace it with Student-academic-staff-member, Student-administrative-staff-member, and so on.

The author suspects that the database designer will simply give up and provide a schema that is an oversimplification of the real world. The application programmer ends up having to solve the problem, typically by adding explicit run-time type checks all over his code.

If the designer can use dynamic types that can intersect in arbitrary ways, the above modeling problems simply do not arise.

11 Summary

The consequences of completely abandoning the view that every value belongs to one and only one type are profound. One can achieve no more than a partial abandonment, however, unless one also holds that a type is simply a set of values and allow any set of values to serve in the role of a type.

The interpretation of a type as no more than a set of values forces one to conclude that type constructors are simply set operators. This in turn simplifies the resolution of questions about type equivalence and suggests that structural equivalence is appropriate.

Structural equivalence leads to problems when applied to record constructors. Records have also proved troublesome to database designers[6] and have been successfully done away with in DAPLEX[10]. This suggests that records are not "nice" constructs and that "post-Hoare" programming languages might better off without them.

Another consequence of regarding types as nothing more than sets of values, is that operators take on an independent existence and that the notion of operator inheritance by sub-types thus becomes simple. Also, types can be made into first-class values by treating sets as first-class values. This is more intuitive than other approaches[7], but requires implementers to tackle lazy evaluation. Finally, polymorphism and the ability to do type membership tests are automatic consequences of making types equivalent to sets.

In classical Algol-style languages the prevalence of disjoint types and highly restricted mechanisms for introducing subtypes make "nearly" static type checking possible. When any set of values can serve as a type, however, it must be accepted that non-trivial run-time type checks will have to be performed in some situations — typically those situations where the application programmer would have to provide them if the language does not.

Acceptance of non-trivial run-time type checking makes it reasonable to accept furthermore that, in some cases, even the values making up a type will not be known at compile-time. This leads to the very useful concept of a dynamic type that makes it possible to support abstract types in a non-trADitional, more natural way, and to integrate concepts from database languages and object-oriented languages into Algol-style languages.

The author has recently designed an Algol-style language that supports and exploits unrestricted type intersection. A prototype version has been in use since early 1989 and our experiences with its expressivity are very positive. The design of the type system of the language is re-
ported in more detail in [11] and the language itself is defined (in under 20 pages) in [12].

The author has little doubt that the active exploration of the design space of languages that allow and exploit unrestricted type intersection will result in some very exciting and powerful new programming languages appearing on the scene.

12 Conclusion

If one assumes that the "software crisis" will not allow restrictions on the expressivity of programming languages to survive advances in compiler technology, it is also safe to assume that statically typed languages like Ada, C++, Eiffel and so on have no future.

The author believes that the future belongs to persistent, polymorphic, dynamically typed, object-oriented languages that allow types to intersect freely. This article has shown that these need not be radical languages that are as different from Algol family as LISP, Prolog or Smalltalk are, but can just as well be Algol-style languages.

References


