Combining Static and Dynamic Types

Ronald Garcia

Gradual Typing

Dynamic Typing:
(Agility)

Static Typing:
(Robustness)

Siek and Taha 2006,2007

Gradual Typing

Dynamic Typing:
(Agility)

Static Typing:
(Robustness)

Siek and Taha 2006,2007

Gradual Typing

```
move(x, dx) {
  return x + dx;
}

p = 0;
a = 1;
x2 = move(p,a);
```

Gradual Typing

```
move(int x, dx) {
  return x + dx;
}

int p = 0;
a = 1;
x2 = move(p,a);
```

Gradual Typing

```
int move(int x, dx) {
  return x + dx;
}

int p = 0;
a = 1;
int x2 = move(p,a);
```

Gradual Typing

```
int move(int x, int dx) {
  return x + dx;
}

int p = 0;
int a = 1;
int x2 = move(p,a);
```

Gradual Typing

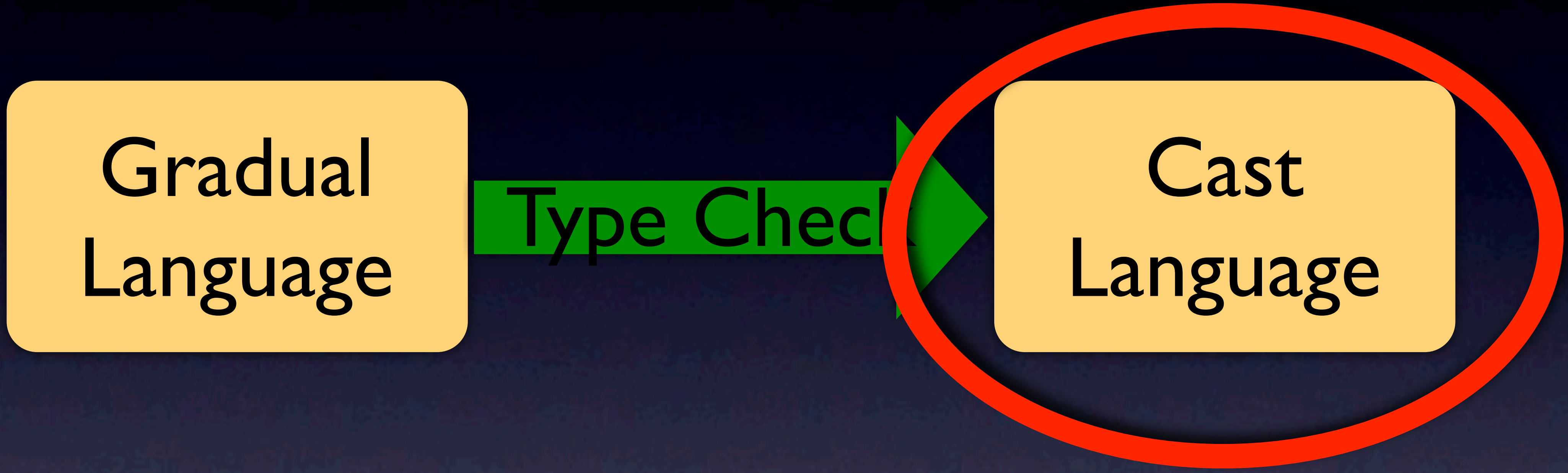
```
if (SomethingTrue()) {
  x = 7.0
} else {
  x = “Good Grief!”
}
y = sqrt(x)
```

From GT to Casts

Gradual Language

Type Check

Cast Language



Gradual Language

Type Check

Cast Language

Cast-based Languages

Dyn

- type of dynamic values

$\langle T \Leftarrow S \rangle e$

- Expression to cast S values to T values

First-Order Casts

$$\langle \mathit{Int} \Leftarrow \mathit{Dyn} \rangle \langle \mathit{Dyn} \Leftarrow \mathit{Int} \rangle 4 \longmapsto^{*} 4$$

$$\langle \mathit{Bool} \Leftarrow \mathit{Dyn} \rangle \langle \mathit{Dyn} \Leftarrow \mathit{Int} \rangle 4 \longmapsto^{*} \mathbf{error}$$

Higher-Order Casts

$$\langle Int \to Int \Leftarrow Int \to Dyn \rangle\ f$$

$$\begin{array}{ll}
 & (\lambda g.\ g\ 1)\ (\langle Int \to Int \Leftarrow Int \to Dyn \rangle f) \\
\longmapsto & (\langle Int \to Int \Leftarrow Int \to Dyn \rangle f)\ 1 \\
\longmapsto & \langle Int \Leftarrow Dyn \rangle (f\ \langle Int \Leftarrow Int \rangle 1) \\
\longmapsto & \langle Int \Leftarrow Dyn \rangle (f\ 1)
\end{array}$$

Findler and Felleisen, 2002

Design Space of Casts

- Two Axes of Design Space
  - How eagerly should we detect errors
  - Which expressions deserve attention
- Low-level implementation semantics

Siek, Garcia, and Taha, ESOP 2009

How Eagerly?

$$\langle \texttt{Bool} \to \texttt{Int} \Leftarrow \texttt{Dyn} \rangle \\ \quad \langle \texttt{Dyn} \Leftarrow \texttt{Int} \to \texttt{Int} \rangle (\lambda x : \texttt{Int}.\ x)$$

- *Lazy Semantics* accept this expression
- *Eager Semantics* reports a cast error.

Who's to Blame?

$$
\begin{array}{l}
(\langle \texttt{Dyn} \rightarrow \texttt{Int} \Leftarrow \texttt{Dyn} \rangle^{l_3} \\
\quad \langle \texttt{Dyn} \Leftarrow \texttt{Bool} \rightarrow \texttt{Bool} \rangle^{l_2} \lambda x : \texttt{Bool}.\, x) \\
\qquad \langle \texttt{Dyn} \Leftarrow \texttt{Int} \rangle^{l_1} 1
\end{array}
$$

- *UD Strategy* blames *the context around* the cast with label $l2$
- *D Strategy* blames *the cast* with label $l3$

$$\begin{array}{l}\langle \texttt{Dyn} \rightarrow \texttt{Int} \Leftarrow \texttt{Dyn}\rangle^{l_3} \\ \qquad \langle \texttt{Dyn} \Leftarrow \texttt{Bool} \rightarrow \texttt{Bool}\rangle^{l_2} \lambda x : \texttt{Bool}.\ x \longmapsto \\ \langle \texttt{Dyn} \rightarrow \texttt{Int} \Leftarrow \texttt{Bool} \rightarrow \texttt{Bool}\rangle^{l_3} \lambda x : \texttt{Bool}.\ x\end{array}$$

- *D strategy*: casts to and from Dyn are explicitly type-tagged operations

$$
\begin{array}{l}
\langle \mathtt{Dyn} \to \mathtt{Int} \Leftarrow \mathtt{Dyn} \rangle^{l_3} \\
\qquad \langle \mathtt{Dyn} \Leftarrow \mathtt{Bool} \to \mathtt{Bool} \rangle^{l_2} \lambda x : \mathtt{Bool}.\ x \longmapsto \\
\langle \mathtt{Dyn} \to \mathtt{Int} \Leftarrow \mathtt{Dyn} \to \mathtt{Dyn} \rangle^{l_3} \\
\qquad \langle \mathtt{Dyn} \to \mathtt{Dyn} \Leftarrow \mathtt{Bool} \to \mathtt{Bool} \rangle^{l_2} \lambda x : \mathtt{Bool}.\ x
\end{array}
$$

- *UD strategy*: casts to and from Dyn are encoded as recursive types

Who's to Blame?

Theorem (Blame Safety)

Let e be a well-typed term with subterm $\langle T \Leftarrow S \rangle^{l} e'$ containing the only occurrences of label l in e.
*If $S <: T$ then $e \not\longrightarrow^{*}$ **blame** l.*

Who's to Blame?

Theorem (Blame Safety)

Let e be a well-typed term with subterm $\langle T \Leftarrow S\rangle^{l} e'$ *containing the only occurrences of label l in e.*
If $S <: T$ *then* $e \not\longrightarrow^{*}$ **blame** l.

- Tells us which casts cannot be blamed
- Doesn't tell us which casts SHOULD be blamed.

Design Space

Lazy UD*	Lazy D
Eager UD	Eager D

* Wadler and Findler 2009

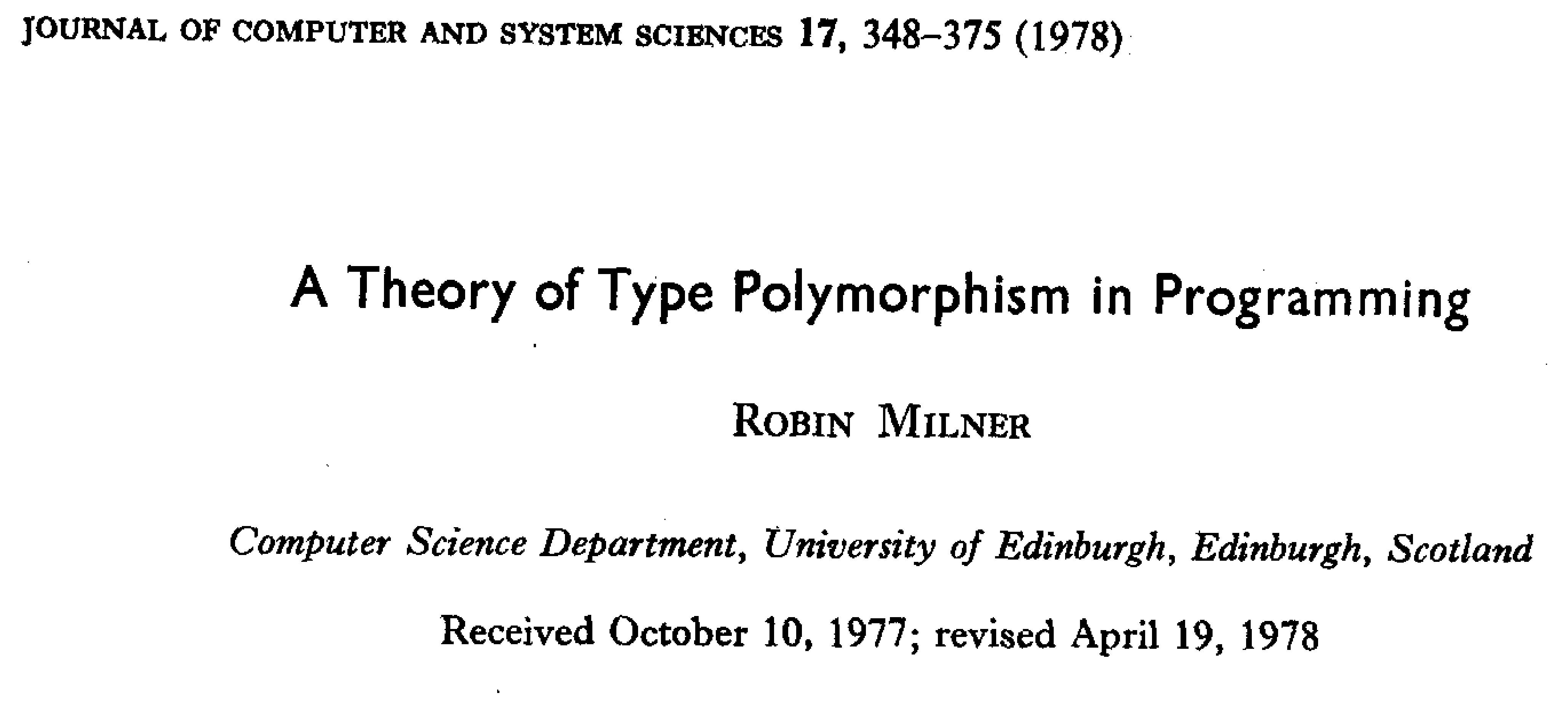
Recent Developments

- High-Level Semantics for Casts*
  - D semantics subsume UD semantics.
  - Eager D and UD semantics are subtle.
- Implementation Strategy
  - Threesomes for the full design space

*Joint work with Jeremy Siek

Desirable Properties

- Reasonable Performance
- Error Detection
- Error Reporting
- Full Language



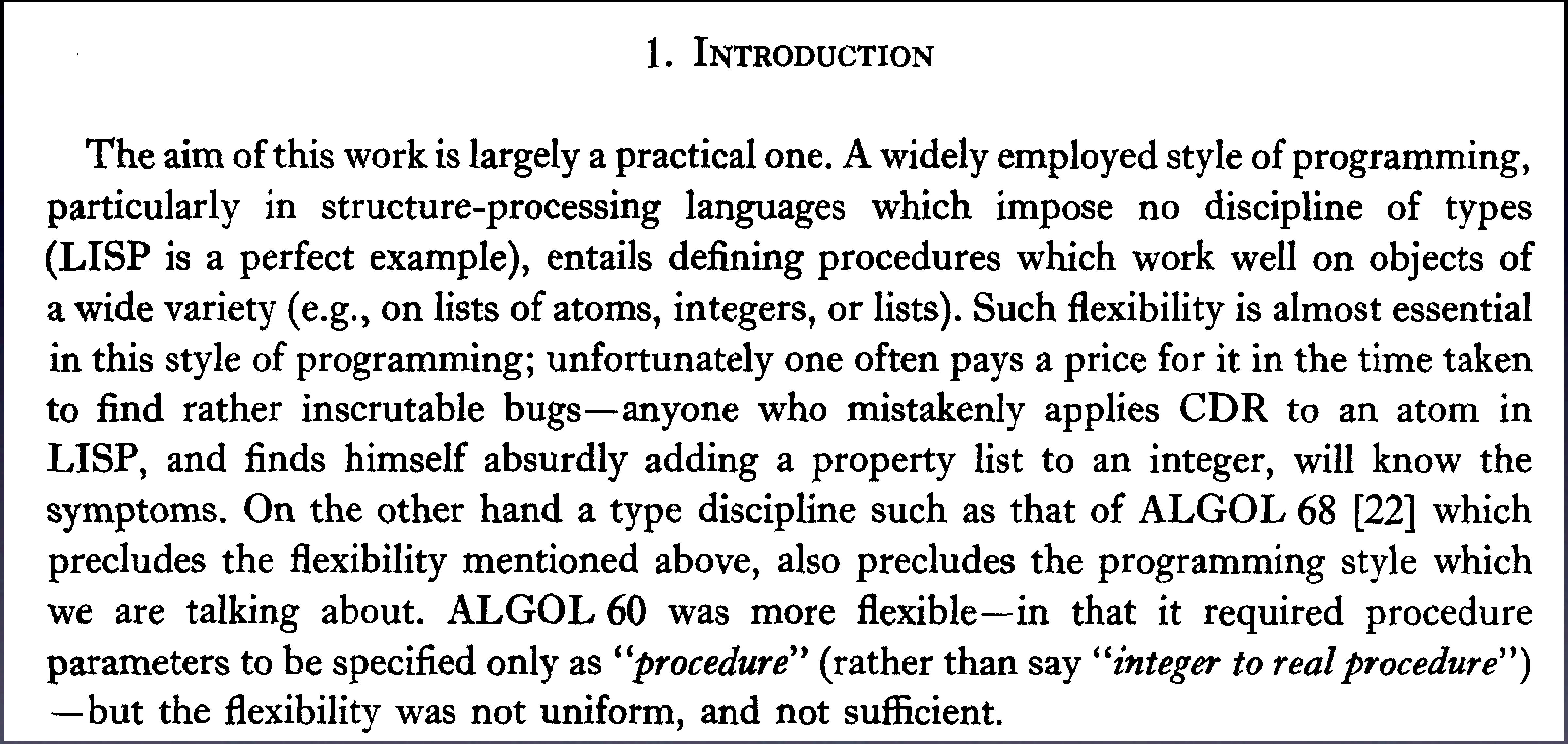
JOURNAL OF COMPUTER AND SYSTEM SCIENCES **17**, 348–375 (1978)

A Theory of Type Polymorphism in Programming

ROBIN MILNER

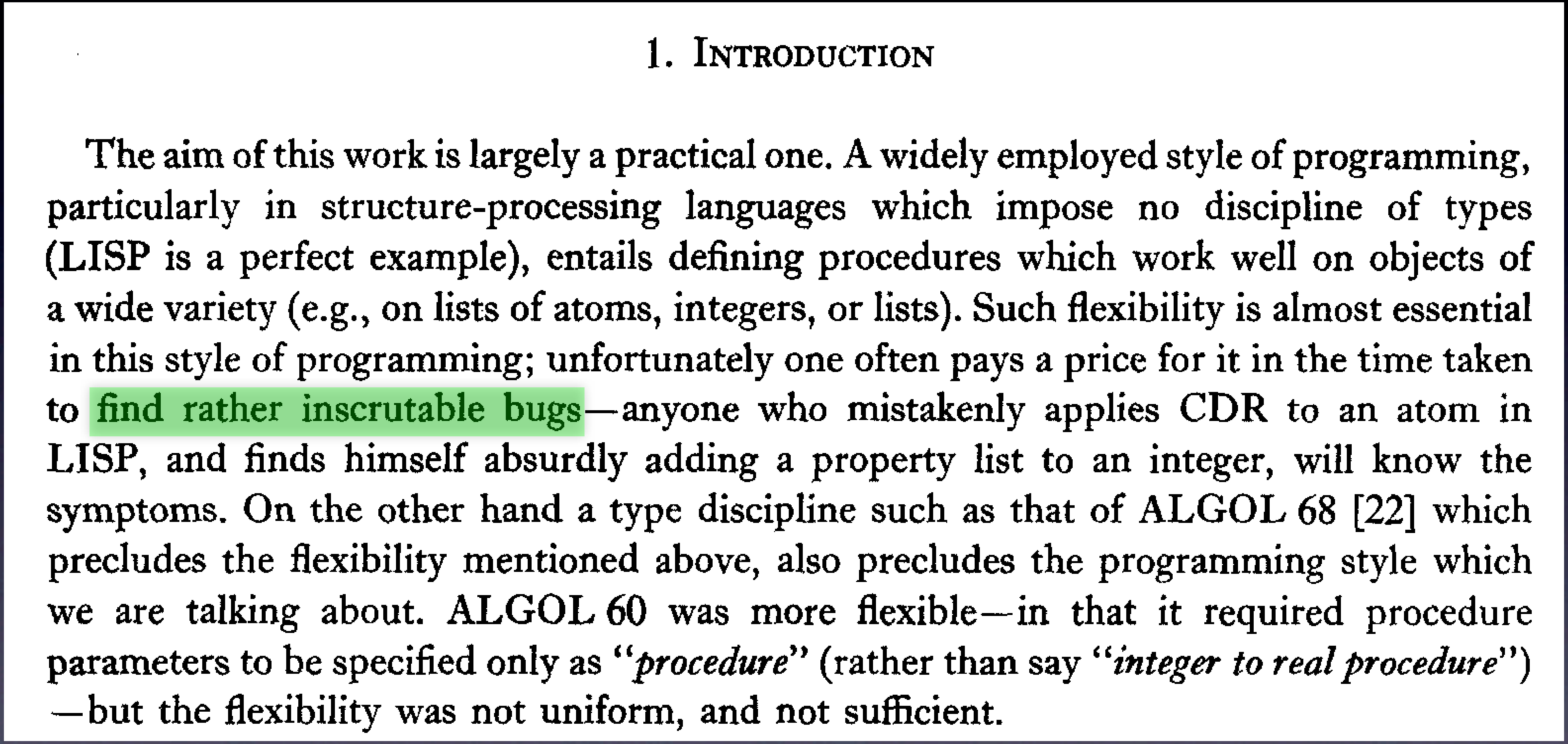
Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978



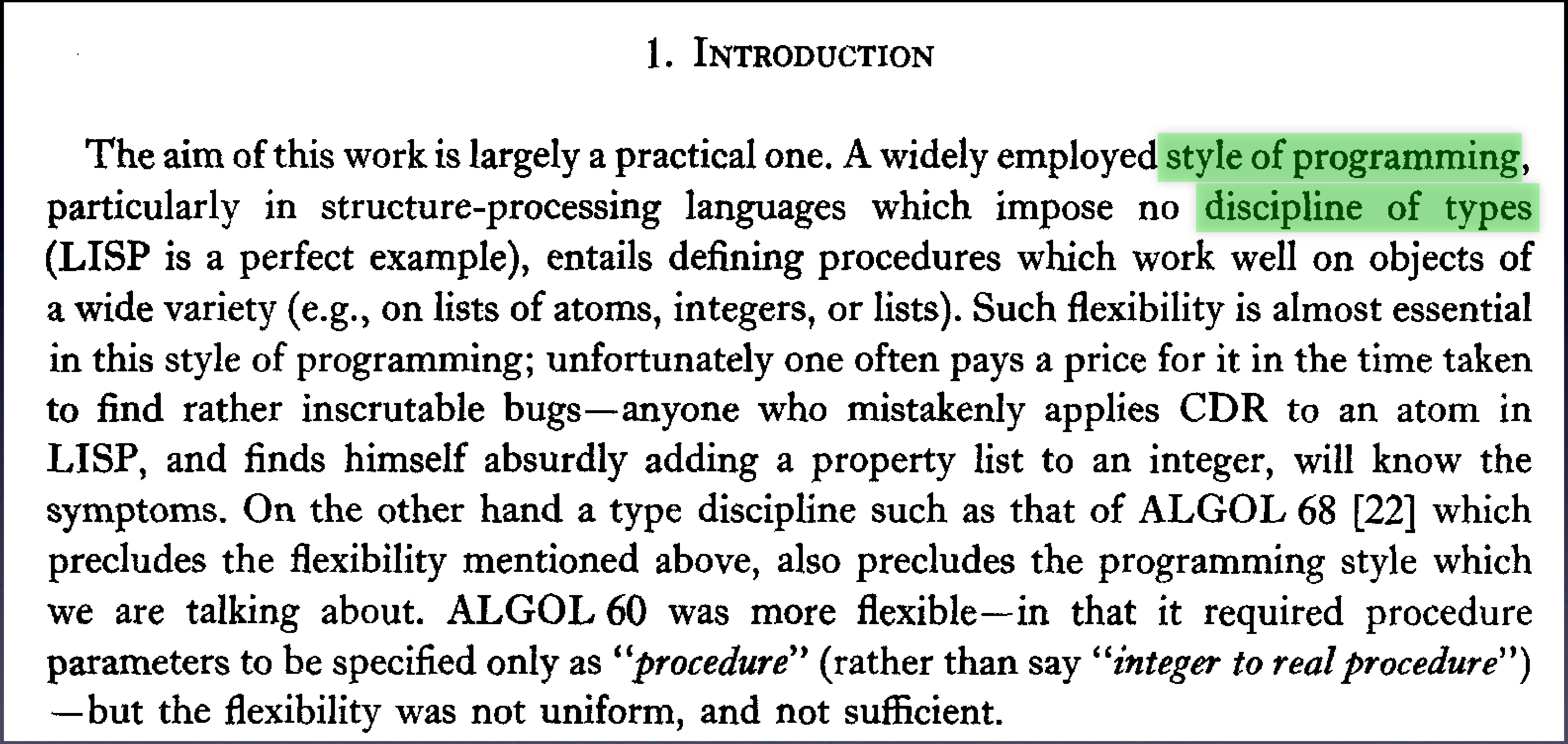
1. Introduction

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types (LISP is a perfect example), entails defining procedures which work well on objects of a wide variety (e.g., on lists of atoms, integers, or lists). Such flexibility is almost essential in this style of programming; unfortunately one often pays a price for it in the time taken to find rather inscrutable bugs—anyone who mistakenly applies CDR to an atom in LISP, and finds himself absurdly adding a property list to an integer, will know the symptoms. On the other hand a type discipline such as that of ALGOL 68 [22] which precludes the flexibility mentioned above, also precludes the programming style which we are talking about. ALGOL 60 was more flexible—in that it required procedure parameters to be specified only as "*procedure*" (rather than say "*integer to real procedure*") —but the flexibility was not uniform, and not sufficient.



1. INTRODUCTION

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types (LISP is a perfect example), entails defining procedures which work well on objects of a wide variety (e.g., on lists of atoms, integers, or lists). Such flexibility is almost essential in this style of programming; unfortunately one often pays a price for it in the time taken to find rather inscrutable bugs—anyone who mistakenly applies CDR to an atom in LISP, and finds himself absurdly adding a property list to an integer, will know the symptoms. On the other hand a type discipline such as that of ALGOL 68 [22] which precludes the flexibility mentioned above, also precludes the programming style which we are talking about. ALGOL 60 was more flexible—in that it required procedure parameters to be specified only as "*procedure*" (rather than say "*integer to real procedure*") —but the flexibility was not uniform, and not sufficient.



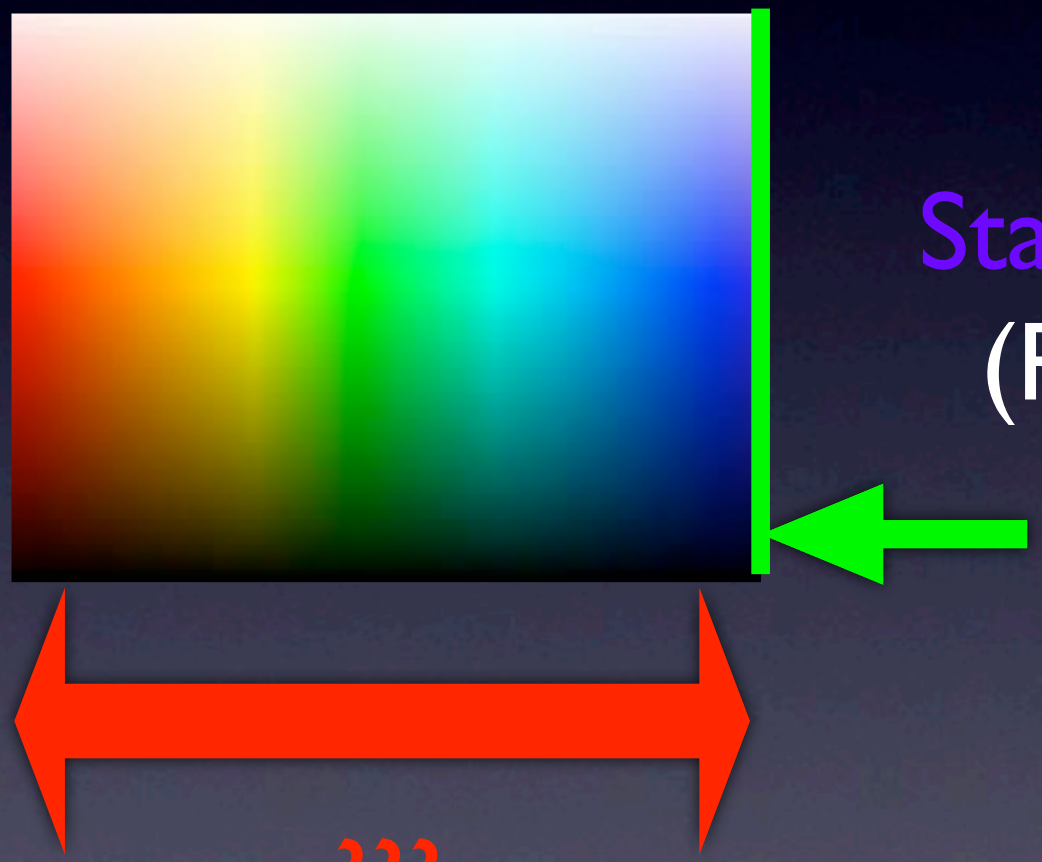
1. Introduction

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types (LISP is a perfect example), entails defining procedures which work well on objects of a wide variety (e.g., on lists of atoms, integers, or lists). Such flexibility is almost essential in this style of programming; unfortunately one often pays a price for it in the time taken to find rather inscrutable bugs—anyone who mistakenly applies CDR to an atom in LISP, and finds himself absurdly adding a property list to an integer, will know the symptoms. On the other hand a type discipline such as that of ALGOL 68 [22] which precludes the flexibility mentioned above, also precludes the programming style which we are talking about. ALGOL 60 was more flexible—in that it required procedure parameters to be specified only as "*procedure*" (rather than say "*integer to real procedure*") —but the flexibility was not uniform, and not sufficient.

Dynamic Typing:
(Agility)

Static Typing:
(Robustness)

Dynamic Typing:
(Agility)



Static Typing:
(Robustness)

✓

???