Images of Type

Stephen P Spackman

A Thesis in the Department of
Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Québec, Canada

April 19, 1992

ISBN 0-315-80945-0

Abstract

Images of Type

Stephen P Spackman

Contemporary general purpose programming languages exhibit a straightforward relationship between code- and data- structuring facilities, in the shape of a formal isomorphism between the main operators of each. We exploit this relationship to construct **Visitors**: high-level, executable type descriptors permitting efficient inductive processing of arbitrary data structures, provided only that the data are strongly typed. **Visitors** allow the programmer to write portable, high-level code, while eliminating inessential manual case analysis; the high level of presentation is exploited to minimise the compromise of representational opacity. **Visitors** are amenable to automatic generation as a standard service of a programming language implementation.

The price paid for the use of **Visitors** is that data typing must be inviolable, though there is widely accepted independent motivation for this restriction. Appropriate reconstructions of many common data structures within these limits are exhibited, including several that would initially seem to require the compromise of the type system.

Visitors have by now been employed in garbage collectors, input-output systems and an interpreter. The most recent implementation underlies a software system that should see public distribution within the next six months. A number of other potential applications for the technique have been identified, including the provision of direct, implicit operating system support for typed data.

Acknowledgements

Many thanks are due my good friends: Dr. H. J. Boom, my original supervisor, who started the ball rolling so long ago by asking an impossible question; Dr. Peter Grogono, who provided steady nerves and much-needed terminal guidance; and Dr. Elizabeth Hinkelman, who tolerated an almost weekly explanation of what I think types might be today—and under whose potted tree about half this text was composed. I am also grateful for the more-than-tolerance of my employer, the University of Chicago's Center for Information and Language Studies, during the final writing of this thesis.

Contents

Symbols

Fonts

ABC	Type names	10
$\mathcal{ABC}$	"Extra letters" naming types internal to the **Visitor** system (*Exemplar*)	
abc	Source code	10
$\alpha\beta\gamma$	"Extra letters" naming values internal to the **Visitor** system (*Exemplar*)	
abc	"Stropped" keywords (*Exemplar*)	10
abc	Syntactic nonterminals (EBNF)	12

Operators

xy	Sequencing: x followed by y (EBNF)	
$x \mid y$	Either x or y (EBNF)	12
$[x]$	Optional x (EBNF)	12
(x)	(Grouped) x (EBNF)	12
$\{x\}$	Repeated x (EBNF)—Kleene closure	12
$\underline{x}\ y$	<u>Operator</u>/operand (syntax specification)	12
$l\colon r$	Production symbol (EBNF)	
$[\,]$	Display brackets (*Exemplar*)	17
$\{\}$	Free algebra brackets (*Exemplar*)	17
$\#T$	Mutable type (*Exemplar*)	19
$\%T$	Reference type (*Exemplar*)	20
$p : v;\ e$	Definition (*Exemplar*)	15
$p :: id;\ e$	Local alias (*Exemplar*)	16
$p >\colon b$	Lambda-abstraction (*Exemplar*)	15
$n\ @\ v$	Direct recursion (*Exemplar*)	16
$T :< f\ @\ p >\colon b;\ e$	Named function introduction (*Exemplar*)	16
e^{x}_{a}	Textual substitution: a substituted for x in e	123
$T \sqsubseteq U$	Subtype relation	72
$\lambda x.e$	Lambda abstraction	126
$T + U$	Type sum	123

$T \times U$	Type product	102
$T \dot{\times} t.U$	Dependent type product	123
T^n	Iterated type product	104
T^U	Finite function type	104
$T \to U$	Function type	105
$T \dot{\to} t.U$	Dependent function type	125
$\ulcorner T \urcorner$	The name of type (or type constructor) T in $\mathcal{T}$	45

Names

0, 1, 2	Primitive types of 0, 1 and 2 members	
N	Type of natural numbers (mathematically, $\mathbf{N}_0$)	98
$\mathsf{T}, \mathsf{U}, \mathsf{V}$	Type variables	
$\mathsf{t}, \mathsf{u}, \mathsf{v}$	Variables of types T, U, V	
Vector	Constructor of fixed-length array types	55
Z	Type of integers (mathematically, $\mathbf{Z}$)	98
$\mathcal{A}$	Visitor-argument type constructor (Visitors)	45
$\mathcal{M}$	Visitation mode-function type (Visitors)	45
$\mathcal{P}$	Internal pointer type (Visitors)	45
$\mathcal{T}$	Type of constructor and type names (Visitors)	45
β_T	Primitive Visitor of type T (Visitors)	45
δ_T	Dereferencing function for type T (Visitors)	48

Chapter 1

Introduction

1.1 Framing the Problem, Naming the Prize

One of the traditional great divides in programming language design is between those languages whose type system is *static* and those whose typing is *dynamic*. The advantages of static typing are greater safety and greater efficiency in both time and space, making this approach the favourite of those who think of software in terms of engineering; while dynamic typing provides lower point-to-point coupling within the programme and better support for *generic* operations, characteristics of great benefit in prototyping and for exploratory development styles.

The gap is narrowing: strongly typed languages supporting polymorphism and declaration-free source syntax are increasingly common, while the efficiency of even the most radically dynamic languages increases steadily. This process, naturally enough, involves a convergence of the underlying object code generated: the best compilers for dynamically typed systems infer large amounts of static type information for use in optimisation, while statically typed languages communicate type-related information dynamically in support of polymorphism.

The support of operations that are not merely transparently polymorphic (treating every object as an indivisible atom) but intelligently generic, performing an action that is tailored to the type of an object no matter what that type may be, is still a problem; doubly so in light of the fact that one of the most crucial underlying linguistic facilities from *both* the engineering and prototyping perspectives—automatic storage reclamation—falls in this class. The problem is so severe that until very recently it has been the standard assumption that any really flexible language, of whichever persuasion, would need a dynamically typed internal data representation.

In actuality, this assumption is not warranted: although it is true that polymorphism appears to require that type information have some runtime representation, it does not have to be stored in the objects it describes. An alternate solution exists in the form of a separate system of runtime type descriptors which need only be referred to from data in the case of the value of a polymorphic variable. One problem is to find a good representation for such descriptors.

Not all such generic processes are so intimately connected with language viscera: some of them, like low-level input-output, are at the level of operating system interface; while still others are fully the concern of the applications programmer. A second concern is thus the flexibility and general utility of the system: it would be a shame if it were only of use for the single system task.

Finally, it must be noted that in contemplating such a system *as a user facility* we are, if not playing with fire, then toying with matches: among the great advantages of strong typing are the opacity and security it can provide to data structures. If we are to cut a chink in the anonymity of the arbitrary datum, we must take care that access to it is well controlled within the conventions of the language.

The facility we are looking for would thus ideally provide the profound support for genericity that is still the province of dynamically typed languages, within the realms of static typing. More importantly, it would do so at the levels of performance customary in the statically typed world, not compromising the temporal or spatial efficiency of the language, even relative to those "old school" languages (like Pascal, C and FØRTRAN) that trade semantic coverage for performance.

1.2 The Form of the Solution

The solution we propose to this puzzle takes the form of a large recursive nest of polymorphic higher order functions that we term **Visitors**. The entire nest is derived automatically by the compiler from the particular type system of the programme. The construction of this object relies on the relationship:

$$\textit{grammar} : \textit{string} :: \textit{programme} : \textit{trace} :: \textit{type} : \textit{datum}$$

in conjunction with the observation that not only are grammars, programmes and types themselves strings with grammars of their own, but that these second-level grammars are, for typical contemporary formalisms, *isomorphic*. Thus, just as it is quite natural to think of the "grammar" of a data structure describing the relative locations of its components, so, too it has a naturally corresponding *code structure*, and it is on this framework that the **Visitor** of a type is constructed.

A natural concern with this whole scheme is efficiency. While the general outline of the constructed functions is straightforward, the flexibility we need to fulfill our promise of general applicability is bought at the expense of functional parameterisation of the whole visitation subsystem, and the resulting object is very abstract indeed. Fortunately, the *right* abstraction brings simplicity. In particular, it appears that (thanks perhaps to the fact that object code is simultaneously a programme and a datum, bridging the levels from below) **Visitors** are perfectly matched to typical hardware architectures and compile beautifully. Their intricate but structurally homogeneous interrelationships evaporate into an ability to ignore many of the usual strictures of code generation, simplifying stack frames and reducing register allocation almost to triviality.

While pretty, our solution is not perfect. It is comparatively cheap, but it does not come free: code is not the most compact form of type description possible, and our policy of non-interference with normal code generation is both somewhat imperfectly executed and dependent upon the existence of sometimes extensive additional structures. We may use more space and/or more time during garbage collection than traditional, more intrusive techniques require. And finally, while we suggest a semantic interface to the visitation system that we feel forms the best compromise between the preservation of representational opacity and the necessary license to perform the kind of type-blind but type-conscious operations that here concern us, we have not yet attempted to devise a source syntax that would permit its reflection into the user-programmer's visible language model.

On the other hand, **Visitors** have proven themselves to be a useful technique in the real world: they have been turned to a number of different tasks (input-output, storage management, even serving as the kernel of an interpreter), requiring no modification once in place. They have increased our freedom in the selection of underlying data structures and, perhaps most tellingly, supported repeated changes in the storage management system of one language without a glitch.[1]

The price of the whole arrangement is paid in the *necessity* of typing: since we provide these operations by manipulation of types, *every* expression in the source language must be strongly and unambiguously typed (though of course the intrusiveness of this requirement on the programmer is a function of the source language design). Fortunately, there is no restriction that the type system we support be simplistic or inflexible; in fact, as will be seen, even when reconstructing characteristic objects from languages of a far more pragmatic and semantically cavalier nature, the devising of a suitable source notation has often proven more difficult than the construction of a suitable type (and its corresponding **Visitor**).

[1]Well, with just one glitch, one that arose from an unintended "hole" in the type system.

1.3 The Overall Structure of this Thesis

This thesis is structured as follows. The systematic reader is nearing the end of the introduction. In the next chapter, we introduce the superficially Algol-like language *Exemplar*, making an attempt to get its formal structure out on the table from the start so that it can be used as a vehicle for later discussion and example, not stopping again to explain details of the formalism. The reader is cautioned that in *Exemplar* the strange is sometimes made familiar and the familiar made strange; but a profound understanding of the language described should prove unnecessary, since we use no subtle consequences of its structure.

Next, the main exposition forms (as it were) a series of three spirals, each spiral at a different level of abstraction. The first is about *structures* and discusses the structural facilities of *Exemplar* and of various classical languages; it proceeds to the construction of **Visitors**. In chapter 4, the second spiral treats of *type*, philosophically, empirically and pragmatically: these types are the abstract objects of which the structures of chapter 3 are source-language images. The final spiral, chapter 5, treats of the eventual machine-level *representations* of types in terms of their actual storage layouts and, ultimately, the physical representation of their **Visitors**. These, then, are the object-level and the *executable* images of type.

In outline, each spiral begins with the abstract form of the object, in its simple cases and as we prefer to see it. Next, we consider something of the "real world," discussing the approaches taken by various other languages or otherwise trying to establish some of the breadth of the field. Then comes the nitty-gritty, where we generalise what we have seen of reality, tackle the hard cases and generally come to grips with meta-recursion. Finally the spiral ends and we move back onto the abstract plane, considering implications, consequences, applications or alternate formulations.

After three spirals we come out of our loop and examine practical applications, starting (in the sixth chapter) with sketches of how **Visitors** might be used in a number of quite different contexts. In the following chapter we look at garbage collection, the lowest-level, most critical, and earliest application of visitation, in some detail. Finally, we consider the various empirical incarnations that **Visitors** have seen and attempt to summarise our practical experience in using them in their successive refinements over the course of several years.

1.4 Points of Interest Along the Road

Along the route from introduction to conclusion we shall be passing a number of points of local interest. This last introductory section is intended as a guide to such secondary concerns as the interested reader might wish to look out for.

Abstraction It might be observed that this thesis is in part an attempt to do low-level systems programming at the highest possible level of abstraction. The particular design heuristic involved is the abstraction of all the parts of a problem to a level where they exhibit the *same* structure and a trivial solution is possible (in this case that common structure is the syntactic form of EBNF). This technique does not always work, but when it works it produces nice solutions. Sometimes abstraction comes free!

Exemplar The *Exemplar* language has as subsidiary motivations the demonstration: (1) that the provision of "convenient Pascal-like syntax" is orthogonal to the imperative/applicative semantic divide in language design philosophy; (2) that such a syntax can combine straightforwardly with a simple and directly manipulable *abstract* syntax à la LISP; and (3) that (although we make no use of this possibility in the present work) there is no barrier to allowing the *user* full access to "keyword notation" either in terms of parsability or readability of the uses of user-defined constructs (note that the description of *Exemplar* lists only the *scheme* for such syntax, not its instances).

Rotation A recurring theme in the construction of efficient implementations is coördinate rotation: wherever one thing names or describes another the intuitive representation puts them close together, but the preferred physical representation is often to move the two into separate parallel structures in the hopes that further standard optimisations can merge the descriptor with its context (or eliminate it entirely). There is an argument to be made that the objects described in this thesis can be derived by repeated application of this transformation to an underlying object-oriented system (driving the process, however, with the early binding of type information that our source language provides).

Pointers Pointers are wonderful and faster than all the alternatives.

Nothing Doing nothing is even faster than using a pointer.

Strong Typing as an Asset Strong typing has traditionally been seen as a positive thing, but a bitter medicine. While the author admits to a pronounced recreational interest in the subject, it is perhaps of general importance in language design to ask what benefit can be expected if such a rigorous discipline is to be enforced. **Visitors** actually provide a more powerful but more controllable service than their dynamically typed analogue (implicit in the structure of object oriented systems), and this is surely not the only such case.

Antiprimitivism Programming languages almost universally exhibit a fundamental divide between builtin and user-defined objects (though some also possess an intermediate class of nonredefinable standard objects). This manifests as reserved names, unusual syntax, special lexical forms, (nonprogrammable) systems of coërcions and so on. This is in utter violation of what we know about software engineering, opacity, orthogonality, human factors, apple pie and child rearing and should be stamped out just as soon as someone works out how.

The $m + n$ Problem One of the most famous conceptual advances in compiler technology was the identification—and solution—of the "$m \times n$" problem (whereby writing mn compilers merely to provide m languages on n machines is avoided by the factoring of the compiler itself into separate front- and back- ends). We are still left, however, with a residual "$m + n$ problem"—writing m front ends and n back ends is still rather a lot of work. Aside from attaining $m = 1$ by designing the One True Universal Language (a grail the pursuit of which still interests the author), the best we can do is to make the "middle end" of the compiler, the truly portable part that comes between the front and the back, bear as much weight as possible. Many of the comments in this thesis are motivated by anticipation of increasingly nontrivial optimisers working at this level.

Chapter 2

Exemplar—an example language

In this chapter we introduce the hypothetical programming language *Exemplar*, which will serve as a vehicle for the coming discussion. *Exemplar* can be characterised as a strongly typed non-pure applicative-order functional programming language with heavy syntax—in short, it numbers among the descendants of Algol 68.

We provide only an informal description of *Exemplar*; and we will focus upon only those of its properties that are of particular relevance to this thesis. The language will be found to be somewhat more complex than is strictly necessary as a source of examples for the techniques to be discussed; but this will permit its use for the description of the code we generate, as well as for posing the problem. It is not, it must be stressed, a *practical*—or even implementable—language: a number of its features are present for expository purposes and are not well matched or particularly useful in the form in which they are presented (and a number of its features are known to pose unsolved technical difficulties, though that will have no impact on our exposition).

2.1 Lexical Rules

As in most languages with free layout, whitespace serves in *Exemplar* only to delimit tokens (and even then is optional in the absence of ambiguity). Having said that, we also make the usual hedges that comments (commencing with ".. " and ending at

letter:	a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z 0|1|2|3|4|5|6|7|8|9|.|_
slanted:	*a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z* *A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z* *0|1|2|3|4|5|6|7|8|9*
opchar:	*(any of a number of punctuation marks, operators and commercial symbols)*
opener:	(|[|{
closer:	}|]|)
quote:	"{*(any character except ")*}"
atom:	(*letter*|*quote*){*letter*|*quote*} *(except ..)*
key:	*slanted*{*slanted*} *(except end)*
opsym:	*opchar*{*opchar*|:}
ternary-opsym:	:{*opchar*|:}
(punctuation):	; *end*

Figure 2.1: Lexical rules of *Exemplar*

the end of a line) are whitespace, despite being non-blank, and that quoted material is never whitespace, be it ever so pale.

Unlike conventional languages (at least those that are as syntactically "heavy" as *Exemplar*), most tokens are considered to be one sort or another of identifier at abstract syntax,[1] even those which on the surface are bracketing constructs. This

[1]Abstract syntax, for the reader unfamiliar with the term, is essentially the output from the parser and the input to semantic analysis: the distinctive feature of LISP [McC60, Ste90] is *not* (as is often stated) that it provides straightforward manipulation of LISP *source code*—which is, after all, represented as a text file as in any other conventional language—but that access is provided to the abstract syntax level, and *it* conforms to the usual data structuring conventions of the language.

greatly simplifies the description of the language, since the parsing of *Exemplar* is independent of its repertoire of semantic features.

The lexical class of **atoms** (see figure 2.1 and the examples of "primaries" on page 13, figure 2.3) consists of sequences of letters, digits, underscores, points and material enclosed between double quotes (" "). It may be divided into several subclasses: **type identifiers** (and the names of a few type-like objects, such as TYPE itself) start with an uppercase letter; conventional object-level identifiers start with a lowercase letter. The remaining possibilities are largely reserved for **literals**, the exceptions being "..." which represents an **elided expression** (this is indeterminately notation and metanotation, in that it is in principle an expression within the language, but executing it would ideally trap to a debugger), and "_" which we use (perhaps confusingly) both as a pattern that matches anything and as the only value of type Void.[2]

A **key** is a word written in a *slanted* font; a **keyword identifier** consists of a *sequence* of such items, with the last being "*end*" (so that we may talk about the object[3] "*if then else end*"). These fulfill the same function as keywords in conventional languages without (*end* excepted) being in any way reserved.

An **operator** is a sequence of punctuation marks and other otherwise un-spoken-for symbols; if the first character of an operator name is a colon, then the operator is **ternary** and is syntactically associated with a subsequent semicolon; else it is a **prefix** or **infix** operator, according to context.[4]

Finally, the **brackets**, ( ) [ ] { }, are tokens, with each matched pair being an identifier (albeit, and sadly, one unavailable to the user).

[2]Thus such useful monstrosities as o"+" are identifiers, while 1962.nov.08 is—at least in lexical form—a literal.

[3]Or, more properly, the object so *named*.

[4]The distinction between prefix and infix uses is formally a distinction of *type* resolved as an overload, as we can see from the fact that both negation and subtraction have the name "(−)"; but this is unrelated to the ternary operator ":− ;". The reason for this admittedly odd distinction is that the author is partial to the operator :: ;, and making ":" and ";" into a pair of brackets would spell its end. As if, good reader, you cared.

primary:	*atom*
	opener closer
	$\underline{\textit{opener}}$ *opsym* $\underline{\textit{closer}}$
	ternary-opsym ;
	key $^{\{}$*key*$^{\}}$ end
secondary:	*primary*
	$\underline{\textit{secondary}}$ *primary*
tertiary:	*secondary*
	$\underline{\textit{opsym}}$ *tertiary*
expression:	*tertiary*
	tertiary $\underline{\textit{opsym}}$ *expression*
	tertiary $\underline{\textit{ternary-opsym}}$ *expression* $\underline{;}$ *expression*
primary (cont'd)*:*	$\underline{\textit{opener}}$ *expression* $\underline{\textit{closer}}$
	$\underline{\textit{key}}$ *expression* $^{\{}$$\underline{\textit{key}}$ *expression*$^{\}}$ $\underline{\text{end}}$

Figure 2.2: Syntax of *Exemplar*

2.2 Syntax

The surface syntax of *Exemplar* is fairly straightforward. An *Exemplar* programme is a single **expression** according to the EBNF grammar of figure 2.2; in each case the operator at abstract syntax (which, as in the case of brackets and keyword syntax, may be textually discontiguous) is underlined, $\underline{\text{thus}}$.[5] Nonterminals are written in an *italic* font, and the regular grammar operators are superscripted: $^{\{}a^{|}b^{\}}$.[6]

A few features of the syntax are worthy of note. First, there is very little notion of operator priority: almost everything associates uniformly (even more so, perhaps, than in APL). Second, the operator/operand assignments made with the $\underline{\text{foo}}$ notation provide the entire translation of the language into abstract syntax: resolution of the

[5] So traditional function-application notation would be rendered: $\underline{f}(x)$.

[6] We employ the bracketing style of EBNF notation; thus the vocabulary of BNF right hand sides is augmented with the (regular) operators $^{\{}x^{\}}$ for Kleene closure, $^{[}x^{]}$ for optionality, and $x^{|}y$ for alternation; $^{(}x^{)}$ is used for grouping.

```
primary:          TYPE                  Character
                  x                     foo":"22                        (identifiers)
                  42                    "badger"
                  6.02.e23              9.8.m..s2                          (literals)
                  ...                  (an elided or unwritten segment of code)
                  _                                          (a null pattern or value)
                  ()                    (*)
                  :- ;                  if then else end

secondary:        TYPE T                Boolean more
                  w re                  f[x]

tertiary:         %Complex[Z]
                  -5
                  +v                         (take the current value of variable v)

expression:       x + 1
                  head[x, n] ++ a ++ tail[x, n]
                  N l  :  length[s]; space ^ width - l / 2
                  n := +n + 1; n                          (increment and return n)

primary (cont'd): (x + 1)
                  {red, green, blue}
                  (N length, Vector[Character, length] text)
                  if x > 0 then x else -x end
```

Figure 2.3: Lexical and syntactic examples

operator/operand relationship is *never* influenced by the particular names appearing in an expression. Third, the production for a *secondary* provides an abstract notation subsuming a number of distinct concrete cases, including function application (function[argument]), field selection (object selector) and variable declaration (Type identifier). Fourth, much of the flavour of serial execution in imperative languages is captured in the ternary operators; where serial imperative operations are to be encoded, this is the preferred notational device. Nonetheless, the vast majority of the uses of this notation do *not* involve side effects, manipulating bindings and scope instead. Finally, the arities of operators and brackets[7] are assumed to be resolved by

[7]Since, of course, {} and {x} denote uses of distinct objects, even though the latter could equally

type inference (which poses few technical difficulties in a paper notation).

2.3 Scoping

Because parsing in *Exemplar* can be accomplished without reference to particular identifiers, we are free to treat scope control as a quasi-semantic function: it is a property of identifiers and expressions, and is computed over the abstract syntax tree. Expressions in *Exemplar* are divided into three **categories**: they can be *values, patterns* or *types*.[8] The category of an overall programme is *value*, and the category of each subexpression is determined by (the specification statically associated, in the present category, with the name of) its operator. In the absence of any other information (and in particular, whenever an identifier has more syntactic arguments than it specifies categories for), the argument of a value is a value; while the arguments of types and patterns are patterns.

Patterns are "left-hand-side" expressions, used to *dismantle* values, just as values build them; as value-expressions employ identifiers, so patterns introduce them. The discussion of types is deferred to a later section.

As with category, the scopes and bindings of identifiers are also controlled locally by the main operators of expressions, and as part of the same category scheme—at least as regards their passing up or across the abstract syntax tree; for as is conventional, they pass down it automatically. Patterns can introduce new names (typically by "applying" a type to an identifier), and types and values new bindings; these are by default passed back out through types and patterns, and by specific design of the value operator : ;.

---

have been written "{} x".

[8]This is a bald lie. In fact there are also (at least) type-patterns and type-types, but introducing them into this discussion would muddy the waters rather than clearing them. Suffice it to say that "TYPE" is to a type as a type is to its members, and the locution "TYPE T t" would be a pattern introducing a value t of polymorphic type (locally named T). The potential ambiguity that arises in free algebra construction (described below), between {x[T]} matching exactly x[T] or any x[t] (t in T) is resolved in favour of the latter, since types are considered to be incomparable at runtime (type identity is resolved at the level of abstract syntax).

Names and bindings passed out in this manner can be intercepted by the operator determining the class of the subexpression, and it may reëxport them to any explicitly determined arguments further to the right. A name always carries type with it, and such reëxport serves to introduce a new overload on the name, overriding only overloads with which it conflicts.[9]

Within this general "scoping theory," the conventions followed by the *Exemplar* entities in this thesis are roughly that names are exported, wherever meaningful, to arguments semantically controlled by the argument introducing them (whether or not this is the primary purpose of the construction); thus for example, declarations from the *if*-part of a conditional expression will be visible in the controlled branches.

Note that these rules imply that all upward and rightward transfers of visibility are explicitly and iteratively licensed by the operator (or its category), and leftward such transfers are impossible—*Exemplar* is, in sum, a strict declaration-before-use language.

2.3.1 Binding operators

Exemplar as here presented possesses several operators that exist for the particular purpose of binding names to values.[10] There are five of them, as follows:–

$p : v; b$ (*let*-**binding**) The expression v is evaluated and matched to the pattern p; this match must succeed. The resulting bindings are put into effect for b, which is evaluated to yield the result of the expression.

$p >: e$ (**λ-abstraction**) The expression e is abstracted on the pattern p, which matches the eventual argument list of the function so resulting (or, conversely, a function is constructed which is evaluated by

[9]And, for present purposes, damn the torpedoes.

[10]There are other places in which binding occurs; in particular a *for*-clause in a loop binds its control variable. In this section we omit reference to such cases of "collateral" binding as being fundamentally uninteresting.

matching its argument list to yield a set of bindings under which e is then evaluated for its result). The match is allowed to fail in the case that ther. are several such abstractions bound to the same name, and one of the others succeeds.[11]

n @ e **(Circular naming)** The expression e is evaluated in an environment in which n is bound to the result of e's evaluation (which is also the value of the whole expression). *Caveat* caller: if the language's semantics cause this object not to be well-defined, no magic will save the programme from its doom. The binding of n to e is exported, principally for the use of:

T :< f ; b **(Abbreviated function definition)** Parallel to : ;, this introduces a function f with *return type* T to the scope of b, which is the value of the expression. Note that f must be a function-to-name *binding*, and typically has the form n @ p >: e.

p :: n; b **(Local naming)** Unlike the above (which are acceptable both as value- and type- expressions) this is a type-expression only. It yields the product of the type of the pattern p and the type b, but with the value eventually bound in p bound *also* to n in b. Only the former binding is exported as a nameable "field" of the type. (In particular, if p makes no binding, this field will be internally named but externally anonymous.)

Here is a simple example of the use of several of the above, in the form of a function quadratic of arguments a, b and c, which yields the function (over the integers, Z) $\lambda x.ax^2 + bx + c$ (which may subsequently be evaluated for any desired values of x):

[11]In a practical language we would here have to concern ourselves with how the type compatibility rules distinguish overloads from clausally defined functions.

```
    Z[Z] :< quadratic @ (Z a, Z b, Z c) >:
      Z x >: (a * x * x) + (b * x) + c
  ; ...
```

Note that the return type of the outer function is a function type in its turn. Since dependent types are a relatively arcane topic, we defer further discussion of the local naming operator, :: ; to a later chapter.

2.4 Types

This section deals briefly with general aspects of *Exemplar*'s type system, in particular as it differs from more conventional languages in its manner of dividing up the space of possibilities.

Along with a number of **atomic types** whose existence we hall simply assume as we need them, *Exemplar* provides a data structure description mechanism based on the syntax of its control constructs; we shall temporarily defer the discussion of this mechanism, since it is integral to the substance of the next two chapters. The values of structured types are given as bracket ([]) delimited displays. We shall also have occasional recourse to **free algebras**, types introduced by listing, within braces ({}), a sequence of signatures that the constructors of the type satisfy; information stored with the constructors thus introduced is maintained losslessly[12] and may be extracted through the use of pattern constructors homonymous with the value constructors. This mechanism of course subsumes the familiar enumerated types of Pascal and its relatives, in the case that the signatures are trivial.

In addition to these data structures, we provide **function types**: types whose members are themselves mappings from one type to another. *Exemplar* functions are first class objects, their nonlocal references lexically bound (as they must be for functions to be "values" in any meaningful sense); they are created by λ-abstraction

[12]But see section 3.5, where we (briefly) consider an extension to the language in which these free algebras are taken modulo constraints to build arbitrary types in the "algebraic data types" style.

with the operator >:—and eliminated by juxtaposition with a structure display giving their arguments. The argument list itself is in fact considered to *be* a structure: the same facilities are provided to construct and interpret the two (though anonymous fields only positionally nameable are of less use in an argument list than in structures, since they would be inaccessible to the function body).

We assume that types are opaque except in so far as their structure is (transitively) lexically associated with the name under which they are located. Since the name of a function counts for this purpose (in the identification of the type of its argument list), no difficulty arises.

One of the superficially novel features of *Exemplar* is the *extension* construction: from any type T we can construct new types differing from T only in the addition of a set of new, freely constructed elements; this is written by "applying" T to the appropriate brace-enclosed signature list. The extension elements are used in exactly the same way as the elements of a free type; and the members of the base type remain accessible through their original notation (except where new elements shade it).[13] In actuality this facility does no more than legitimise the practise of using guard values and exception codes, providing a type secure and lexically transparent notation for these notions, within the language.[14]

Finally, there is TYPE, a type-like structure whose values are the types them-

[13]The practical expectation is that, wherever possible, extension elements will be implemented with unused code space in the base type, rather than consuming additional storage. Of course, given the generality of this mechanism, no such guarantee is made. One application of extension types that is being investigated is their integration with an exception handling system: it turns out to be convenient to provide an automatic conversion from any extended type to its basetype with the property that if conversion of an extension element is attempted the exception of the same name (and type signature) is raised. This allows interfaces to be written in a notation that is clearly defined independent of exceptions, restricting exception handling to the lexical domain. Experience will show whether or not this is ultimately desirable.

[14]Eventually we hope to be able to extend this notion further, to a general mechanism for distributed clausal definition of computational objects, covering not only functions but values and types, and permitting a consistent interpretation for such extensions occurring in local scopes. There remains to be resolved, however, a technical question of under what circumstances it is desirable for an extension to propagate back to the original definition.

selves.[15]

In light of the fact that this is merely a paper notation and given the immense human aptitude for type inference, we shall not give a type assignment algorithm for *Exemplar*; indeed, it may be that, as the notation stands, no adequate type inference algorithm is possible.

2.4.1 Modifiers and attributes

Not really forming part of the basic type system of the language (for though *Exemplar* contains constructions syntactically and pragmatically reminiscent of block-structured imperative languages, they have been designed to operate unimpaired in a side-effect-free programme), but enabling us to express information about the implementation of programmes and to permit programming in the more classical imperative style, we provide two additional constructors for manipulating impure data: # and %.

The constructor # builds, for any type T, a new type #T of **mutable** T's which, like the classical "variables" of FØRTRAN and Pascal can be "modified" by the side-effecting operator := ;;[16,17] mutable values are introduced with the constructor | and, following Bliss, we provide an explicit demuting operator, +.[18] It should perhaps be observed that the operator | is actually responsible—at least in the most direct reading—for the allocation of storage, and is the only place in *Exemplar* where this is

[15]The language design from which *Exemplar* derives provides a number of such pseudo-types, sharing the property that a lexically interpreted intensional semantics affecting source interpretation overlies a class of specialised runtime values. These pseudo-types include PATT, VALU and NAME, and are used to *define* such constructs as *if then else end*, : ; and >:. They are the means by which the details of category and scope control are initially specified.

[16]Strictly speaking, of course, it is the (type T) *contents* of the #T that are changed, not the #T itself.

[17]We avoid the use of the word "variable" for these objects, preferring "mutable," (1) because of potential confusion with the "logical variables" of Prolog; (2) because in functional languages (to which class *Exemplar* belongs, even though it contains the impure constructions described in this section) the term normally refers to named objects which *cannot* change (modulo instance unidentity); and (3) it seems clear that the mathematical tradition—that "variables" differ from "constants" only in that they are locally more arbitrary—has priority.

[18]Which choice of symbolism should prove mnemonic to those readers lucky enough to be able to remember the operator priority tables for C.

unambiguously the case (since values with no identifiable "location" need not literally exist). There are no restrictions on the lifetimes of mutables in *Exemplar*: if they are passed out of their original scopes they are presumably preserved in the heap.

In strictly parallel fashion, the % constructor builds **pointer types**, which provide references to objects.[19] Pointers may be constructed with the & operator and eliminated with *.

Although *Exemplar* as used in this thesis makes no formal reference to them itself, in the text we shall have recourse to the terms *pure* and *impure, naughty* and *nice*[20], in reference to programmes that may make use of mutable values; and we shall occasionally have reason to speak of the type "Naughty T". In general use a distinction is drawn between *pure* expressions which do *not* involve side-effects and *impure* expressions which do; it turns out to be useful to analyse the situation further, characterising an expression on the one hand as **pure** if its value is independent of the "state" of computation (independent, that is, of any assignments or analogous operations in primitive functions), and on the other as **nice** if it, in turn, has no impact on the state.[21]

2.5 Summary

The language *Exemplar*, while Algol-like, attempts to provide clean lexical and syntactic layers with an explicitly compositional flavour. Issues relating to type and notational interpretation are resolved as much as possible *within* the language, but at

[19]Those familiar with Algol 68 will note that we have here unbundled the ref into its distinct referential and mutable aspects. This is justified by the fact that references to objects with mutable *parts* are very useful; in our analysis the resulting object is *impure* (see below) but immutable. More relevant to the present undertaking, perhaps, is that references and mutables have very different characteristics at the implementation level, and it is appropriate to our present purposes to allow this to shine through into the upper language.

[20]On the reading of "well behaved," of course.

[21]The properties of effective purity and niceness—in which the attributes are taken modulo some abstraction barrier—turn out to be fascinating, important, and hard to determine. A typical buffering or caching scheme will, for instance, turn out to be *effectively nice* while nonetheless manipulating a great deal of internal state.

a level very close to the surface. A sequential flavour is arrived at through notational ruse: the standard "structured control" operators remain effective (as we shall later see) in the pure functional subset of the language. Finally, (though we shall not see this until the next chapter) the control and type specification sublanguages are made very tightly parallel.

The approach of handling the mechanics of the language as independently as possible from its specific features allows us to defer the discussion of particular *semantic* facilities until they become directly relevant to our discussion. We trust that, through these characteristics, ignorance of any given feature will result in strictly bounded amounts of reader confusion, a property that cannot be claimed of many previous language designs.

Chapter 3

Data Structures, Control Structures

In this chapter we discuss **structures**, the abstract tree-like entities of which modern programming languages (and, for that matter, contemporary theories of natural language[1]) seem almost universally to be constructed. We progress from a consideration of control and data structures in *Exemplar* to a survey of parallel facilities in more typical languages. Finally we bring these threads together in the development of the **Visitor**. The chapter closes with a brief investigation of an extension of algebraic data types which provides an alternate construction for **Visitors**, one more closely related to the high-level constructs of some contemporary programming languages.

3.1 Structures in *Exemplar*

As is typical of languages in its class, *Exemplar* provides a number of high-level, block-structured methods of describing (or, up to isomorphism, prescribing) both the *flow of control* and the *arrangement of data*.[2] In the present case it will be found

[1]At least at a syntactic level. [Sad91] in particular presents a syntactic and morphosyntactic theory based on multiple parallel context-free analyses that is of interest to formal language theory as well as to linguistics; [GKPS85] presents a more conservative (though computationally enlightened) view of the topic.

[2]This contrasts both with early, pre-structured approaches where block structure was absent, and with a common modern sentiment that perhaps such extensive, centralised structuring facilities demonstrate an undue concern for details of implementation better left to the evaluation mechanism. In fact, it is the author's contention that having more than a minimal repertoire of such structures in the *kernel* of the language is indeed a mistake, but we here limit our discussion of this point

that these two sets of structures are closely related in both form and meaning; and in fact it will be argued that such a close parallelism can be anticipated, at least underlyingly, in most general-purpose programming languages.

Since our concern is with structures rather than applications, we here ignore the terminal elements of programmes: the atomic data and expressions from which they are composed. In practise we shall assume the "usual" facilities: integral and character types with their conventional algebraic operations, and assignment and value extraction constructions from which "impure" and "naughty" expressions may be built. Theoretically, of course, we assume that a language implementation merely provides a library of such semantic domains upon which the programmer may draw freely.

Similarly, we shall for the moment ignore function types—in their data-object aspect—and the representation of their instances as closures. Since they capture suspended *computations*, their representation, though straightforward, can be expected to be far more implementation-specific than those of more "usual" data. In fact they may be treated more or less uniformly with other data structures, as described in section 5.4.

3.1.1 Control structures

Sequencing: The simplest of *Exemplar*'s control structures is the familiar grouping-and-sequencing operator corresponding to Pascal's begin...end block, and to LISP's progn. In *Exemplar* this is written with parentheses surrounding the group and commas separating the items: (a, b, c).[3] Operationally, this indicates that the comma-separated expressions are executed in textual sequence from left to right, returning

to the observation that it would seem advantageous in any case to represent all control constructs in their recursive formulations in the early stages of compilation; and this opens the door for their direct definition *within* the language.

[3]Technically speaking, of course, the parentheses name the polymorphic universal identity function, mathematically I, while the comma is a binary go-on operator of type $\forall$T.Naughty Void $\rightarrow$ T $\rightarrow$ Naughty T. This is straightforwardly related to the semantics given informally above, by currying or destructuring, once noted that the notion of sequencing here resides wholly in the comma.

the value of the last expression evaluated. More abstractly, since it controls the order in which the statements are executed, it specifies the manner in which information may flow between them: it flows from earliest to latest, from left to right.

Selection: Terminal control bifurcation can be specified through a conventional "if-statement" of the form *if condition then sequent end* or *if condition then sequent else alternate end* where the *sequent* or the *alternate* (if any) is executed, depending as the Boolean *condition* is true or false. The *else*-less form is of type Naughty Void, while the two-branched version (since the branches cover all immediate execution futures) is uniformly polymorphic in the branches and the result. A second form of conditional, akin to the familiar case-statement, has the form

```
if expression
is pattern_1  then action_1
is pattern_2  then action_2
  ⋮
is pattern_n  then action_n
else action_default
end
```

where the $action_i$ must all have the same type, the result type, which further must be Naughty Void in the absence of an *else*-clause. Here the first clause whose pattern[4] matches the given *expression* (or the default if none does) is evaluated to yield the overall result of the *if*-expression.

Thus the effect of any of these *if*-constructions is to execute one of a number of alternative expressions, chosen on the basis of the state of the programme when the construction is reached.

Iteration: Loops in *Exemplar* are notated in a manner similar to that of Algol 68: there is a single basic loop structure that can carry independent specifications

[4]At least in the full language of which *Exemplar* is a subset, a pattern is simply a formal parameter, and the *if-is* construction has the same behaviour as a clausally defined function called in place—except that the arms of the construction are within the scope of any declarations exported by the *if* argument.

for termination conditions (*while* ⌈*then*⌉...), control variables (⌈*for*⌉ ⌈*from*⌉ ⌈*to*⌉ ⌈*by*⌉ ⌈*then*⌉...), induction variables (*with from*...*as*...) and body (*do*...). From these components we can construct the classic while-loop:

```
while condition
do body
end
```

a for-loop:

```
for pattern from expression to limit by step
do body
end
```

and any number of hybrid forms like this fragment, which sums the results of f until first it returns 0:

```
with N r from 0                                   .. Accumulate result as r
  for N i from 1                         .. Iterate over successive naturals
  while N x : f[i]; x ≠ 0 then r                  .. Test; perhaps exit with r
as r + x                                                   .. Compute new r
end
```

The various prefixes can be combined and repeated *ad libitum* in a single loop; and they are are considered in left-to-right sequence on each iteration (which is to say that they interact elementwise, not cross-wise, as they would for nested loops).

The *then*-clause is used when it is desired to construct a **yielding** loop; its expression is evaluated if the associated loop controller causes the loop to terminate, and its value is the value of the loop. If any controller has a *then*-clause, then all must; and they must all have the same type.

The *for from as* group is used to maintain non-controlling induction variables over the loop, with the *from* clause giving the initial value and the *as* clause updating it; as can be seen in the third example above, the *as*-part need not be adjacent to the first two clauses of the group. Multiple *as*-clauses are associated with *for*-clauses in

LIFO order rather than by textual adjacency so that the induction process may benefit from the loop body itself.

The overall effect of any of these looping constructs, however specified, is that the controlled expression(s) are evaluated *repeatedly,* for a total number of iterations controlled both by the conditions above the loop on entry and by the results of each previous iteration of the loop in turn.

Declaration and invocation: By now we have seen a number of examples of declarations. Declarations of functions, in combination with function invocation, form another control structure, and one that is in many senses more primitive than those discussed so far (in the sense that any data structure can be constructed straightforwardly within the λ-calculus): they are the general mechanism provided in most programming languages whereby large (and even infinite) objects can be described by a finite and practical notation, and structures such as loops and if-statements are readily coded in terms of them.[5] Declarations in *Exemplar* are accomplished through a form of let-statement:

pattern : *expression*; *body*

Note that this construction is *not* an assignment statement (which we shall write with the operator := ;), deviating both in that the left hand side is a *pattern* rather than a (mutably-valued) expression, and in that it denotes a *lexical* rather than a side-effecting operation.[6]

Function invocation is denoted by following a function-yielding expression with its possibly empty bracketed argument list.[7]

[5]In the case of lazy languages it is entirely possible to describe infinite structures without the use of functions, but it will be found that the underlying implementation employs intimately related closure-like structures for all but the structurally most trivial.

[6]It might be notationally convenient to permit the introduction and initialisation of a named mutable within a pattern, however. This is the correct formal analysis of the situation in a language that permits "structures" of variables in a left-hand-side.

[7]Technically speaking this obviates the need even for the declaration syntax just described: we could operate purely in the combinatory realm. Such an undertaking is, however, more than slightly cumbersome from a notational perspective.

The combined effect of function declaration and invocation is (as in any applicative-order language) that a sequence of programme text—the function body and whatever it may recursively invoke—is effectively completely evaluated before proceeding, even though the size and complexity of this execution subhistory may arbitrarily exceed that of the text of the call. Furthermore, it provides a mechanism for combining information derived from different contexts via the separate routes of the argument list and the lexical environment of the called function; this combined information may influence the flow of control within the function, as well as its result.

We have not yet introduced any direct mechanism for recursion[8]; we provide an operator, @, for the purpose, such that n @ v is precisely v evaluated in a context where n is bound to v. There is in fact more than one interpretation available for such recursion, depending on what is done about the apparent paradox that v must be evaluated in such a way that its own value is already available.

(1) We can require that @ introduce only a well-founded induction (or recursion): this is the classical interpretation in programming languages. If v is a function some of whose range does *not* depend circularly on v itself, and every other point in its range is reducible to one of these base cases through a finite number of reïnvocations of the function, then clearly enough every actual *invocation* of v is finite and noncircular and there is no difficulty in practise (an "optimising" implementation must simply take care not to outsmart itself by indefinitely expanding recursive invocations of a function inline!).[9]

(2) We can in principle (it will become clear in the next section, in our discussion of types, why this case is of particular interest even though it does not lend itself to practical implementation) add to these instances those that are precisely *circular*, in

[8]While it is in principle possible to encode the Y combinator directly (Y is defined such that for all f, $\mathbf{Y}f = f(\mathbf{Y}f)$), this only pushes the difficulty back a step in that the type that would have to be assigned to Y is itself pathologically recursive.

[9]This corresponds, in the scale of objects generated, to, say, to the subset of the reals expressible in "scientific notation," if both mantissa and exponent are allowed to be a arbitrary integers.

the sense that v (or a parameterisation of v) is allowed to refer (directly or indirectly) to itself in those cases where it is evident how to compute a fixed point for the object; such a facility in fact constitutes an exceptional and idiosyncratic execution mechanism.[10]

(3) We can admit expressions computing values that are *constructively* infinite: each increment of structure in the result must become available after only a bounded amount of computation, and we manipulate the evaluation order of v in such a way that if a programme only attempts to examine a finite part of v (as, given the finitude of the real world, it will), it will terminate, leaving the remainder of an infinite v almost everywhere suspended in mid-calculation.[11]

(4) We can contemplate the admission of *classically* infinite objects (for which no such condition of guaranteed headway is imposed), at the expense of implementability.[12]

In light of the fact that *Exemplar* is a fairly low-level language, sticking close to the underlying semantics of the von Neumann machine, we adopt the first of these interpretations—though circular objects can still in fact be built by virtue of the existence of mutable values.

For the convenience of the programmer in the construction of functions there is a subsidiary use of the @ operator. Because of a notational clumsiness in the definition of named functions (in that both the patterns in λ-expressions and the denotations of function types themselves require the specification of the types of the arguments, producing a redundancy), we follow Algol 68 in introducing an abbreviated form of function declaration, writing T :< *name* @ (*args*) >: *body*;... for T[*args*] name: (*args*) >: *body*;....

[10]This corresponds to the rational numbers, whose decimal expansions all eventually repeat (and the manipulation of which requires distinct algorithms from the manipulation of general decimal expansions, for this very reason).

[11]This strategy corresponds to the constructive reals, which are apparently as close an approximation to the classical reals as we can expect to see implemented directly.

[12]Corresponding, of course, to the classical real numbers in all their glory.

3.1.2 Data structures

In many, perhaps most, general purpose programming languages there is a close parallelism between the available data structuring and code structuring facilities (as we shall see in section 3.2); this turns out to be a very useful consequence of the natural constraints on data structure selection for von Neumann machines. In *Exemplar*, using the freedom of a paper notation, we have made the syntax of the two fully coïncident.[13] Here, then, we examine the data structuring facilities as they parallel the control structures of the last section.

Sequencing: While the variety of "sequencing" that arises naturally in control structures is normally interpreted as *temporal* in nature (though in effect it amounts "merely" to causal (partial-) ordering[14]), conventional data description languages also exhibit a natural sequencing operator, this in the (abstract, address-) spatial domain. It is the operator that constructs *structures* or *records*.[15]

In *Exemplar*, the spatial sequencing of data structures is written with the same (textual) operator as the the sequencing of code: the comma. Type grouping, similarly, is written with parentheses: (). Conceptually, this construction results in the various fields of a structure being laid down in sequence, and it restricts information to passing between them in left-to-right order.[16]

[13]A few technical problems remain in implementing such an arrangement in a practical language, though we have every confidence that they can in fact be resolved.

[14]And is thus very much like the notion of time familiar from Leslie Lamport's work on synchronisation [Lam78].

[15]In actuality it is *also* a causal partial order: just as a code sequence is causally ordered by (potential) effect, data sequence is ordered by potential control. An interesting situation can arise when two structures of compatible control orders each have elements of the same types in the same positions: they can be understood to be overlapping types, neither containing the other, where the values in the intersection are at least freely interconvertible. Recall, for instance, that the intersection of two regular grammars is not in general regular.

[16]Formally, just as in the code case, the comma operator is *binary* and the parentheses have no semantic function at all; since Cartesian product (and even its dependent analogue, $\dot{\times}$, which we shall meet in chapter 4) is associative up to isomorphism, and since field name scoping in *Exemplar* is decoupled from bracketing in pattern contexts, this presents no formal or practical difficulty.

Selection: The part played by the union constructions of Algol 68 and C, and by Pascal's variant records, is taken in *Exemplar* by the type-valued *if*-construction. As with its code-structuring counterpart it has two forms, depending on whether the controlling condition is simply boolean or relies on pattern matching. Thus we have:

```
if condition then substructure_true else substructure_false end
```

and

```
if expression
is pattern_1  then substructure_1
is pattern_2  then substructure_2
      ⋮
is pattern_n  then substructure_n
else substructure_default
end
```

In both these constructions the data structure that is actually instantiated in a particular value is a function of the expression following the *if* key.

In the case of control structures it is clear enough (on the basis of the accumulated traditions of programming language notation, at least) what data the controlling expression is permitted to depend on: basically, on any names declared in scopes surrounding the *if*-construction itself (in a modern functional language it is generally guaranteed that these will still be "live" and accessible, regardless of cost: in fact this is quite acceptable because the compiler can take steps to ensure that no overhead is involved dealing with variables that are *not* captured). Furthermore, if any of these variables is mutable (a conventional, "impure" variable, *i.e.*, the value of a single instance of which can change over time), its value at the moment of evaluation may be extracted and employed.

The situation with data structures is somewhat different, since each time a data structure is examined it must exhibit the same structure[17] (unlike an impure "state-

[17]We here ignore the (independent) possibility that a datum might be accessed under more than one type, a possibility that is not *necessarily* in conflict with strong typing, as we shall see.

ment," which is in general data-dependent even within an instance). It is thus presumably inappropriate to permit structures to depend on the values of mutables, or other impure expressions. Since data may furthermore be examined asynchronously with respect to code on either side of an abstraction barrier which they cross—this being the general property that makes the issue of choice of argument passing conventions significant—they must also depend only on *nice* values, not *changing* the values of mutable objects, lest their examination affect the behaviour of remo‘e parts of the programme. This does *not* imply that there is any need of an injunction against references to variables (in the mathematical, non-mutable sense) defined above and to the left of a controlling expression, whether these variables be other, constant fields of the data structure, explicit arguments to the structure, or values from scopes enclosing the structure declaration itself (though in this last case—as in languages like C++, though for different reasons—the familiar transparency of the relationship between the data structure as declared and the underlying representation in memory is necessarily impaired, since the data structure must be closed in basically the same manner that function closures are constructed). In sum, the control expression must be (in effect) both *closed* and *fixed*.

There is an additional tradeoff: since a typical data structure is examined far more often than once (in the presence of, say, a garbage collector, it may be examined arbitrarily often, and at a moment when computational resources are particularly scarce—see chapter 7), it will often be advantageous to store the results of any auxiliary expressions used in building the type, in the structure itself (once again at a cost in transparency of implementation), rather than recomputing them each time the structure is accessed. Obviously this decision should be based on information about the actual complexity of the expression on the host architecture, and the expense of the storage—saving one or two instructions will certainly not be worth the cost of an extra field, while saving a call to a statically undeterminable user-defined function assurèdly will. This technique, of caching the results of complex expressions within structures, could also be employed to forcibly fix naughty or impure expressions to their meanings at the time of structure instantiation, if permitting them at all were considered desirable.

The overall effect of this caching method is to impose *at the implementation level* a restriction, possibly to triviality, on the complexity of control expressions in structures, but in such a manner as to be transparent to the programmer.[18]

Iteration: The faculty provided in other languages by arrays and their analogues manifests in *Exemplar* as an iterative (linearly inductive, if you will) structure completely parallel to the looping constructs for statements—though at some points in this thesis we shall assume for convenience' sake that a specialised type constructor, Vector, has also been provided to this end, as part of some standard library.

From a certain perspective this is a somewhat peculiar unification of concepts, since loops are generally thought of as *serial* constructions, while arrays are contrasted with lists for their random accessibility of elements. In point of fact, the relationship is quite straightforward, for only fully homogeneous arrays lack an informational ordering in this way (and these correspond only to *fully parallelisable* loops). In the first place, though random access may indeed be possible to an iterative, array-like structure, many operations for building or manipulating arrays *as a whole* work sequentially, visiting each element in turn in the very sequence that they are reached in their defining construction. Secondly, it is apparent that not all such structures *are* randomly accessible, at least not at uniform cost; the LISP list *is* a kind of vector, and with its linked structure is the most apparent such example, but the C string—at first glance a conventional array—can only be *safely* indexed if a prescan has been performed to locate its terminating NUL and thus the upper bound on legal

[18]See section 5.7.3 for an explanation of how this can be assured even in the case where it is arranged for physical storage layout to be specified in detail.

indices.[19] Finally, it can be established that the cost of field addressability is in fact an implementation question entirely (though certain expectations and even certain guarantees are not unreasonable): as a limiting example, consider access to fields that are (due perhaps to an *if*-construct) only conditionally present. Here, superficially constant-cost access can be requested at the source level (language permitting), even to entities which at run time do not exist(!)—a circumstance that can be expected to prevent the expression from ever terminating.

Thus constant cost access to array elements is a joint property of the chosen implementation and (possibly) the naming (*i.e.* indexing) convention designating the elements. As with the indexing of conditional structures, *Exemplar* permits this naming convention to be quite arbitrary (though it must be unambiguous), reserving the right to maintain a copy of the naming expression if it is too complex by some implementation-determined criterion; resulting in this worst case in an indirectly-indexed array.

To take an example, an array of ten integers would be rendered thus:

for N i *to* 10 *do* Z [i] *end*[20]

and would presumably receive precisely the conventional representation (the fact that $\lambda i.i$ is a trivially computed transformation, not requiring representation, being, we hope, obvious to the compiler).

For somewhat more complex examples, see section 4.6.4.

Declaration and invocation: As below, so above: the declaration of new, named types is accomplished in *Exemplar* through precisely the same mechanism as the

[19]Although there are perfectly sensible C programmes that manipulate "string" addresses beyond the final NUL (in particular, consider those involving strcat()), these must be understood from a formal perspective as manipulating an underlying (conventional) array in which the string elements are stored and whose bounds must be known *a priori* to the programme. See section 4.6.4 for further discussion of C strings within our present framework.

[20]Here "[i]" is the *name* of the element and this "i" is an *expression* rather than a new identifier.

declaration of functions, except that the "type" of a type is TYPE. Like functions, we permit types to be parameterised, but as with all structure specifications these type-valued functions must be both pure and nice. Thus we write:

```
    TYPE :< Vector @ (TYPE T, N limit) >:
      for N i to limit do T [i] end
  ; ...
    Vector[Z, 10] x : ...              .. An array of ten integers, x[0] to x[9]
  ; ...
```

Once again we find that several interpretations of a type are possible in the presence of circularity: corresponding to the members of a type, we may consider as the "instances" of an expression its evaluation histories. Thus the same recursive type declaration could be interpreted as admitting (1) only well-founded instances; (2) instances that are either well-founded or actually circular (and thus having strictly finite information content);[21] (3) all constructive instances;[22] or (4) completely arbitrary objects (which will in general prevent the programme from terminating normally).

3.1.3 Discussion

In *Exemplar*, then, we find a formalism in which the notations used for expressing code and data structures are strictly parallel. If one thinks in operational terms, the sequence in which statements are executed in a programme corresponds to the sequence in which the fields of a structure are "laid down" in memory; if in functional terms, as control passes down into the leaves of the (value-) expression tree and values pass back up, so control passes down into the (type-) tree and the constructed structure is returned on retraction. Thus the *trace* of an executing programme—if represented with each function invocation on a separate track or "tier"[23]—corresponds to the *instances* of a data structure.

[21]Such are the data structures constructible in in most conventional programming languages.

[22]This is the behaviour exhibited by provably terminating programmes in normal reduction order functional programming languages.

[23]The origin of this notion is the observation in autosegmental phonology—see, for instance, [Gol90]—that independent morphemes are best analysed as at least originating with independent autosegmental tiers. The analogy is surprisingly tight.

In the sections that follow we shall see that this formal relationship between code and data, while sometimes obscured by the details of the syntax, exists in many other languages and notations of computer science. First, however, we indulge in a brief aside on the subject of the non-obvious compilation of types to types and code to code.

3.1.4 A note on non-transparent compilation

It is perhaps worth our while to take a few moments out for the consideration of an example of how non-transparent representation might arise straightforwardly (especially since it is traditional for compilers to be naïve about data structure representation in general). Consider the compilation of an expression (x + y) re (for x and y in Complex[N]), in a context where speed optimisation has been determined to be necessary. [24]

It is possible that an advanced compiler having a rule-based transformational module would be aware of the distributivity of (re)-projection over Complex addition.[25] Far more mundanely, even a conservative compiler might elect to inline the complex addition operation, or otherwise subject it to joint (interprocedural) analysis with its calling context. Once this step has been taken, normal compilation processes will result in the *real* portions of the addends being loaded into registers for manipulation, while the imaginary part of the result, the computation deriving it, the (never existing!) registers holding the imaginary components of the operands and the conceptual RAM backing them are all removed by dead-value elimination.

The overall effect, then, is of a series of complex numbers being represented as

[24]This is not the same thing as speed optimisation having been requested by the user: one of the amusing facts about optimisation is that on a virtual memory platform, better time performance can result from performing *space* optimisation on the less-used portions of the code, to increase spatiotemporal coherence (see [PH90] for similar considerations).

[25]In the absence of exceptions, at any rate; had we used a bounded-precision integral representation which fails by signalling (rather than by returning an exceptional *value*, the arrangement favoured by the author for precisely this class of reason) in a context handling arithmetic exceptions, this transformation would have had to be blocked.

integers in registers.

Even more dramatic effects can obtain in the case of Boolean values, which frequently disappear into the control flow graph of the translated programme. A sufficiently sophisticated compiler, however, might even end up *substituting* one type for another throughout an entire programme: consider the case of a (textual) symbol that is repeatedly looked up in table throughout the course of a calculation, to determine different characteristics of the entity it represents. The modularisation of the programme may be such that the programmer has no choice in this: the symbol table accepts a name and an attribute class and returns a value in that class. The compiler, however, would have access to the internal representation of the table, where the various attributes of each symbol are in all probability grouped together into a single structure. A form of global common subexpression elimination would result straightforwardly in all references to the symbol table (after the first) being replaced by references to the appropriate entry in that table—and subsequently by the substitution of references to the internal symbol descriptor replacing stored values of the symbol's textual identifier in the various data structures of the programme.

3.2 Structures Elsewhere

Now let us examine the various structures that appear in several more classical notations of informatics.

3.2.1 Classical FØRTRAN

FØRTRAN arrived on the scene before the advent of the dogma of structured programming, and (at least originally) was not arranged in such a way as to highlight structure in the sense that we have been using the term (a fact that is in some ways quite ironic, since FØRTRAN seems to be the language on which the greatest efforts at optimised implementation have been focussed, and it transpires that the hypothesis that structured programming facilitates automatic optimisation was easier to establish than that it facilitates software development). In recent years, of course, FØRTRAN has married into the Pascal branch of the Algol family. The (ancient) dialect we consider here is FØRTRAN IV [ibm].

The most clearly "structured" facilities of FØRTRAN IV (though they do not manifest as inviolable blocks, by any means—see our discussion of the GØ TØ statement and its correlate, below) are array variables and the DØ statement (or, rather, the whole loop that it heads). Though the language does not make the relationship notationally explicit,[26] these two facilities, one for code and one for data, fall in the same parallelism as their consciously homographic analogues in *Exemplar*. It is clear, furthermore, that such was the intention of the language's designers: manipulation of arrays in the absence of loops is next to impossible, and loops in the absence of arrays are of limited utility.

Although FØRTRAN IV lacks data abstraction facilities as such—there is a limited vocabulary of built-in types, adequate to numeric calculation, no more—it does provide for the introduction of nonprimitive functions and subroutines, subject to the understanding that they not be recursive. This provides for fairly arbitrary re-use of code segments within the programme, though the failure to require provisions for recursion means that only one "copy" of the code can be used at one point in the programme trace. Under a similar restriction—that only one *instance* of the structure be active at any point—arbitrary FØRTRAN IV *data* can be named and reüsed, with the "labeled CØMMØN" facility. As with much else in FØRTRAN,[27] it is a no-seat-belts mechanism: each CØMMØN declaration is actually in a position to reïnterpret the contents of its common block(s) arbitrarily, but the provision of an independent EQUIVALENCE statement (and the requirement that named common be of globally

[26]except, perhaps, in the "implied-DØ" construction, wherein the two notations are hybridised.
[27]and in common with C, where type consistency, at least between compilation modules, relies on "good programming practise" rather than automatic checking.

uniform length) seems to suggest that the intention was to provide something very similar in spirit to the SUBRØUTINE.

Finally, and fascinatingly, there is a striking parallelism between two characteristic *unstructured* features of the language: the GØ TØ statement for arbitrary transfer of flow of control, and the EQUIVALENCE declaration, which identifies separately declared data objects by fiat. Each of them can be interpreted as "splicing" arbitrarily selected (but nameable) elements out of one stream of interpretation into the ongoing history of another. The "original future" (to embezzle a phrase from stories of time travel) of the stream of interpretation is lost. The relationship is imperfect, since EQUIVALENCE has, on the one hand, impact only on the objects named (and the things they end up overlapping); but on the other, extends "backwards," in that arbitrary elements of arrays may be equivalenced—whereas a GØ TØ has impact only on what is, from its perspective, the future of the calculation. This is, of course, a consequence of the different temporal paradigms of homogeneous arrays and (side-effectual) loops: as has already been noted, the informationally induced ordering on homogeneous arrays is empty, while that on loops is in general total. In fact, since FØRTRAN IV lacks conditionally or multiply instantiated data, *all* data structures lack informational ordering at runtime (though of course *values* are dynamic, and interrelated by programme execution; and the precise effects of EQUIVALENCE declarations are crucially dependent on the source ordering of variable declarations).

3.2.2 Pascal

The programming language Pascal [JW74] provides a much richer type system than did FØRTRAN IV. In Pascal (unsurprisingly, since the languages share Algol ancestry), we find direct analogues of a number of *Exemplar*'s constructions: there are identifiable grouping, sequencing, selection and iteration operators in the code structure:

```
begin ... end
```

```
;
case ... of ...:... {;...:...} end                    if ... then ... [else ...]
for ... : ... to ... do ...                              while ... do ...
```

and in the type language:

```
record ... end
;
case ...: ... of ...: (...) {; ...: (...)}
array [... {, ...}] of ...
```

both—though in the data case the various functions lack sharp mutual definition.

Abstraction in Pascal is available through function, procedure and type definitions. Once again, however, there is no strong sense of the relationship of code and data in the language: type definitions have a notation distinct from function, constant and variable declarations (which are, further, mutually distinct)[28]. No mechanism is provided to parameterise user-defined types, and because of the transparent storage representations which Pascal assumes, recursion is only permitted through pointer types (in fact, languages that provide unrestricted recursion in types will in general need to implement them with pointers, except in the case that tail-recursion-elimination—possibly following field re-ordering—succeeds).[29]

3.2.3 EBNF

The familiar syntactic specification language (or family of such languages) known as EBNF—Extended Backus-Naur Form—would not normally be understood as a programming language; but it, like Prolog, can be seen as an underspecified procedural notation (a notion that has been given concrete reality—though not for EBNF itself—in the Unix tool, yacc). The grammar specifies what analyses *might* be arrived at

[28]Type declarations are most similar to constant definitions—at least in so far as they use the separator '='—but, incomprehensibly, arithmetic expressions are forbidden on the right in constant definitions, but permitted in actual parameters—while type expressions are permitted in type definitions, and forbidden in formal parameters (and, for that matter, as the return types of functions, as the types of discriminant fields in variant records and as the arguments of the '↑' (or '^') pointer type constructor).

[29]This manner of scheme sees implementation in the representation of lists on the LISP Machine [Moo85].

by a parser (or traces followed by a generator) of a string; and for any *given* string the "correct" analysis determines a recursive left-to-right traversal of the productions very reminiscent of programme execution.

The structuring operators of (a typical instance of, and the one we employ in this thesis) EBNF include a **grouping** operator, $^{(}\ldots^{)}$; a **sequencing** operator (in our notation simple juxtaposition, but often ','); two **selection** operators (the **one-sided** $^{[}\ldots^{]}$, corresponding to *if* ... *then* ... *end*, and the **two-sided** $^{|}$, corresponding to *if* ... *then* ... *else* ... *end*); and an **iteration** operator, $^{\{}\ldots^{\}}$. The part of remote reference is, of course, played by the nonterminals themselves, and the productions defining them.[30]

The analogy with programming languages is precise: on the one hand, we can consider a conventional (procedural) programming language abstracted of its expression syntax: the remaining skeletonic control structures (the very things on which we have been focusing in this chapter)—which, lacking expressions, must now receive a non-deterministic interpretation—are exactly the operators of EBNF (that is, the "usual" constructors of regular expressions, extended with nonterminals—in this case, function calls). The strings generated (or parsed) by the grammar are the statement-by-statement execution traces of the programme, and their parse trees are call trees.

Similarly, excepting pointers (which actually correspond to a second, incommensurate, alphabetic interpretation of the underlying string—see section 5.3) there is a clear relationship between typical data structure declarations and EBNF grammars: the transparently related grammar *is* the grammar of the data instances corresponding to the structural type description. The strings are the objects. The formal distinction between the two systems resides again in the locus of control: (computer-scientific, parsing) grammars generally obtain their control information through the

[30]Actually, in order to drive our analogy through, we need to assume (without loss of generality) that each nonterminal is defined by a *single*, possibly disjunctive, production—or that clausal definition is allowed for code. But see the next section.

supplementation of their leftward syntactic context with information derived from lookahead—not an option in a type system, where examination of unidentified data is impossible. Conversely, control information in data structures is in general *computed* from data in the left context—something not in general possible for (finite) context-free grammars.[31]

3.2.4 Minimalist structuring: Core LISP, Prolog and BNF

Not every complete programming language—or syntactic metanotation—has such a broad range of structuring operators as those we have just described. Minimally, it seems, one could get by, in conjunction with the ability to perform arbitrary recursive embeddings of structures within each other, with only two such operators: a **product** operator, juxtaposing structures in a sequential manner; and a **sum** operator, combining them as alternatives.

While, strictly speaking, these minimal operators need only be binary, they are both associative (up to isomorphism in the objects they describe).[32] In their notationally less cumbersome variadic forms, they turn out to be very familiar structures indeed. Interpreted (as we did with EBNF) as a grammatical formalism, these operators are precisely those of BNF, the usual minimal notation for context free grammars: the product operator is the separator between the elements of each right-hand-side,

[31]Note, however, that the grammars of computationally influenced linguistics frequently *do* permit at least unification-based attribute computation to control parsing [Bre82, GKPS85, Tom87]. Furthermore, the analogy between data structure declarations and grammars is most clearly as we have described for LL parsers—LR parsers exploit a great deal more nondeterminism (at the source level, not in the implementation—though [Tom87, Tom91] uses virtually parallelised LR techniques to parse arbitrary context-free grammars), in the process obscuring the relationship to the original source text.

[32]It might be thought that the sum operator was also commutative and idempotent. Remember, however, that in the present discussion we are abstracting away from control information which can select between intensionally distinct, if extensionally identical, trace futures. If we permit the collapse of these branches (on the grounds that they represent the "same thing happening"), we must also admit that the *product* operator shares these same two properties. As a practical matter, the assumed idempotence and commutativity of (in particular) type union turned out to be one of the greater misfeatures of Algol 68. On the other hand, the weakening of the basic composition operators to their extensional behaviour is the theoretical essence of code-motion optimisations in code generation; but code generators operate at the extensional level essentially.

while the sum operator is the (implicit) combination relation holding between the various productions defining each nonterminal.

As control structures, we see the minimal arrangement underlying the notationally simpler members of both the functional and relational classes of contemporary programming languages; and with other rule-based systems, however classified. Prolog programmes, for instance, have precisely the structure we just described for BNF: once we have abstracted away from the details of control determination, relations are defined by lists of (alternative) clauses, each a sequence of (references to) other relations. Prolog programmes are, of course, non-deterministic (or, more strictly speaking, exhibit largely reversible execution[33]); but fully deterministic programmes can be written with a structurally similar notation: consider, for instance, a simple functional programming language in which functions are defined clausally, with clause selection performed by pattern matching on the arguments.

The two-operator arrangement similarly appears in a familiar notation for data structures: that of LISP. LISP values are (implicit) disjunctions over all LISP objects, and the nonprimitive objects are (canonically) lists of LISP values. Extensionally, of course, LISP values are single-sorted—LISP is a *unitypic*, "untyped" language—but this implies merely that the type of values comes to the programmer closed up under our primitive type constructors. The very power of LISP data structuring, and its pragmatic weakness, arise from the fact that *all* of the other structures we have been describing are mathematically reducible to this formalism.

3.2.5 Summary

In this section, we have seen that the "usual" repertoire of structuring facilities—whether for code, for data, or, as it transpires, for the description of context free languages—are all drawn from the same stock. On the one hand, they are all re-

[33]The fact that they are really deterministic and *sequential* actually provides a paradigm case for the *non*-commutativity of the sum operator, as described in footnote 32.

ducible to the operation of just two operators,[34] a sequencing operator and a selection operator; on the other, more extensive, "practical" notations for their description generally turn out to employ a quite uniform collection of ideas: definite and indefinite iteration, distinct but related one-, two- and many- sided selection, and, sometimes, functional abstraction.[35] In the preceding section we exhibited a pair of identical notations—or, rather, two applications of the same notation—for the description of the code and data objects of a single programming language.

There appears, then, to be a very profound structural similarity between programme structure, data types, and grammars; and, correspondingly, between programme execution traces, particular data objects, and strings. What we have abstracted away to exhibit these similarities is control: the actual computational mechanism by which one trace, object or string is selected from the various "possibilities" licensed by the defining structure. This is in fact precisely to be expected: for though the relationship between strings and data is quite transparent (we are familiar enough, after all—under the influence of untyped operating systems such as Unix as much as from our experience with paper notations—with strings as external representations of more general data), a data structure is related to code in being somehow more "fixed," in having had its conceptual indeterminacy transferred from the "temporal" execution domain to the "spatial" data domain.[36] We will return to this theme, in section 3.5 below.

In the next section we present **Visitors**, objects that encode type very generally as code, and arise from the taking seriously of code/data parallelism, as a matter of practical import.

[34]for reasons rooted in category theory, if you happen to belong to that school which believes that category theory *explains* anything.

[35]One might speculate that the explanation for *these* recurring themes is to be found in the particular linguistic processing apparatus with which the members of our species are endowed.

[36]Nowhere, of course, is this idea more manifest than within the implementation of a graph reduction interpreter.

3.3 Visitors

Having established the existence of an isomorphism between code and data structures (or, more strictly speaking, between the formalisms with which they are typically described) we now examine its practical nature. Considering first the mapping from code to data, we see that, as already implied, the data structure description produced by the direct mapping of the *terminal* statements of a code segment into field specifications—made preserving its formal structure—is essentially a concrete data representation suitable for the (naïve) recording of the segment's execution traces. The crucial control expressions in the structuring operators can be carried across directly, and serve to elucidate the actual resolution of the potential choices available to the thread of execution; true nondeterminism in the execution of primitive expressions is represented by making a record of their results, from which dependent control decisions can be reconstructed precisely.[37]

The converse mapping, from types into code, is (at least if we are—as we have been silently throughout this chapter—ignoring the complications engendered by such implementationally opaque entities as function values, the discussion of which we shall actually defer—justified by that very opacity—until section 5.4) in fact more straightforward; but as the primary subject of this thesis we shall devote to it considerably more attention.

The code structure obtained by direct mapping from the data structure description has precisely the form required of a programme designed to traverse the data in depth-first, left-to-right sequence. The overall effect of this traversal is determined by the (as yet unspecified) mapping from the primitive—terminal—data fields to the statements or expressions of our executable image of the type.

Applying now a powerful software engineering technique—that of delaying deci-

[37]This is precisely the "fixing" operation of section 3.1.2. Once again we must finesse the handling of imperative-style variables: as entities with non-local semantics we must treat them as (locally) non-deterministic values, which are thus also "fixed" by copy into trace fields.

sions until the last possible moment[38], we introduce the **Visitor**: the most general homogeneous translation of a data structure description, obtained by direct out-abstraction—into a set of functional parameters—of the details of leaf rerepresentation.

The **Visitor** of type T is a function β_T of type[39]

$$\mathcal{P} \times \mathcal{A}[\ulcorner T \urcorner] \times \mathcal{M} \to \mathcal{P}$$

where

$\mathcal{M}$ is defined to be $\mathcal{T} \dot{\to} t.\mathcal{P} \times \mathcal{A}[t] \times \mathcal{M} \to \mathcal{P}$, the type of visitation **controllers**.

$\mathcal{P}$ is the type of in-core data representations (pointers, in a typical physical implementation),

$\mathcal{T}$ is the type of names of type constructors (including atomic types as the zero-argument boundary case), $\{\ulcorner T \urcorner\}$, where $\ulcorner T \urcorner$ is the name of a particular type (or constructor) T,

$\mathcal{A}[t]$ is the type of the argument list of the type or type constructor whose name (in $\mathcal{T}$) is t

$\dot{\to}$ designates a *dependent map,* analogous to a function type, but in which the type of the result is allowed to depend on the *value* of the argument—t, here, in $\mathcal{A}[t]$—an arrangement that is explored more fully in section 4.6.2.

[38]And implementation longer still, contextual slack permitting!

[39]**Visitors** will ultimately be used to invoke semantic actions on behalf of the caller. In order that these be useful, they must be sensitive to the context of the original call into the visitation system. Typically, this implies either that the semantic routines be closures with access to state variables, or (in the side-effect-free style we prefer) that the **Visitor** itself be augmented by the pairing of each pointer argument of type $\mathcal{P}$ with a semantic argument of some type $\mathcal{S}$, which is in turn a parameter of each invocation of the **Visitor** system. Thus the corrected type of β_T is $\mathcal{P} \times \mathcal{S} \times \mathcal{A}[\ulcorner T \urcorner] \times \mathcal{M} \to \mathcal{P} \times \mathcal{S}$. In the interest of simplicity (and since the object of type $\mathcal{S}$ is—from the perspective of the **Visitor**—simply passed through the entire callgraph without change) we omit this argument throughout the present exposition.

The *text* of β_{T} is structurally isomorphic to T's original description, with each atomic field mapped into a call to a functional parameter, $\mathcal{M}[\ulcorner \mathsf{U} \urcorner]$, where U is the field's type. It should be noted that this function deals with internal objects of the language and is *not* well-typed at the source level (though we are using source syntax for expositional purposes): $\mathcal{P}$ in practise is an (opaque) image of underlying machine pointers, while $\mathcal{T}$ and $\mathcal{A}[\,]$ provide access to type information drawn from the (link-time) symbol table. In the applications we envision, type security is not compromised because **Visitors** are generated automatically from type declarations and are not made directly visible to the programmer.[40] Specialisations of the visitor type (with $\mathcal{P}$ replaced by particular user types) can be made available to the user on a context-by-context basis, though we have not yet devised an elegant notation for the purpose.

3.3.1 Mapping types to Visitors

In this section we give the algorithm by which the **Visitors** for *Exemplar* are constructed. Since *Exemplar*'s structures generalise those of most languages, this algorithm will apply *mutatis mutandis* to them as well, though it can be expected to suffer somewhat in terms of transparency.

The algorithm is given first informally in English, then as an *Exemplar* fragment implementing the computation for a kernel of the language. The programme as presented assumes that all loops have been reduced to simple *while*-loops, and that executable objects have had representations chosen for them from within the kernel language, in the manner discussed in section 5.4; an actual implementation would in all probability implement more distinct cases, but this would seem to be without expository value.[41]

[40]**Visitors** thus represent a structure that is not well-typed (and in fact not well-typable), which the programmer would not be in a position to write and which could not be computed automatically if the source language were not itself strongly typed. What weak typing provides on the manual system, strong typing here provides "automagically."

[41]In fact, our hope is that the translation can be implemented *metarecursively*, as a visitation, removing the issue of determining an appropriate covering set of structure translations completely

The Argument Packet: The first step in constructing a **Visitor** for a type or constructor T is the determination of the form of its argument packet, $\mathcal{A}[\ulcorner T \urcorner]$. If T is in fact a fully instantiated type, this is straightforward: no auxiliary information need be conveyed and $\mathcal{A}[\ulcorner T \urcorner]$ is just the empty structure, (). Otherwise, we must be visiting a constructor, and the argument packet contains an argument for each argument the constructor takes. For those parameters that are not themselves types, the type (and ultimately the value) of this argument is exactly as for the constructor. Type parameters (which arise for carrier types like Vector or List, and for the primitive constructors % and #), however, are mapped not into arguments of "type" TYPE, but pairs of the t in $\mathcal{T}$ that is the symbolic *name* of the original actual parameter, and *its* actual argument packet (which is of type $\mathcal{A}[t]$).

The Visitor Proper: Informally, the algorithm that relates a type T to its **Visitor**, β_T is as follows: each structure in the type description is mapped to the corresponding code structure in the **Visitor**: loops are mapped to loops,[42] conditionals to conditionals, ()-groups to ()-groups. Each such structure is now considered a "leaf field" with respect to its immediate parent; that is to say that the following discussion of leaf field translation applies to each argument of a structure, whether compound or simple.

The overall function β_T is, as already noted, of type $\mathcal{P} \times \mathcal{A}[\ulcorner T \urcorner] \times \mathcal{M} \rightarrow \mathcal{P}$; let the

from the implementor's concern. Such a presentation is not given here, however, because of unsolved problems in designing a suitable source-level syntax for invoking visitation from the applications level.

[42]Since (type) *do*-loops are not yielding constructs—they always have type Void—while **Visitors** in fact have values, we translate

...*do* U u...

to

...*with* $\mathcal{P}$ p *from* p_1 ... *then* p... *as* $m[\ulcorner U \urcorner][p, a_1, m]$...

using a (yielding) accumulation construction (the derivation of the argument fields in this translation is regular, and is given immediately below). Similarly, where fields are declared inside other iterated fields—such as in *while*-clauses they are moved out into *with*-*as* blocks.

arguments be named p_0, a and m, respectively. We can now describe the translation of leaf expressions.

The image of a field declaration of type U is a local constant declaration of the form

$$\mathcal{P}\ p_{i+1} : m[{}^{\ulcorner}U^{\urcorner}][p_i, a_i, m]; \ldots$$

where p_i is the pointer value introduced by the translation of the immediately preceding field. Here m is the original m from the **Visitor**'s own argument list, and a_i is the argument packet appropriate to the field being visited.

If the leaf is not a field but an expression (or in the case—such as the argument of a parameterised field type—of an expression embedded within a field-leaf), it is passed through unchanged but for the variables it mentions. If a variable is a parameter of the type it is rewritten as a reference to the corresponding entry in the **Visitor**'s argument packet; otherwise it must be a reference to a field v of some type V within the structure,[43] and it is rewritten as $\delta_V[p_v]$, where p_v is the p_i passed to m in the visitation of field v (*not* the result of that call), and δ_V is a function provided by the implementation that provides field access at visitation time.[44]

If any expression is too complex (in a sense that can be decided arbitrarily by the implementor, if exact structure layout is not required—though expressions that are not both pure and nice are necessarily in this category, if permitted at all) it can be simplified, at some cost in space, by recasting it as a stored field initialised to the

[43]If types are permitted to bind variables—if they are permitted in local scopes—then the underlying type structure must presumably have been rewritten, for implementation purposes, in one of two ways: either the context dependencies have been made parameters of the type, or they have been made accessible (directly or indirectly) through one or more fields added to the structure. These are the same two basic techniques (respectively the combinatorial method and the closure method) available for the implementation of function closures (which suffer precisely the same difficulties. In fact, if an activation record is viewed as a structure the two situations collapse). The present algorithm is assumed to apply *after* this transformation, and so there will be no additional complication. We can assume that variables are scoped no wider than the parameter list of the type.

[44]Consistently with our declaration-before-use policy, there is a restriction placed on δ_V, that it only be called *after* v's own visitation. From a theoretical viewpoint this ensures that the data structures of which the **Visitors** are the images are uniquely defined; from a pragmatic standpoint it makes it feasible to use **Visitors** to perform transput, for instance, where unvisited fields may not be available to the δ operator.

```
TYPE Code : Expression .. Extended by...
  { Code operator ? Code operand                              .. Formal application
  , Code left ?, Code right                                                  .. a, b
  , code_if Code condition then Code sequent end
  , code_if Code condition then Code sequent else Code alternate end
  , code_while Code condition do Code body end

  , Code pattern ?: Code value; Code rest
  , [[Code code]]                                                            .. [x]
  , ...                                          .. Other value-expression operators
  }

.. Support functions ..

; Code :< typeConstructor @ Code type >:
    ...                                .. Given a type expression, return its head
; Code :< typeArguments @ Code type >:
    ...                           .. Given a type expression, return its argument list
```

Figure 3.1: Abstract syntax of *Exemplar* as seen by visitor

value of the expression, now followed by a reference to the new field. This forces the evaluation of the expression to be performed at structure creation time, rather than during the execution of the **Visitor**.

The value returned by the **Visitor** is the final p_k.

In the examples of the following section we edit the output of this algorithm to reduce the number of auxiliary variables introduced and to keep the code as readable as possible.

The Translation in *Exemplar*: In this section, we assume that the abstract syntax of *Exemplar* data structures has been represented with the type shown in figure 3.1. In order to avoid notational overhead from manipulating variadic operators, the type descriptions have already been simplified by (1) reducing *if* $^{\{}$*is then*$^{\}}$ $^{[}$*else*$^{]}$ *end*

```
Code :< visitor @ (Code type, Code p0, Code m) >:
  .. Compute the visitor for a (simplified) type expression
  if  type
  is  Code type ? Code pattern                              .. Base case
  then
      ( m ? [[ typeConstructor[type] ]])
    ? [[ p0 ?, (typeArguments[type] ?, m ]]
  is  Code left ?, Code right                                .. Sequence
  then
      Code p1 : visitor[left, p0, m]
    ; visitor[right, p1, m]
  is  code_if Code condition then Code sequent end
  then
      (Code c, Code p1) : visitExpr[condition, p0, m]
    ; code_if c then visitor[sequent, p1, m] else p0 end
  is  code_if Code condition then Code sequent else Code alternate end
  then
      (Code c, Code p1) : visitExpr[condition, p0, m]
    ; code_if c
      then visitor[sequent, p1, m]
      else visitor[alternate, p1, m]
      end
  is  code_whileCode condition do Code body end
  then                                         .. This is the one subtle case
      (Code c, Code p1) : visitExpr[condition, p0, m]
    ; Symbol p : new[]                          .. New induction variable
    ; code_with 𝒫 ? p
      from p0
        while c
        then p
      as visitor[body, p1, m]
      end
  end
;
```

Figure 3.2: Simplified *Exemplar* code computing a Visitor from a structure

```
 (Code, Code) :< visitExpr @ (Code c, Code p0, Code m) >:
   .. Visit an expression embedded in a type
   .. An uninitialised local is a field of the parent structure
   if  c
   is  Code type ? Code pattern
   then
       .. Special case: both an lvalue and an rvalue!
       Code p1 : visitor[c, p0, m]
     ; [(deref ? [[p0]]), p1]
   is  Code type ?, Code expression
   then
       Code p1 : visitor[type, p0, m]
     ; visitExpr[expression, p1, m]
   is  Code type ?: Code value; Code rest
   then
       (Code c1, Code p1) : visitExpr[rest, p0, m]
     ; [(type ?: value; c1), p1]
   else
       [c, p0]                    .. Actually need to be careful to insert derefs
   end
;
```

Figure 3.3: Auxilliary function of figure 3.2

constructs to equivalent applications of *if then* $^{\lceil}$*else*$^{\rceil}$ *end*; (2) reducing all loops to applications of *while do end*; and (3) providing each field with a single unique name (eliminating the use of ':: ;' and anonymous fields).

The visitation of functions is, as already noted, not handled at this level, since **Visitors** are computed from storage layouts, which in the case of executable objects is only determined during actual code generation; this topic is discussed in detail in section 5.4.

The actual calculation of the **Visitor** is performed by the function visitor in figure 3.2; it employs the auxilliary function visitExpr of figure 3.3 to handle expressions embedded within structures (typically control expressions).

Note that when the code computed for **Visitors** (*i.e.*, the output of these functions) is eventually generated (elsewhere in the compiler), *all* abstract dereference operators applied to type %T must initially be translated into explicit calls to δ_T, though as we shall later (see especially section 5.6.5) the basic machine operation can often be recovered in optimisation, depending on the application to which the **Visitor** is ultimately put.

3.3.2 Sample translations

In this section we illustrate the effect of the algorithm of the preceding paragraphs through a number of examples. These translations are once again exhibited at the level of source notation; symbols such as $\mathcal{P}$, $\mathcal{M}$, β_T, δ_T and $^{\sqcap}$ should be understood as extensions to the source language for the purpose of capturing such low level concepts as "a position within the representation of an object"; they arise because, strictly speaking, **Visitors**, operate at the level of the implementation, and not within the user's domain at all. This minor abuse of the notation will, however, permit us to defer the discussion of the machine level representation of **Visitors** until section 5.6.

First, then, let us examine a trivial record structure of three fields:

```
    TYPE Tapir :
      ( N          limbCount
      , Distance   noseLength
      , HatSize    hatSize
      )
    ; ...
```

The **Visitor** for this type simply scans the three components sequentially:

```
    TYPE :< 𝒜 @ ⌜Tapir⌝ >: ()

  ; 𝒫 :< β_Tapir @ (𝒫 p0, 𝒜[⌜Tapir⌝] a, ℳ m) >:
      ( 𝒫 p1 : m[⌜N⌝][p0, [], m]                          .. limbCount
      ; 𝒫 p2 : m[⌜Distance⌝][p1, [], m]                  .. noseLength
      ; m[⌜HatSize⌝][p2, [], m]                              .. hatSize
      )
  ; ...
```

An equivalent rendition of β_{Tapir} in the perhaps more familiar imperative style would reüse a pointer variable:

```
    𝒫 :< β_Tapir @ (𝒫 p0, 𝒜[⌜Tapir⌝] a, ℳ m) >:
      ( (#𝒫) p : |p0                                   .. walking pointer
      ; p := m[⌜N⌝][+p, [], m]                              .. limbCount
      ; p := m[⌜Distance⌝][+p, [], m]                      .. noseLength
      ; m[⌜HatSize⌝][+p, [], m]                               .. hatSize
      )
  ; ...
```

Here we see that the argument structure for the non-parametric type Tapir is—consistently—() (an empty structure having exactly one member, namely []). The **Visitor** β_{Tapir} thus derives information only from its pointer parameter, p0, which it decomposes for the caller via the callback parameter m, invoked for each of the fields in turn. As it happens, the types of these fields are themselves all self-contained; thus in each case the second argument of the type-specific mode function (m[...]) is [], as it would be in any invocation of β_{Tapir} itself. The final result of the **Visitor** is exactly the value returned by m[⌜HatSize⌝], since the follow of the structure is the follow of the contents of the structure.[45] The auxiliary pointer constants p1 and p2 serve no purpose here but to preserve the original textual sequence; from a semantic viewpoint we might just as well have written the more heavily embedded

```
    𝒫 :< β_Tapir @ (𝒫 p0, 𝒜[⌜Tapir⌝] a, ℳ m) >:
      m[⌜HatSize⌝][m[⌜Distance⌝][m[⌜N⌝][p0, [], m], [], m], [], m]
  ; ...
```

and in fact we can expect a compiler to generate the same instruction sequence for each of the three descriptors, embedded, pure and imperative.

Consider now the type, or types, of general complex numbers (general in the sense that we do not assume, as do many languages—though Haskell got this right—that only approximate arithmetic is performed over the complex domain). We define a type constructor (or "type-valued function"), Complex:

```
    TYPE :< Complex @ TYPE T >: T re, T im
  ; ...
```

In this case, since the type is parameterised by the element type, T, the argument packet 𝒜[⌜Complex⌝] is no longer void. The argument that it takes, furthermore, is compound: since the argument type may be parameterised in its turn,[46] we make arrangements to pass the appropriate information—as specified by 𝒜[t]—through to the sub-**Visitor**:

```
    TYPE :< 𝒜 @ ⌜Complex⌝ >: (𝒯 t, 𝒜[t] a) t
  ; 𝒫 :< β_Complex @ 𝒫 p0, 𝒜[⌜Complex⌝] a, ℳ m >:
      𝒫 p1 : m[a t t][p0, a t a, m]                         .. re
    ; m[a t t][p1, a t a, m]                                .. im
  ; ...
```

[45]We are here in fact glossing over a slight subtlety, at least at the level of physical implementation: we are assuming the absence of fragmentation. In languages like C in which (visible) assistance is provided to the underlying hardware in forcing the alignment of a structure to compatibility with that of all its fields, with internal fragmentation otherwise avoided (something most homogeneously-heaped garbage collected systems don't bother trying to do), there is a need for address arithmetic at the end of any Visitor that copes with mixed field widths. Making this detail explicit would probably add nothing to the present discussion, however.

[46]As with fixed-point numbers, whose precision and scaling factor may perhaps be communicated.

Next we turn our attention to an instance in which *Exemplar* diverges from (any approximation of) standard notation in favour of one exhibiting the symmetry between code and data. Here is the definition of the Vector constructor:

```
    TYPE :< Vector @ (TYPE T, N n) >: for N i to n do T [i] end
  ; ...
```

The **Visitor** for this type must accept among its arguments not only the type of the elements of the array, but the array bound as well; for the arguments of β_{Vector} are, by construction, directly related to those of Vector itself.

```
    TYPE :< 𝒜 @ ⌜Vector⌝ >: (𝒯 t, 𝒜[t] a) t, N n
  ; 𝒫 :< β_Vector @ (𝒫 p0, 𝒜[⌜Vector⌝] a, ℳ m) >:
      with 𝒫 p from p0
        for N i to a n then p
      as m[a t t][p, a t a, m]                                  .. [i]
      end
  ; ...
```

As the body of the type has the form of a *for*-loop, so does that of the its **Visitor**; there is (as there was for our original Tapir example) a slight skew in surface form engendered by the fact that the **Visitor** returns a value. Here the pointer p is an induction variable that is "walked" up the array from p0 to (just after) the end, whence it is returned. The overall effect is just as for a fixed structure, except for potential variation in the parameter n; this is in exact accord with the mathematical identity between a vector of fixed dimension and the cartesian product over that number of copies of the individual coördinate structure.

Thus far every structure we have examined has been a product of one kind or another; these differ from sum types in that they exhibit no *internal* data dependency. The following structure, however:[47]

```
    TYPE InterestingThing :
      if {tapir, cuttlefish, stone} kind, kind
```

[47]Note that a field "kind" of enumerated type is declared locally and then immediately used within the *if*-part, the selector-expression of the case construction.

```
    is tapir      then Tapir tapirDetails
    is cuttlefish then Distance meanTentacleLength
    is stone      then N cucumberFrameMishapCount
    end
  ; ...
```

has three different forms depending on the choice of value for the kind field. The corresponding **Visitor**,

```
  TYPE :< 𝒜 @ ⌜InterestingThing⌝ >: ()
; 𝒫 :< β_InterestingThing @ (𝒫 p0, 𝒜[⌜InterestingThing⌝] a, ℳ m) >:
    if 𝒫 p1 : m[⌜{tapir, cuttlefish, stone}⌝][p0, [], m]
    ; δ_{tapir, cuttlefish, stone} [p0]
    is tapir      then m[⌜Tapir⌝][p1, [], m]                  .. tapirDetails
    is cuttlefish then m[⌜Distance⌝][p1, [], m]         .. mean TentacleLength
    is stone      then m[⌜N⌝][p1, [], m]         .. cucumberFrameMishapCount
    end
  ; ...
```

depends on exactly the same control structure to dispatch between its various cases. The presence, in the *if*-clause, of the function δ_T is a technical device providing access to the implementation's internal mapping between representations and and the values they represent; in essence it is the dereference operation associated with the type $\mathcal{P}$.[48]

When a parameterised type is instantiated as a new, named type,

```
  TYPE ComplexRational : Complex[Rational]
; ...
```

the **Visitor** must bind the same information—in the form, in this case, of the *name* (within $\mathcal{T}$) of the argument type, ⌜Rational⌝.

[48]It is expressed as a parametric function partly to make explicit the type it impresses on its result, and partly because its expected implementation does, indeed, involve a function call: in many of the applications to which Visitors may be put the normal hardware supported mechanism for dereference may be unavailable or inappropriate. Note that with the use of δ, despite the obvious operational interpretation of Visitors we we have been emphasising in this section of examples, these functions are in fact completely decoupled from the physical representation—they manipulate abstract images of representations, not the representations themselves. This can be put to practical use, as for example, when determining the length of an object given its Visitor, even in the absence of a physical instance.

```
  TYPE :< $\mathcal{A}$ @ ⌜ComplexRational⌝ >: ()
; $\mathcal{P}$ :< $\beta_{\text{ComplexRational}}$ @ ($\mathcal{P}$ p0, $\mathcal{A}$[⌜ComplexRational⌝] a, $\mathcal{M}$ m) >:
    m[⌜Complex⌝][p0, [⌜Rational⌝, []], m]
; ...
```

Taking this one step further, we exhibit a case where not only is the newly defined type itself numerically parameterised (and a defined type in its own right), but information is passed through it to its internal components.

```
  TYPE :< ComplexRationalVector @ N n >: Vector[Complex[Rational], n]
; ...
```

The **Visitor** for this type is the involved but by now completely predictable structure,

```
  TYPE :< $\mathcal{A}$ @ ⌜ComplexRationalVector⌝ >: N n
; $\mathcal{P}$ :< $\beta_{\text{ComplexRationalVector}}$ @ ($\mathcal{P}$ p0, $\mathcal{A}$[⌜ComplexRationalVector⌝] a, $\mathcal{M}$ m) >:
    m[⌜Vector⌝][p0, [⌜Complex⌝, [⌜Rational⌝, []], a n], m]
; ...
```

Finally, let us examine a common structure whose inductive pattern stretches the bounds of what is generally considered (strongly) typable:

```
  TYPE CChar : Character{nil}
; TYPE CString : *while* CChar c[49], c $\neq$ nil *end*
; ...
```

Here we see a simple rendition of the programming language C's notion of a string. Its elements are more-or-less characters, though we have provided a distinct guard value nil which is *not* itself a character, a nice distinction which C itself does not make. A CString is then a *while*-loop over this extended alphabet, consisting of a sequence of characters in the basic character set terminated by the distinguished element. The corresponding **Visitor** is, of course, also a loop:

```
  TYPE :< $\mathcal{A}$ @ ⌜CString⌝ >: ()
; $\mathcal{P}$ :< $\beta_{\text{CString}}$ @ ($\mathcal{P}$ p0, $\mathcal{A}$[⌜CString⌝] a, $\mathcal{M}$ m) >:
    *with* $\mathcal{P}$ p *from* p0
```

[49]Here we do not take care to maintain the random accessibility of string elements. We will exhibit a more detailed representation of C strings in section 4.6.4.

```
        while $\mathcal{P}$ p1 : m[⌜CChar⌝][p, [], m]; $\delta_{\text{CChar}}$[p] $\neq$ nil then p1
        as p1
        end
    ; ...
```

though as with the Vector example above, our translation has introduced auxiliary induction clauses to keep a "thumb" into the structure. As in the InterestingThing example we use the δ operator to examine the control expression, though in this case it iterates over every character in the string. Finally we reach the guard value; the position *after* the guard is the position after the CString as a whole.

For this, our final example, we once again provide a rendition in imperative terms, which may be a little closer to what we would expect a compiler to generate as output:

```
      $\mathcal{P}$ :< $\beta_{\text{CString}}$ @ ($\mathcal{P}$ p0, $\mathcal{A}$[⌜CString⌝] a, $\mathcal{M}$ m) >:
        (#$\mathcal{P}$) p : |p0
      ; while $\mathcal{P}$ p1 : +p; p := m[⌜CChar⌝][+p, [], m]; $\delta_{\text{CChar}}$,p1] $\neq$ nil
        then +p
        end
    ; ...
```

From the examples above, it should be clear that although **Visitors** are higher-order polymorphic functions, they are have a very restricted form. In general we find that efficient machine implementations exist for the **Visitors** of all the "usual" types, a fact that we will ultimately investigate, and exploit, in section 5.6.

3.4 Levels of Visitation

In the preceding section we assumed that whenever a field was declared in a structure—at every leaf—the **Visitor** of that structure would contain a call to the Mode entry (m in $\mathcal{M}$) for that leaf. In fact, however, many applications of **Visitors**—in particular, garbage collection, for which they were originally devised—are interested not in the full "parse" of a data structure as it was declared by the programmer, but only in the sequence of atomic constituents that represent it. This situation is strictly parallel to that of code generation: in all but a very few circumstances (interactive debugging being paradigmatic) the programmer is not concerned with whether the original structure of the programme is preserved throughout compilation, so long as the overall effect is as if it had.

Efficiency would be enhanced, and (in the cases where compositional structure is not of interest) functionality not compromised, then, if the familiar "inlining" optimisation from code generation were applied to **Visitors**: wherever one structure is bodily embedded in another, we can replace the call to the sub**Visitor** with the text of the called routine. Similarly, we replace remaining $\mathcal{M}$-calls for user-defined types T—ones that are represented indirectly, via pointers—with calls, not to the mode of $\ulcorner\%\mathsf{T}\urcorner$, but to $\mathsf{m}[\ulcorner\mathcal{P}\urcorner]$; $\mathcal{A}[\ulcorner\mathcal{P}\urcorner]$ (since pointers must point *to* something) is in fact

$$\forall \mathsf{T}.(\mathcal{P} \times \mathcal{A}[\ulcorner\mathsf{T}\urcorner] \times \mathcal{M} \rightarrow \mathcal{P}) \times \mathcal{A}[\ulcorner\mathsf{T}\urcorner]$$

—the type of **Visitors** in conjunction with their particular semantically determined argument structures—and we pass $[\beta_{\mathsf{T}}, \mathsf{a}]$ as its semantic argument.

The effect of this transformation is that the domain of m is now just the (presumably finite) set of *primitive* types and type constructors, paving the way for a very efficient implementation indeed.

Even in the case where $\mathcal{T}$, the domain of $\mathcal{M}$, ranges over all the possible types from a source programme, however, it is still a *finite* enumeration of type names, since in the case that a type is defined schematically it encodes only the head term (which must presumably have some textual correlate); any arguments (or contextual information) are, or may be, bound up in the argument structure.

3.4.1 Variable visitation levels

In fact, the selection of a level of "primitive objects" within a programming language is entirely arbitrary. Despite that an implementation may provide some object pre-built and leave the making of another to the programmer's care, or that the source syntax may have specialised notation for one facility and leave another to fall back on more general resources of the language, there is no requirement of the language processor to treat the one in a manner different from the other—just so long as *some* implementation is provided for every part of the language.[50] This is no less true for types than it is for functions: the type Boolean, for instance, is as easily implemented as the "defined" type {false, true} as it is built in—though there is an important distinction of a different nature between the official core of the language which the user's programmes may rely upon, and those facilities which are built on top of it.

That being the case, we can see that the level of abstraction at which visitation is performed is potentially a parameter, under the control of the user of the mechanism.

An implementation might capitalise on this possibility by providing that the user can define the desired level of detail in visitation, in the following manner: the user is provided with a function m in $\mathcal{M}$, which maps each type name into its **Visitor** (and does nothing else). The overall effect of invoking the **Visitor** of some data structure with this $\mathcal{M}$-function will be to visit each of its fields, but to perform no action in the process. If, however, we now allow the programmer to perform pointwise updates on the function, changing its behaviour at any type,[51] it will be possible to induce the **Visitor** to perform useful work. In particular, if the set of types covered includes *at*

[50]It is in fact commonplace that certain "primitive" operators of a language be implemented as subroutines—especially on hardware with a limited instruction set—but the reverse case, while possible, is less common: this would correspond to a feature *not* in the language as set forth in the language definition, which nonetheless has a specialised translation. A nice example might be of a compiler that recognises certain explicitly coded loops as implementing hardware-provided string instructions, and then uses the latter in code generation. Such recognition of idioms is probably most common in the world of APL compilation.

[51]Any type, that is, that is nameable by the programmer under the usual constraints.

least all of those that form part of the language core,[52] all reachable parts of a data structure can be processed in a user-specified way.

Types that are not named—whether because they are not of interest to the programmer, or because they are not nameable in the scope of construction of the programmer's $\mathcal{M}$-function, are traversed silently, decomposed if they are compound, or passed over if they are atomic. The programmer thus need not be concerned with abstractions built either over or under the structures being considered.

3.4.2 Opacity under visitation

An important consideration in the deployment of **Visitors** is the impact they have on modularity. Clearly, giving the application programmer free access to the facility impairs modular opacity, strictly interpreted, in two ways. First, since it analyses structures of arbitrary type without imposing upon the caller to decompose them, it can be employed to determine the manner in which an object has been (requested to be) implemented. Second, just as it gives access to remote modules in the user's code, so it provides a viewport into the language implementation itself: given an object it can be used to determine how that object was *in fact* represented to the machine.

In effect these are countervailing difficulties: in particular, **Visitors** certainly do not in general provide a *reliable* view of type implementations chosen in remote scopes, since, as we, have seen,[53] the ministrations of the language processor may have resulted in quite arbitrary mutation of these implementations during compilation and processing. The situation is thus quite similar to that of languages which fail to make promises regarding evaluation order of arguments in calls: it is true that this

[52]If the language provides no mechanism for *deleting* objects from visibility, we can even say: all of those that form part of the language core *as understood from the perspective of the application*. [pen83] argues that facilities for such deletion are in fact desirable when constructing a specialised sublanguage tailored for application to a specific project; but in that case it is presumably incumbent upon the creator of the sublanguage to provide the appropriately specialised version of the **Visitor** (if visitation is required at all).

[53]Unless steps have been taken to the contrary, as when **Visitors** are used to provide an external representation of an object to a remote agent with which a detailed object representation contract may not be outstanding.

situation can be exploited by the programmer to gather information both about the behaviour of supposedly opaque external modules, and about the language's compilation strategy (which, we would argue, amounts to more or less the same thing, if the programmer of the caller is "playing the modularity game"); but ultimately, if the programmer wishes to play Mastermind with the implementation, there are always virtual memory monitoring facilities—providing military security levels between modules of the same programme is not the object of the present undertaking. On the other hand, it is certainly the case that no programmer should be tempted to use the present facilities to violate *conceptual* modularity constraints, since, although they *do* cross module boundaries, they do so in an unspecified and unspecifiable manner. In particular, no violation of modularity is *required* to employ visitation—as is not the case for the obvious alternatives.

Notably, the **Visitor** mechanism (at least if its interface be implemented as suggested in the preceding section) *does* take care always to present an understanding of the world couched in terms of the locally visible types and no others. Hopefully no more than a modicum of programmerly wisdom is therefore necessary in its employment.

3.5 Structures as Suspensions[54]

In this section we present an alternate understanding of data types, one which motivates the more general application of **Visitors** that is contemplated in chapter 6.

It is traditional to view data types algebraically under their opaque aspect, and structurally where they are transparent. Inspired by the λ-calculus, however, let us consider an algebraic view of transparent structures.

First, let us remark on two rather well-known facts: the first is that λ-calculus permits the representation of arbitrary computational objects, including data structures,

[54]This brief section is based on material originally presented by the author at the Montreal meeting of IFIP Working Group 2.1.

```
.. Here we give an example of Bag[T], showing how constrained free
.. constructors work. Note that the injection axioms fall out from
.. the free construction method.

.. "∀" is here an infix operator taking a pattern on
.. the left and, on the right, a Boolean expression over the
.. variables it binds.

   TYPE :< Bag @ TYPE T >: {0, bag[T], (⊎)[TYPE B, B]}
     | (B = Bag[T])                                   .. Utility definition
     | (B b ∀ b = 0 ⊎ b)                                       .. Identity
     | ((B b, B c) ∀ (b ⊎ c) = (c ⊎ b))                 .. Commutativity
     | (  (B b, B c, B d)
           ∀ ((b ⊎ c) ⊎ d) = (b ⊎ (c ⊎ d)))              .. Associativity
; ...
```

Figure 3.4: An algebraically defined type

as λ-terms (which are in turn representable as closures in conventional functional languages[55]); and the second, that arbitrary algebraic structures can be factored into two components: a (possibly many-sorted) *free* algebra of the function symbols,[56] and a set of constraining algebraic *identities*. See figure 3.4 for an example in (a somewhat extended) *Exemplar*.

Normally when constructing types (or algebraically specified objects in general), the constraints placed on the algebra are understood in the sense of *quotients:* they reduce the size (and increase the information content of the description) of the underlying free algebra until only the desired structural relationships between the symbols remain. Since, however (at least on the understanding of the identity of indiscernibles), objects may be identified with their input-output behaviour, it is equally valid to impose the algebraic identities as *static type constraints* on the functions that

[55]Be it noted that "classical" LISP implementations do not fall in this category, since they lack (context-independent) first class functions.

[56]And, *a fortiori*—their being equivalently but functions of no arguments—constant symbols.

manipulate them.[57]

In particular, if we represent an instance of an algebraically specified data type by the open term in the appropriate free algebra over the constructors of the type, λ-abstracted on the entire set of such constructors (in some canonical order), we obtain an implementation of the object that is mathematically adequate precisely when the particular eventual arguments of the term are jointly restricted to obeying the defining algebraic identities of the type as a whole, each in its corresponding rôle.[58]

This formulation works equally well in the case of **carrier** types, in which not all of the sorts of the algebra are in fact closed over in the type definition, and of types derivative of other types: the only additional restriction now being, as one might expect, that the eventual arguments provide a consistent assignment of types to the sortal constants and free sort variables of the original algebra.

The main point of difference between this style of "abstract implementation" and the more usual technique of strong involvement of external facilities in the algebraic specification of the type itself—the latter a more operational approach to the input-output behaviour of the type, if you will—is that it provides a clear formation/introduction/elimination format[59] uniform across all types, whether they be terminal, carrier types, or even functional in themselves. Of especial relevance to us is that this outlook provides for the **unitary elimination of elements:** no matter that the object be large and complex, the "natural" method of eliminating it from (consuming it in) a computation is monolithic. To take a concrete example, in this style the formally obvious method of using a list value would not be by decomposition through null, first and rest operations, but by direct elimination with reduce (fold... in Haskell terms). See figure 3.5.

[57]The notion of equalities as types is of course not new; see [ML84], and [Dev84, CAB+86] for applications to programming languages

[58]The reader may observe that this is in fact a reasonably accurate, if abstract, description of the contents of the heap in a normal-reduction-order implementation of a functional programming language which employs a tagged data representation despite a strongly typed source notation.

[59]The reader unfamiliar with this notion is referred ahead to section 4 2.1.

```
.. Now for the List[T] example, showing the familiar operations
.. as defined:

  TYPE :< List @ TYPET >: (nil, cons[T, List[T]])
; .. No restrictions since we used the inherently asymmetric definition.

  Boolean :< null @ List[TYPE T] l >: l[true, List[T] _ >: false]
;
  (TYPE T){firstOfNil} :< first @ List[T] l >:
    l[firstOfNil, (T t, List[T] _) >: t]
  ;
  List[(TYPE T){restOfNil}] :< rest @ List[T] l >:
    if l [  [nil, restOfNil]
         , (T t, (List[T] l, List[T]{restOfNil}_)) >: [cons[t, l], l]
         ]
    is [_, x] then x
    end
;
  TYPE U :< reduce @ List[TYPE T] l, (T → U → U) f, U u >: l[u, f]
; ...
```

Figure 3.5: Direct elimination in an algebraic type

Considering λ-abstraction itself, as applied in the creation of these representations of the elements of abstract data types, we see that it is in some sense the *most general interpreter* for the datum: it is a function which is parameterisable to provide all and (in consideration of the type restrictions we impose) only its legal interpretations.

The relevance of this representation to the present thesis, of course, is that this most-general-interpreter representation of a datum is precisely the function that is computed by a recursive nest of **Visitors**, where the interpretation parameters are passed in as user-specified components of the range of the $\mathcal{M}$-function. The division between the algebraically specified type and the type symbols free in its signature has been dissolved (without loss of generality) into the precise pattern of parameterisation specified by the programmer.

The most-general-interpreter formalism—possibly generalised to the **Visitor**'s more global view of an application's entire type system—is in fact technically preferable to the visitation mechanism as presented (though far less straightforwardly implementable, since the checking of algebraic identities cannot be automated without full-scale proof generation techniques), since the algebraic identities ensure *complete* modular opacity, apparently without sacrifice of functionality; seemingly, however, it can only find application in a restricted family of languages, those with equational types, while **Visitors** require only that their host language be strongly typed, an issue we shall explore in the next chapter.

Chapter 4

Type Systems

In the preceding chapter we explored the nature of, and a number of issues concerning, data structures; here we discuss data *types*. The distinction is perhaps not a standard one, but it is important to the present discussion, for the mechanism of the **Visitor** manipulates structures but is justified by types. For our purposes, we take **type** as an *intensional* notion: it refers to the intended interpretation of an expression or a concrete datum; whilst **structure** is *extensional,* and concerns the actual representation of the datum in terms of other types or the underlying facilities of the machine. A type will, however, often have a **natural structure** to which it corresponds by parallelism of construction)

The reader will note that there is a crossing of levels apparent in this description: we allow "extensional" structures to be built out of "intensional" types. The reason for this seeming anomaly is the crucial rôle that abstraction plays in programming: for representational opacity—the local inaccessibility of "implementation"—is no other than the local coïncidence of type and structure, of extensional and intensional interpretation, in the form of a nonce-atomic datum. The next chapter explores the concrete aspects of this relationship between types and data structures; for now let us content ourselves with the observation that the ultimate implementation, the compiler and the object code that it emits, *always* has access to the lowest level representation of a datum, while a strongly typed language may exclude the *programmer* from this level altogether.[1] The purpose of the Visitors of the preceding chapter is to permit controlled interaction between these levels, at a finer grain than has hitherto been possible.

4.1 Philosophy of Type

In this thesis we take the view that strong typing is necessary to the implementation of several highly desirable language features, and sufficient to nearly all practical programming tasks that are not already overconstrained by back-compatibility issues (though see section 5.7.3 for an approach to even this issue). Here we attempt to establish the conceptual credibility of this stance, in terms of the nature of the programmer's task.

We assume for the moment that computer programmes are ultimately used to model "physical" systems, broadly enough construed to embrace, for instance, psychology. Any such model will be based on inputs derived from observations of the system; and any observation derives from an observer, whether intelligent or mechanical. The nature of the observer manifestly induces restrictions on the value measured: there is always some **domain** from which reported values are drawn.[2,3]

If a model is of any interest, furthermore, it surely predicts values of variables or parameters of the modeled system that are themselves—directly or indirectly,

[1]This applies especially to functions, whose empirical extension is, in conventional language implementations, only tenuously related to their source form.

[2]Our notion of type, then, is a direct generalisation of the notion of "dimension" from physics; a data structure corresponds to a particular measurement, complete with its units. Failures of the meter correspond to "exception values" and may themselves be folded into the measurement domain—to the extent that the failure mode is itself subject to measurement. While we shall not be making further use of this analogy, it is one that the author has found very useful in pinning down relationships between types and structures.

[3]It might be argued that in the case of a *human* observer the domain of measurement is not predetermined, humans being endowed with "free will"; but on reflection this will be seen to be fallacious: while—for sound mathematical reasons—it is not easy, as a human, to predict the full range of possible human responses to a situation, nonetheless both the input and output bandwidth of the human organism are bounded—all the more so in the case of linguistically encoded data—and thus *a fortiori* our argument applies to the human observer.

timely or otherwise, in principle or in practise—measurable[4] (even of astrology is this purported). In order to be intelligible, these variables must, also, be drawn from *a priori* known domains; for otherwise they will fail of interpretability. Even (once again) in the case of linguistically coded data one must know to expect language, to anticipate its modality (spoken, written, modulated on an electromagnetic carrier, ...), something about the range of languages to be expected, and—as it transpires—a *great* deal of pre-established context.

Clearly, the input and output domains may be interpreted as types (for they have representation and an interpretation as (a report on) a measurement of some kind), and the overall computation is globally typèdness-preserving. Further, we may observe that even *internal* aspects of a programmer-constructed model derive, ultimately, from observation, and are passed by communication; the same argument of *a priori* typèdness would seem to apply to them also, *mutatis mutandis*.

Under a fairly weak entropic assumption, that type information, once lost, cannot be recovered[5], we can infer that *every* value derived for the input (in either code or data) and destined for output is typed—though whether the typing be explicit in the notation or implicit in the behaviour of the model we cannot say.

Thus, while we have not established that all computation is internally typed (a position that we *will* later argue on the basis of hardware engineering techniques[6]), we are now—modulo our very concrete ontology—in a position to suggest that meaningful computations may be typed without loss of generality. Much of this thesis attempts to establish that the benefits of maintaining type information in detail throughout a computation are great, and the convenience to be had from discarding

[4]In the determination of the values of "hidden variables" of a system, the scientific method, at least, would insist on there being some other method of checking the result—for otherwise the value will be vacuous, devoid of scientific meaning.

[5]An assumption that is not at odds with the possibility of self-organising systems: for to serve as a counter-example, such a system would have to recover the *original* input information, rather than constructing information "of its own": exhibiting not attraction but synchronicity.

[6]The theoretical basis of this argument has, however, only the same status as the observation that the output of software systems is limited by resource bounds to regularity.

it is less than nothing.

MOMMI: **Meaning Out Means Meaning In:** In looking at the matter from the other direction, it seems that computer science is blessed with a singularly apt acronymic aphorism, GIGO, *viz.:* Garbage In—Garbage Out. If we take this—uncontroversially, I trust—as both a logical implication and a practical truth, we are justified in asserting also its contrapositive: wherever meaningful output is required, meaningful input must be provided. In sum: Meaning Out Means Meaning In.[7]

In the context of the present discussion, this would appear to imply that there is no hope of constructing a value whose interpretation is known—which, in short, is typed—from input data that do not also share this property.

We are left with the question of whether we can build type systems that are *practical* for the kinds of engineering tasks that arise in the real world. In the following we establish that they can cover almost everything that is handled by languages with weaker (but more liberal) type models, and with greater convenience; this, at least, is a start.

4.2 Definitions and Concepts

In this section our objective is first to lay out the particular style of type formalism on which we base our analysis of programming languages, and then to contrast it with a number of similar concepts (sets, classes, and underlying implementations) with which we believe they should not be confused—though in each case there is excellent historical precedent for such confusion.

[7]Technically speaking, "NGONGI" would appear to be the acronymic contrapositive of GIGO, but it seems somehow too exotic for such a motherhood argument.

4.2.1 Type formation

We follow what has become standard practise in characterising a type, or family of types, in terms of its *formation, introduction* and *elimination* rules. The **formation** rules of a type describe the syntactic form of the type's name (nontrivial in the case of a member of a parameterised family of types) and the conditions for its wellformèdness. Similarly, the **introduction** rules give the forms and conditions for the construction of *members* of the type (including the types of any arguments they may take). The introduction rules are balanced by **elimination** rules, which give forms for the extraction of the information bound into the type's members, and the types of the results. The asymmetry here—that values are introduced and eliminated, while types are formed but not "de-formed"—is accounted for by the facts (1) that types[8] shaie with functions the property that their expressive power would be restricted by a stipulation of (general) decomposability; and (2) that the construction of a type is "justified" by the subsequent introduction of its members: as with a function, it is the *use* of a type that is its *raison d'être*.

An example is provided by the simple type N of natural numbers: its formation rule (assuming, plausibly enough in a programming context, that it is a primitive of the system) is a simple stipulation: N exists, and is a type. It has two introduction rules, one providing that 0 is a natural number, and one that, on condition that n is in N, then so is the successor of n. Finally it will provide one or more elimination rules (subtraction and an equality test would suffice in the presence of recursion, and be conventional from a programming language viewpoint; some form of definite iteration construct would be equally suitable[9]).

Cartesian product is only slightly more involved: the product, $\times$, of T and U is defined whenever T and U are both types; there is a single introduction rule stating

[8]At least in their more sophisticated forms.
[9]ITT [ML84] provides what amounts to (a "lazified" form of) the latter.

that for any t in T and u in U, $\langle t, u \rangle$ is in $T \times U$; and the elimination rules might well provide a pair of projection functions from the product back to the original argument types (or some kind of splitting function as in ITT, or as would be natural in a language possessed, like *Exemplar*, of patterns).

4.2.2 Type restriction

To this point, intension and extension coïncide. To achieve this effect we have cheated somewhat in having eliminators extract "the" information from a member of a type; ITT, for instance, would have us specify in each case a series of **equality** rules specifying which combinations (or reïterations) of operations yield results equal within the type, or indeed between types. In particular this is necessary for functions, where textual equality is certainly not the appropriate general method of determining equality between results—there must be, somewhere, an evaluation mechanism.[10]

Such equality rules (if they go beyond specifying the usual semantics of free algebras) have the same effect as the identities of algebraic type systems. Since they restrict either the possible values of a type or (if we use the dual interpretation presented in section 3.5) their permissible *interpretations,* they have the effect both of moving in the direction of intension—thereby distinguishing the type from its underlying natural structure—and of inducing a subtype relation between types with the same structural properties but different degrees of restriction: a more restricted type is a subtype of a less restricted type (symbolically, $T \sqsubseteq U$ when T is a subtype of U). Since the natural structures of these types are identical, a straightforward implementation need not even concern itself with the induced distinction once type correctness is satisfied.

A less superficial form of subtype relation may be produced (at the type system's—and possibly the programmer's—option) when one type is *embedded* in another, as,

[10]The well-informed reader will have noticed that the implication that the reduction mechanism is localised specifically in the equality rules of function types is not technically accurate, but this would not seem to be the place to explore the issue in detail.

for instance, in a Cartesian product. Since the addition of fields (with their own complement of validity constraints) actually[11] *restricts* the values available for earlier fields, the extended type *as projected back to its "parent"* is a subtype of its original.[12] Implementation is still straightforward (though here we delve further into the realm of real machines), at least for the case of the product, since the most direct implementation of the *natural* structure of a type will result in component structures being contiguous; thus simple pointer arithmetic, or often just use of a pointer, can convert from the subtype to the supertype. 194z Actually, an embedded subtype need not be among the "component" types of the supertype; it would suffice if the subtype could be reconstituted from values projected from the supertype. This observation underlies the notion of multiple inheritance,[13] and, surprisingly enough, satisfactorily efficient implementations based on pointer manipulation are also known for at least the restricted product instances of this case [PW90].

Taking our development one step further, one can imagine a "subtype" being declared by the provision of an arbitrary function from the subtype to the supertype; in the context of a programming language, such a "declared" subtyping is nothing other than a user-defined coërcion. While there may be practical arguments against introducing such a facility into a practical programming language,[14] coërcion is certainly something that language designers have employed freely in the past on their own behalf, and here we see its theoretical semantic grounding.[15]

[11]By the magic of the Galois relation.

[12]The reader is here cautioned that while in the case of type restriction, subtyping corresponded to subsetting, here the subtype is potentially of greater cardinality than the supertype. Furthermore it is in general impossible to convert members of the supertype to the subtype, since values would have to be "invented" for the newly added fields. The direction of the subtyping relation is in fact determined by the existence of natural conversions between types, which is itself a function of the formal descriptions of the types, a property not directly related to the set-theoretic properties of models of the type. This is as it should be given that types are not, for us, an extensional notion.

[13]At least, in its type-theoretic version.

[14]Notably ambiguity of conversion path, ambiguity of *intended* type for an expression, and ambiguity of the resulting overall notation!

[15]In order for the term "subtype" to remain technically justified we also have to ensure that the overall subtype relation remains a partial order and that coërcions never fail. In the previous, more syntactic, subtype constructions this followed by construction (as can be seen by contemplation of

Now our journey from extension to intension is complete: types that may be arbitrarily rerepresented in passing from subtype to supertype are fully abstracted from their concrete representations. In fact, of course, types are *always* distinct from their structures; here we have just surpassed the limit at which practical compiler stupidity can no longer suffice to make them coïncide.[16]

4.2.3 Strong typing

In the formalism we have just presented, **strong typing** is preserved by construction: the well-formèdness of types follows from the restrictions that the formation rules impose, while the typèdness of values follows from the inductive composition of expressions. Each basic operator in the expression tree interlocks with the result types of its arguments to form a well-typed structure overall. Functions are no different from any other objects: the operation of function application takes a function argument and an argument argument of the function's argument type, and returns a

the geometry of the parallel *structures*), but in the case of arbitrary coërcions it cannot be checked locally at all.

[16]The reader may be wondering why very general systems of subtyping are not widely implemented in programming languages. There are a number of reasons for this: compiler complexity is one, the potential for ambiguity in the source code is another, runtime cost is a third. In this last case, while less general conversions on statically typed systems may have constant cost through clever manipulation of representation, dynamically typed systems and cases where the layouts of the two related types are incompatible must pay at least the cost of copying the object, and thus are linear in object size.

The single most important reason why we do not see more general type hierarchies, though, is the difficulty of establishing how they behave in the presence of defined types. Here *Exemplar* offers the extreme case, for in general extensional structure equality in *Exemplar* is equivalent to function equality, and is thus not computable.

In the case of ground types, objects of "type" TYPE with arguments fixed* at the moment of analysis, this is not a problem: the types can be unraveled and compared *extensionally* (extensionally within TYPE, that is).

Furthermore, it *is* possible to demonstrate straightforwardly that if T $\sqsubseteq$ U then Vector n T $\sqsubseteq$ Vector n U (since static inspection of Vector reveals that its type argument is used only positively), but (at least without detailed knowledge of the operation of the *for*-lop in terms of which Vector is, as it happens, ultimately defined) it is *not* possible to determine that if $n < m$ then Vector n T $\sqsubseteq$ Vector m T; and while this is not perhaps obvious, it is not even possible (without special knowledge about N) to detect when Vector n T = Vector m T, since we do not (for obvious reasons, since functions have types) require all types to support runtime equality tests. Note however that, one way or the other, all such computations rest on the local decomposability of the type.

*But note that this includes having knowledge of the *values* of any fields on which dependent types (see below) depend.

result of the function's result type.

If we introduce a subtyping relation as described in the previous section, slightly more care is required: for the formal parameters of the various type constructors must be classified into those that are *positive positions* and those that are *negative positions* (a position that is both positive and negative is **exact**; positions that are neither can in principle exist, and may ultimately describe a *truly untyped* expression that can be omitted from the computation on grounds of irrelevance). In essence, **positive positions** are those that correspond, in the introduction rules, to *value expressions,* while **negative positions** correspond to types used for *pattern expressions,* such as the formal parameters of functions.

The reason for this distinction is as follows: it is clear when constructing a value from other values, that providing an argument that is *more* constrained or *more* detailed than is formally required—in short, a member of a subtype—will do no harm (our assumption that we have an official subtyping mechanism assures us that the implementation is willing to supply any requisite "fixup" on the underlying structures, of course). On the other hand, since a pattern that is constructed from a type will (as just noted, effectively) tolerate a member of a *sub*type, it itself may consistently accept a *super*type of that formally required. (In exact positions, of course, type equivalence must be maintained.) In sum: in positive positions, subtypes are acceptable; while supertypes are acceptable in negative ones.[2]

[2]A curious manifestation of this same duality is what we refer to as the note phenomenon (after a hypothetical programming language construct of that name. While many programming languages provide one or both of an assert-statement, which records a predicate that must hold at some point in the programme and instructs the implementation to abort execution with a diagnostic message if it fails, and some form of "constraint pragma" facility for making observations of use to the optimiser (such as the necessary inequality of two pointers or array indices, for instance), it is not generally appreciated that these constructs have the *same* semantics and could straightforwardly be combined. Each of them records a predicate that *must* be true for the programme to be correct; the only difference is that while assertion requests abortion if the condition is not satisfied on the left, constraint empowers the implementation to assume its truth on the right. One can easily imagine an implementation, then, that uses only a single source construct—the note—compiling a check whenever "safety" optimisations are requested, and subsequently assuming that the check succeeded (or would have succeeded) whenever "speed" is wanted.

4.2.4 Contrasting types with sets

Having set out our approach to typing, we now contrast types (in our sense) with a number of distinct but related concepts with which we feel they have been confused in the past. The first of these is that of the *set*.

One of the familiar explanations of the computational concept of type is that a type actually corresponds to the mathematical notion of a set, taken in conjunction with a collection of operations on that set. A *prima facie* difficulty with this approach (at least in its usual presentation) is that it can capture only *internal* relations on the type, and thus fails in describing the equally important—and often richer—relationships *between* types.

Worse, this characterisation can be quite misleading, since it encourages one to focus intellectually on an *extensional* notion of the type, namely the membership of the set. On the one hand, we wish to view types as *intensional* structures, having to do with the intended meanings of data; and on the other, the actual sets we are typically invited to contemplate are abstract mathematical objects and not merely an extension but, if you will, the *wrong* extension: they may *model* the physical implementation of a datum in a practical machine, but they are not that implementation; nor are they the best way of thinking about that implementation if that is indeed our concern. The best we can say in terms of set theory (in order to capture the desired intensionality) is that we are interested in the elements of a set *under some interpretation function,* which is also a part of the specification of the type.

The particular relevance of this distinction between types and sets is that in standard set theory, objects possess a life of their own: they come possessed of a "free" equality relation,[3] can be placed in sets arbitrarily and without restriction, and so forth. In particular, we can know that an object is equal to itself and to nothing else without knowing of any independently motivated set to which it belongs. In contrast,

[3]No pun intended.

whether two objects receive the same interpretation (are equal, *i.e.*) or whether an object receives *any* interpretation, cannot be known without reference to some type. This means that types must be known *a priori* where values are to be manipulated;[4] in fact, the failure of this requirement is a fairly familiar empirical phenomenon in weakly typed languages where it often surfaces as such a beast as the "wild pointer bug."

Aside from the MOMMI argument made above, that nothing of practical value is lost in requiring that we compute only with interpretable data, there is a further argument that in fact types as we present them are more appropriate to the explication of programming languages than are sets: that the notion of interpretability that underlies our type system can be restricted further to *practically computable*[5] interpretations (an idea that we will in fact capitalise upon below) of data. This is a considerable improvement over a description in terms that would seem to lack such computational relevance.

4.2.5 Contrasting types with classes

The object oriented programming community[6] is presently in the throes of a debate over the precise nature of the relationship between the "class" and the "type." The emerging consensus seems to be that, at least in the object-oriented context, the word "type" ought best be used for abstract protocols that various classes might choose to respect: that certain messages are recognised by the instances of the class (and presumably that these result in uniform intended behaviours, though this cannot be mechanically controlled).[7]

[4]Modulo, at least, information actually to be used in the object's interpretation—a subtle equivocation that is of relevance in nonstrict evaluation paradigms.

[5]The precise sense of practicality needed here actually depends on the style of the programming language itself: certain languages (such as C and Pascal) restrict basic facilities to constant-cost operations, while others, such as COMMON LISP, provide much more expensive operations in their basic repertoire; the same kind of style dichotomy could manifest at this level.

[6]OOPCOM ☺

[7]Confusingly, to say the least, the designers of the functional language Haskell have chosen to use the words "type" and "class" with precisely the reverse interpretations: in Haskell, a *class* is a

This notion of type corresponds quite closely with ours (except, perhaps, in the weakness of its typical concrete manifestation in the language), having an intensional flavour and being the guardian of semantic coherence in the language. The *class,* however, is for us no more than a particular collection of values (the fact that a class—and its values—can pertain to more than one type is simply indicative of a certain kind of richness in the type system[8]), ones sharing a value under the classOf function.

Thus the difference between classes and types (in our sense) is that class is a property of a *value,* (and hence, at least classically, dynamic), while type is a property of an *expression* (and thus static). This has practical implications in two distinct regards. From the perspective of pure performance, the dynamic association between an expression and its interpretation must (at least superficially) bear a heavy cost: although implementations are improving steadily, it is still the case that a significant proportion of the execution cost of an object-oriented programme is consumed by **method lookup**,[9] the operation of locating the definition associated by a class with some name; and the fact that objects must typically record their own classes imposes a burden of space on every instance.

From a semantic standpoint, since the class of an object is recorded *in* the value, it is inherently tied to object extension: there is no possibility for the interpretation of an object to vary by declared intension.[10] The expressive power of the two systems is (on the assumption, at any rate, that there are near-first-class types in the language, so that values of parametric type can be manipulated) the same; but from an engineering perspective (if not perhaps a prototyping perspective), a strong type system seems to be the better investment where a choice must be made.

published interface protocol to which a *type* may elect to conform!

[8]A richness that is, perhaps, paid for in that all objects will typically share a *nontrivial* type.

[9]And, the closer we get to the "frames" outlook, slot lookup.

[10]This is insurmountable only in the case of mutable objects—for otherwise it can be solved, at some cost in time, via transfer functions—but the object oriented paradigm is (thus far) strongly imperative in outlook and biased towards mutability.

4.2.6 Contrasting types with implementations

Although we present an approach stressing the inductive construction of types out of pre-existing types, it is important not to confuse the type with its implementation—with the structure (or structures) underlying the type. In this thesis, at least, we rely on the fact that the implementation *exists* and that it must possess certain properties and relations to its preïmage in virtue of its (presumed) utility; but we do not insist that the mapping between them be transparent or even consistent between different instances of a given type.

Furthermore, it is clear that there is (for most constructs at least) a "natural" implementation corresponding in the data domain to the "operational" interpretation of a code structure, and this serves as a meaningful baseline for description. Most extant language implementations in fact do little more than provide a transparent implementation of this baseline[11]—in fact, typical compilers seem to be so utterly doctrinaire about the transparent implementation of defined types that they do not even consider them or their components to be candidates for registering, though the relevant safety criteria are essentially identical to those for scalars in the same context[12].

In essence, the situation is exactly as it is with the executable portion of a computer programme: the source programme *can* be interpreted as a description of how the computer should proceed at each step (in the case of data types, while "laying out the successive fields" in memory), but since it is the *interpretation* of the type that is of interest to its clientele (precisely as it is the input-output behaviour of the overall

[11]SETL [SDDS86] is the usually cited example of a language providing nontrivial compilation of data types, but in practise this seems to amount to little more than a library of alternative implementations for some of its more expensive structures it provides—more akin, perhaps, to the provision of several different sort routines in a standard subroutine library than to the kind of transformation carried out on the executable component of a programme by a modern compiler. In the more recent SETL2 [Sny90] the data representation sublanguage has been deleted.

[12]As we shall see below, a stack frame is for our purposes best considered as a structure anyway, with environment captured in a closure being a conventional reference to it; so in fact some *ad-hoc* decomposition of structures is arguably already being performed in the most naïve of compilers.

programme that is of interest to the programmer), the compiler is free to provide any implementation which preserves that interface intact (which is no more than to reïterate our point that type is intensional, of course!).

There are any number of ways in which an implementation might diverge from a naïve reading of the type declaration; here, by way of illustration, we consider four of them. It should be noted that each of these can be employed as a potentially useful optimisation, relying on the presence of a strong type system to protect the representation from unintended[13] interpretation; while they could be applied to less strongly typed languages, they would then be restricted in application to areas and objects of the programme that happen in effect to obey a strong type discipline.

First, we might find that the ordering of the fields in a structure is not preserved; in particular an optimising compiler might reorder fields when it can be determined that more efficient access can thereby be provided. Among the more obvious reasons that this might be desirable are: that reordering fields can reduce external fragmentation resulting from data alignment constraints; that it can place heavily used or frequently-dereferenced fields at low or zero offsets, where many architectures provide preferential access; and that reordering, especially when negative offsets are permitted, can provide for cheaper support of (especially multiple) inheritance [PW90]. The compiler must naturally see to it that the proper field access code is emitted for the layout actually chosen.

Second, the implementation might determine that one or more fields can be combined into a single, joint representation. On one interpretation this is the theoretical justification for "null pointers" (which do not, in fact, point) in Pascal and C, and for the various infinities and NANs in IEEE floating point arithmetic; a different example is provided by "packed" structures in which information about the domains of fields is used to reduce internal fragmentation.[14] These rearrangements, of course,

[13]An intensional pun, this time.

[14]To understand this example it may be necessary to recall that the typical "integer"—for

particularly rely on the compiler having suitable information about the underlying representations of the components; one cannot (in general) arbitrarily assign unusually formatted or overlapping storage to different objects (as is done with the C union construction, for example, where a burden of recording disambiguating information elsewhere is placed upon the programmer) without loss of information.

Third, the same type might be represented in different fashion at different points in the programme. The most straightforward case of this would be where some or all of an object is moved into registers in a local context where it is known that no nonlocal reference to the structure is possible (or even globally to a function nest in the case of a global variable or common argument structure), or eliminated entirely in favour of immediate operands if it is constant[15]. More complex applications of this technique might divide the uses of a type into two or more categories deserving of different optimisations, and provide distinct representations, even distinct general-purpose representations, for these populations, with conversions supplied automatically as required.

Finally, one can imagine a case where the above techniques are generalised into a fully-fledged compiler facility, whereby the programmer or the implementation can provide multiple implementations of any type and let the optimiser choose between them and variants of them as it will, inserting conversions and modifying access paths to maintain consistency.

A special case of all of the above re-representations of a type is that in which some component lacks information content (relative to some context) or is "dead" in the data flow sense, and need not be represented at the object level at all. This is, of course, in direct analogy with the dead code elimination optimisation.[16]

instance—in a programming language is actually a *32-bit* (or so) integer, and so storing it in a narrower field, even where this is known to preserve the values that will be called upon to represent, does violence to its "obvious" *extension*.

[15]This has the effect of embedding data in the code and relies in general on the fact that code sequences themselves are typed by our criteria; see section 5.4.

[16]In the above we have stressed the familiar case of conventional data structures, but it is worth

In all of these cases it should be noted that the implementation is constrained in its rearrangement of the user's description of the type to respect *two* distinct interfaces: on the one hand, the external, ultimately intensional, behaviour of the type must be preserved (depending on modularity implementation constraints, as discussed below, this may be no more than a constraint on the behaviour of the programme as a whole, or it may impose requirements on whatever types—or instances of types—are exposed at module boundaries); and on the other it must respect a *downward* interface to the types in terms of which the user's type is implemented: the chosen extension, the representation, the actual data structure, must be a well-formed *underlying* object (or, possibly, objects). Crucially, however, it need not respect the letter of the user's mapping between them; and it is only necessary that whatever functionality is actually *used* of that which is declared be resynthesisable at need.

In sum, the implementation must *be* an implementation: it must preserve the observable behaviour of the type relative to its formal definition, and it must represent it at the lower abstraction level successfully and, one hopes, in a tolerably efficient manner. It need do no more than this.

4.3 Typing in Extant Languages

Before going further, let us turn our attention to the ways in which typing—as opposed to mere data structuring—manifests itself in various extant programming language families.

Practically every programming language provides for the manipulation of values drawn from a number of different domains. Typically these might include "integers" (often a very restricted subset of the mathematical integers), floating point numbers (frequently and erroneously referred to as "reals"—the presence of at least the classical

noting that precisely the same analysis applies to functions and (because this is a large part of what a functional type representation amounts to) their calling and stack formatting conventions. Even the tail-call optimisation is arguably dead-data elimination performed on the stack frame.

reals in a programming language would be *quite* undesirable in practise!), characters drawn from some character set, truth values, strings of characters, arrays of floats, "atoms" as in LISP, various kinds of functions and even—in some cases—types themselves [DD85]. These domains, possibly in conjunction with others introduced by the programmer, are the stuff of the type system of the language—though precisely how the type system is arranged varies widely.

4.3.1 Weak static typing: the post-Algol family

Perhaps the language group with which the word "typed" is most strongly associated in the popular mind is that of the post-Algol languages of the ilk of C and Pascal. This is somewhat ironic, since, although their type systems are very visible to the programmer, they are generally among the least strongly typed of all high level languages.

Among such languages as these the type system serves three main purposes: first, to assist the compiler in generating efficient code for the manipulation of data in the various built-in domains; second, to streamline the notation for the operations on these data; and finally, to provide data *structuring* facilities (like those described in chapter 3) to the programmer. Each identifier is associated with a manifest (and, except possibly in the case of array bounds, fully specified) type at its point of declaration, and a set of inductive rules is used to assign a fixed type to every expression in the programme.

The mechanisms for introducing new types are typically very stylised: the usual repertoire of defined types includes finite enumerations, integral ranges, records, unions, pointers and arrays of uniformly typed elements. In particular, first class functions and other objects of a high-order flavour do not occur. Historically, a strict notational and semantic divide has been enforced between primitive and non-primitive types; only in the more recent members of the family do mechanisms appear for concealing their defined nature. While this separation may serve in part to simplify the translator, it clearly compromises the goals of streamlined notation and modular opacity. It is perhaps for such a reason that the drawing of a sharp distinction between atomic and compound objects is among the objectives that seem to have been dropped from the structured programming agenda in these latter days when scale tells.

It is not clear whether as a cause or an effect of this situation, but the implementation manuals of these languages often provide detailed descriptions of the actual storage layouts used by the compiler for various data structures. This presumably helps facilitate interfacing to foreign software components without requiring the provision of facilities specialised to that end. It *is* clear that this would be far less feasible in a more opaque system or one in which higher order objects appeared.

Although the mechanism by which expressions are assigned types in languages of this class is potentially sound, still we find that their type systems are insecure. While not every such language provides the same set of loopholes, each seems to have one or more escape mechanisms for stepping outside of the type system, and there are a number of ways in which this can be exploited by the programmer (see section 4.5). While there is good reason to believe that such evasion of type discipline is not a good thing, in *these* languages, at least, it appears to be a necessary evil: for in the presence of a barely fulfilled promise of high performance, they combine rigidity of structure with poverty of expression in both code and data. Something must be done to ease the programmer's lot. Thus there are locutions provided in which the type determined for an expression by the inductive type assignment algorithm is at odds with that of the value actually delivered, where no semantic fixup (as there would be with a coërcion) is provided, and where the construction is not formally erroneous (though it is typically of "undefined"—which is to say, underspecified—effect). So, for instance, the declaration

```
int strange = *(int *)(void *)"odd";
```

is legal C, though it results in a character array being loaded directly into a signed integral variable, with unspecified effect. Similarly, most Pascal compilers will admit

```
type
  dodgy =
    record
    case Boolean of            {omission of discriminant explicitly permitted}
    true:  (r: real);
    false: (i: integer)
    end;

var
  dodge: dodgy;
  strange: integer;

begin
  ...
  dodgy.r = 3.14159;
  strange := dodgy.i;
  ...
end.
```

The official intent of these constructions is often that they are merely cases where the type correctness of an expression is not *statically* determinable, but is placed as a burden on the programmer; the results of these operations are then defined when, and only when, appropriate dynamic conditions are met. In fact, however, they seem to constitute an incitement to ignore the programming model provided by the high level language in favour of an operational outlook on what the particular implementation actually does. Were this not also the mind of the language designer, we would anticipate the generation of far more dynamic type-checking code than we tend to find in actual implementations (and the examples above would be trapped as erroneous).[17]

[17]Such error checking is easily accomplished: in the Pascal example we have but to represent the anonymous Boolean discriminant field, while in C the anonymous "void *" could be made a duplex structure encoding the type of its referent directly.

Although these mechanisms manage to provide for a number of possibilities that would otherwise require some combination of a more sophisticated type system, an explicit external interface mechanism, and higher order objects—possibly, granted, at some loss in efficiency—it is our view that they do not constitute a net gain, in that they make it impossible to exploit some of the more interesting possibilities of the typed paradigm (such as automatic storage recycling and the **Visitor**-based facilities to be discussed in chapter 6) and, perhaps more importantly, tie the hands of the compiler by compromising the semantics of the language itself and restricting it to particular, transparent, implementation strategies.

The overall effect is that the type systems of these languages provide not so much interface contracts as "gentlemen's agreements"—having in particular the characteristic that they cannot be relied upon by third parties.

4.3.2 Algol 68 and the strongly typed languages

Algol 68 itself, by which the languages of the previous section were at least somewhat influenced, is a language of which the type system is both less expressive (though it does indeed provide for function values under not intolerable restrictions[18]) and completely inviolable. The representational abstraction provided by the language is opaque. This results in a language which can be (and is) garbage collected.

Algol 68 itself is hampered by an extensional approach to typing that is at odds with the necessarily intensional demands of software engineering. The modern descendants of the Algol philosophy include the strongly typed functional programming languages [MTH90, HPW91, Tur85], often employing extended type inference techniques as in [Mil78, Gro85] to reduce the burden on the programmer of providing detailed type information. These newer languages have moved in the direction of more intensional models of typing and, in the case of Haskell, have added a semi-

[18]Basically that they only be used when their scope of declaration is still live.

systematic approach to (static) subtyping.[19]

4.3.3 Dynamic typing

Among the historically interpretive languages like APL and its descendants (notably J), various LISPs and SNOBOL (and subsequently ICON), we find a different pattern. In these languages there is a type system and the typing (as with Algol 68) is *strong,* in the sense that no action of the programmer can result in an undetected violation of the type system.[20] Usually, however, they are characterised as "untyped." We can provide two distinct analyses of what precisely is going on.

On the first view, these languages provide objects of only *one* type, itself the (discriminated) **grand union** of all the domains from which valid data may be drawn. Since there is only one type for any expression, there is no opportunity for a type violation, and strong typèdness follows *a fortiori*—and trivially—from this circumstance. On this reading, the various "types" of the language are merely the manifold distinct variants of a single underlying object type: the language is (we might say) **unitypic**.

On the contrary, we could understand such a language to possess a nontrivial type system (with each of the data domains constituting a distinct type), but with type checking performed *dynamically,* during programme execution, on the basis of the histories of objects, rather than as a (static) property of the expressions in which they appear.

Technically speaking, the former interpretation is more generally accurate: operations are commonly provided to *enquire* as to the form of an object, and implicit in the operation of enquiry is that at least enough is known about the object's type to pass it as an argument, which from the static (and hence—since the object code is

[19]Though its "derived instance" mechanism is apparently at odds with this interpretation.
[20]At least if we account the raising of an exception as successful avoidance of an error—certainly the programme in this case continues to exhibit well-defined behaviour.

itself static—implementational)[21] perspective is tantamount to its type being already known.

That this "type" information is best viewed as part of the value of the datum is brought out clearly[22] by the fact that notational complexity is frequently reduced by making the operations of the language dispatch on the types of their arguments, their possibly behaving very differently in each case (an operator "+" for example, might designate unbounded exact addition over integers; bounded, approximate addition over floating point numbers; concatenation over strings, and so on—note that these operations are not even algebraically similar, since the second does not associate and the third does not commute). In Smalltalk-80 this notational device has been raised to the status of a major programme structuring facility: the (externally) interesting characteristic of a class being not how its members are represented, nor even what class these objects actually belong to, but what message protocol they observe—what, that is, function *names* can be applied to them.[23] Thus, for instance, the set of objects that respond to (at least) the message printOn:at: would form a type, as would those that respond to all of +, − and *.

This extension of class-dispatch to a primary semantic feature, and the concomitant enrichment of the class system, was linked with the alleviation of one of the common problems found in LISP, namely that although such dispatching can be important to the operation of software (because in the presence of such a system information is in practise frequently encoded in object class), the comparatively static and inexpressive type system provided by LISPs frequently provides insufficient reso-

[21]Unless compilation is being carried out at runtime—and thereby hangs a proportionately epic worm-can.

[22]Though see [Gro85] for a fairly successful approach to the reconstruction of the facility within the constraints of static typing—where the operat*ors*, not operat*ions*, dispatch according to "actual" type.

[23]Smalltalk-80's adoption of the idea that functions have distinct—and potentially unrelated—behaviour for each class of object they might be applied to, would appear to *suffer from* severe difficulties of namespace control, since it provides no mechanism other than inspection to relate the various uses of a message name.

lution for fairly straightforward applications. In particular, many LISPs lack a clear distinction accessible to the programmer between lists and functions, while at the same time eschewing a well-defined relationship between them. Similarly, the only way of constituting a new "flavour" of list is to recode the entire list facility on top of some variety of record type (assuming the latter to be available). This manner of difficulty is in fact quite typical (though far from necessary).[24]

As has been observed, the languages of this category are all at least descended from interpretive stock. In most cases, of course, partial compilation is performed, and full compilation to native object code is by now commonplace. In the process of compilation, optimisations may be performed which have the effect of changing the flavour of the type system at the implementation level (though of course those compilers which implement different semantics from those with which the language was defined can scarcely be said to be correct—or rather, to be implementing the same dialect). For instance, even rudimentary static data flow analysis (interprocedurally or just in combination with information about the behaviour of some set of "primitive" functions) is often sufficient to determine that certain expressions can only be used to manipulate values of some fixed class. Then the elimination of code necessary to deal with other classes and even the discriminant tag specifying which class is being manipulated can be accomplished in the course of normal optimisation processes, resulting in a system that is in the usual case effectively statically typed in the sense of Algol 68 (along with most, though not all, of the efficiency benefits this can bring), though dynamically typed in principle. Such a system can actually recover *more* information than would be provided explicitly by the "equivalent" statically typed system (though there is no reason why statically typed systems should not avail

[24]It is altogether unclear how such a distinctly functional programming language has managed to get by with a weak and inconsistent conception of the nature of a function. Perhaps—to wax cynical—it is because the bulk of the LISP code in the world would have made better FØRTRAN—data structures permitting.

themselves of the same techniques and make further gains in turn).[25] Some of the most aggressive results in this regard have been achieved in SELF [HCCU90].

The text-based record processing language Awk is not atypical of another group of "dynamically" typed languages. In Awk, variables are typed by their first use, and subsequent uses of a variable are required to respect the type already determined. There is no question, however, that the language is typed: an attempt to use an array-valued variable in a scalar context, or vice-versa, results in abrupt programme termination. Among scalars there is no further type distinction: numeric and string values are transparently interconverted as part of the semantics of the primitive operators of the language, and this conversion is accomplished dynamically. As with LISP, a sufficiently devious programmer can encode information in the actual variant according to which the data are stored.

4.3.4 Lexicosyntactic typing

A quite distinct notational approach (though one ultimately having the same implications for the runtime system as a language with strong, static typing) is found in Perl, INTERCAL, many dialects of BASIC and perhaps the earliest versions of FØRTRAN. Here the binding of types to expressions is performed even earlier than in more typical statically typed languages: it occurs at the moment of the programmer's original selection of identifiers, the forms of variable names explicitly encoding their types. There is a vague argument, perhaps, that this practise makes a programme clearer, since the reader can be in no doubt as to what the type of a given expression might be;[26] yet this is bought at the expense of a type system so restrictive that only a handful of types exist—for otherwise, the coding-space overhead of maintaining explicit type information in every identifier would be prohibitive. Furthermore, it is not evident that the clarity of a programme *is* increased by reducing the expressiveness

[25]This is standard practise in both APL and COMMON LISP compilers; COMMON LISP and SETL provide (very different) schemes of pragmatic declaration to similar ends.

[26]Though Perl still consistently confuses the author.

of its type system to the point that manipulation of all but a very few types requires extensive convention and gross circumlocution. Even in the case of a language intended for a very limited application it is to hard to argue against the provision of some support for general-purpose programming.

In clause-structured languages (where the scope of a variable name is severely restricted and the mnemonicity of names can be compromised to some extent) this difficulty has sometimes been attacked by reversing the relative weights of the two components of such type-tagged names, almost eliminating the name of a variable in favour of that of its type. This produces identifiers like expression-1, Person#3 or NP_1. Most commonly this is found in special-purpose languages such as expert system shells, attribute grammar formalisms and, for that matter, the semiformal notations employed by mathematicians and linguists.

4.3.5 Full typing: ITT

Developments in mathematical type theory have given rise to a new class of programming languages: those based explicitly on a powerful mathematical type formalism. One influential example is that of Per Martin-Lof [sp?]'s Intuitionistic Type Theory ITT [ML84], which has been implemented directly as a programming language at least twice: once by Constable *et al.* [CAB+86] as the (automatically generated) proof language underpinning the NuPRL proof development system, and once as the total correctness language Brouwer [Dev84]. ITT in particular has been a heavy influence on the (far more pragmatic) type system with which we work in this thesis, and underlies our belief that "escapes" from the type system are unnecessary in almost all practical applications.

These type systems provide not only functions but types as first class objects, but types can nonetheless be interpreted in a fully static manner (due in part to language semantics independent of actual reduction order), and require in theory no runtime representation—though the finite resources of practical machines can be greatly extended if some runtime representation of types (possibly in the form of **Visitors**) is provided.

4.3.6 The raw machine

One final category of programming language, and with yet a different perspective on the issue of typing, is that in which access is provided more or less directly to the underlying machine. Here we find both the multitude of assembly languages and the more-or-less high-level language FORTH.

The issue of whether, and to what extent, an assembly language should be understood to be typed is a tricky one. On the one hand, it is clear that a typical assembler does very little checking for compatibility of run-time values (though static checks for the validity and compatibility of compile-time objects like formal addressing modes and constant values is often extensive): usually integers, pointers, flag-masks and floating point values can be intermixed at will without a peep from the translator. On the other, it is equally true that the behaviour of typical hardware is fully defined (and more or less documented) in the face of whatever manipulation the programmer might attempt. All the information necessary to understand (*e.g.*) what will happen if a single-precision floating-point π is loaded into the supervisor status word register is in fact available to the programmer—and the result is typically well-defined.

Thus, as with the dynamically typed languages, there are two distinct readings of what is going on, depending on how much significance is placed on the conceptual domains being manipulated. If primacy is given to the domains, we have a weakly typed language (possibly one with fairly sophisticated and expressive data structuring facilities, as with ASM-86). If, on the other hand, the existence of a hardware reference manual is taken seriously, it is better to argue that there is only one type, some form of bit-vector, and all manipulations are performed over these. Under this latter interpretation we see that typical hardware architectures are indeed strongly typed.

Unlike the dynamically typed languages, there is certainly no operation that can be performed at runtime to determine the class of a value (though one might conceivably argue this point for the LISP Machine [Moo85] or the SPARC [sun87], which possess tagged pointers, it seems more reasonable to see this as simply a difference in the processing of "dereference" for various argument words, since it is indeed intended that tagging "violations" be trapped), but nor is there any difference in the processing of values of different "types." The only instances in which there seems to be any real case for a theoretical type distinction in an assembly language are those where the assembler provides *transparent* access to an unsafe architecture which (like MIPS [Kan87]) really does exhibit undefined behaviour on certain inputs; and the matter of the code/data distinction, in that most modern architectures require code to be static modulo some notification procedure, while placing no such restriction on data. The majority of assemblers, furthermore (ASM-86 is an exception), provide only symbolic access to the instruction set, while data can be decomposed statically into bits.

In the same category with assembly languages we place the rather peculiar FORTH. FORTH is in many ways a greater success as a "portable assembly language" than C, though C is often accused of being precisely this. The reason is that although the machine on top of which a FORTH runs is a software construct, entirely virtual, the source language actually succeeds (issues of standardisation apart) in providing the kind of transparent access to it that motivates many assembly language programmers. At the same time it provides a high-enough-level programming environment (in particular, one in which the definition and use of functions is comparatively low in overhead, and where syntactic mechanisms are never reserved to the language core) to be convenient for many applications, particularly undertakings like prototyping where interactive development is of greater moment than absolute security.

Although its extensibility (or incremental replaceability, perhaps) somewhat clouds the issue, a typical FORTH provides a notion of type that is comparable to that of less sophisticated modern assemblers running on insecure machines: type does exist in some sense, is crucial to the programme executing in any defined manner at all, and is in no way checked by the compiler. From the point of view of security this is the worst combination of all, and, as with the languages we first considered—those exhibiting weak static typing—it is by no means clear that anything has been gained in expressiveness by this outlay—though since the FORTH programmer is actively encouraged to tinker with the compiler, the reduction in compiler complexity in this case may constitute a justification in itself.

4.3.7 Underlying tags

In a few instances the underlying machine may in fact provide tagged data. One such case is that of hardware optimised for the support of object-oriented or LISP-like languages, where it is deemed that a sufficient proportion of the data that the machine will manipulate are required to be tagged by the language semantics that direct hardware investment is desirable (though recent research suggests that such effort may not be entirely well-advised, in light of the cost of boxing/unboxing operations and the unfavourable interactions between tagged representations and optimisation); another, parallel to the "raw" (but virtual) FORTH machine, arises when the underlying (software) execution mechanism already employs tags to its own ends.

In particular, normal order evaluation can be implemented quite effectively using a data representation in which all objects are directly executable, with types (at the implementation level) being determined by tags at object heads which are themselves short, executable code sequences (this is apparently an oft-invented wheel. See, recently, [KL89].[27]).

[27]The widely distributed Unix "TIGRE" implementation of Alonso is not (despite the claims of its documentation) in fact implemented in this manner, but with a conventional table-driven "byte-code" interpreter. Coding in strict ANSI C (and thus paying an extra indirection over assembly code when locating functions), and running on the SPARC (which admittedly provides very cheap

In both of these situations the cost of a strongly but dynamically typed, tagged, language implementation is apparently reduced by the fact that external factors independently call for this approach. This independence is illusory, however: in the case of the machine with tagged memory, the cost of tagging is still present, but has been assumed by the hardware design and burned into silicon—in a sense, the hardware engineer has taken the language implementation decision to adopt a tagged representation (a decision which, perhaps, ought best not have been taken) out of the compiler writer's hands. In that of the software graph reduction engine, we find that the best available interpretation techniques (as, perhaps, [AJ89b], with Lazy ML) are moving away from representations that require the tagging of every data object.

It has been suggested at various times that the reason the arrow of optimisation points away from tagged representations and dynamic typing is in fact the underlying unsuitability of classical von Neumann architectures for this kind of data; and that a change in architectural perspective might reverse the preference (see the critique in [DP80]). Given that in a tagged representation resources (whether hardware or software) must be devoted to the *dynamic* expression and manipulation of tags representing type information that even in a dynamically typed language is almost entirely predictable (see, for instance, [CU91, DS84]) which could otherwise be employed to further the computation, it is hard to see how the argument might go through.

We hold, then, that tagged representations may in fact be preferable in the presence of substantial underlying hardware or software support, but that the designers of these understrata would do as well to listen to arguments against tagging as would the designers and implementors of programming languages to run on more conventional platforms.

function call for programmes with limited stack travel), the author was able, in the database retrieval language Alfonso, to better the performance of this interpreter by more than a factor of two, largely by adhering more closely to the ideas of the original TIGRE paper.

4.4 Simple Types

In this section we examine the various basic types that we can expect to find in a conventional programming language's type system. In part it will constitute a recapitulation of the survey of data structuring found in chapter 3; but here out perspective will be from a higher level and we will touch on a number of types (in particular function types) which would not normally be considered data structures in their own right (though surely they must be represented as structures underlyingly).

4.4.1 Primitive types

Type systems in general set about constructing the set of all types available in a language inductively, from a basis of "primitive" types that are inherent to the language. It is possible in principle for this collection to be very small (indeed, implementations ultimately make do with bit vectors and bit vectors only; and as is well known from mathematics, character strings, integers, lambda abstractions—any of these will also do), but in practise the hardware provides particularly efficient support for certain kinds of operations and it would compromise the overall effectiveness of the computer system not to provide some suitable abstraction of these operations to the programmer.

Theoretically, it would suffice to construct a compiler sufficiently intelligent to *recover* efficient implementations from a programmer's abstract, mathematical descriptions of data by recognising which structures can be modeled efficiently on the target machine; but this is an approach that is only starting to be explored in code generation, where data structure selection is taken to be already a given [Wen90]. The closest that is seen to this in "practical" languages is that some specify many of their standard types to work "as if" they had been introduced as defined constructs—though a programmer trying it out independently will surely not get the benefits of the specialised knowledge that went into the implementation of that standard kernel.

This being said, it will be understood that (at least in the author's view) there is little *theoretical* meat in the issue of what is primitive and what is not; if the categorically-motivated building blocks **2**, **1** and perhaps **0** be provided along with enough structuring and information hiding to support basic engineering needs, we are home free. The next few paragraphs are thus devoted to practical survey and polemic; the theory will resume in section 4.4.2.

Void: The simplest of the usual atomic types is the Void type, **1**, erroneously described in the ANSI C standard as having no members. Actually, it has exactly one member[28], and thus zero *information content* and no need of a distinctive physical representation—whence, presumably, the confusion.

The theoretical importance of Void is that it permits the formal expression of computations which manipulate no information explicitly: where computations are performed for effect, or where something must be specified for the sake of consistency but is in practise not needed as a value. The former permits the collapse of the historically distinct notions of "function" and "procedure" into a single consistent construction; the need for the latter is very common when specifying interfaces with scope for future enhancement: where there is as yet no interaction between two modules but the path by which the necessary non-information is transmitted (and more information may later pass) must be recorded in the source.

Boolean: The second smallest type of wide instance is **2** (in computer science more generally known as Boolean). The Boolean type has just two members typically understood as representing falsity and truth, and are used to represent, for instance,

[28]For those who enjoy syntactic party tricks, there is even a formal argument to be made that it has notational existence in C, as an explicit zero-character expression akin to the *traces* of contemporary theoretical linguistics: we may write, *e.g.*, "return ;" to leave a function with return type void, while a function whose formal parameter list is "(void)" may be invoked with a textually empty actual parameter list—note the absence of the minus one commas that strict consistency would require to separate its zero arguments!

decidable decisions.

The fact that it is the type most frequently given special syntactic attention—in the form of the *if*-statement—goes deeper than the universal employment of binary representations in contemporary computers (in fact the representation of Boolean values by single bits is astoundingly rare in software): according to most accounting methods it is the simplest mathematical structure capable of representing information (and in fact this is *why* it is a good bet for hardware—all else being equal it maximises noise immunity).

Integral Types: The other class of primitive types that commonly enjoys a clear privileged relationship with the syntax (this time, in the form of *for*-loops) is that of integral types. These represent some part of the set of integers, $\mathbf{Z}$, usually—but not necessarily—with a very small and very hard bound on magnitude.[29] Fixed-limit integers are, of course, usually stored as bit-vectors of predetermined size representing polynomials in 2 of small degree. One might wish that programming language designers would stop using the word "integer" to refer to these other types, at least when the limits are not both checked by the system and selectable by the programmer.

Nonbounded integral values (at least, ones for which the bound is the amount of memory available) can be represented easily enough in a conventional computer, but they are of necessity variable length structures requiring more complex management than their fixed-range counterparts (though the fact that *most* values are still likely to be very small suggests an efficient two-level strategy).

The *for*-loop is justified as an independent control construct by the fact that integers must be handled inductively (while enumeration is adequate for Booleans—

[29]The natural numbers, N (typically taken in computer science as all the integers greater than or equal to zero), is often provided as a distinct type. In part this reflects the difference in likely meaning between values that are naturally represented in each of these two ways, and in part because representing the sign information consumes an *entire bit* (!) of storage, and so is traditionally avoided where possible. (To be fair, typical multiplication and division and division algorithms operate on unsigned values and there may also be a performance benefit in using unsigned values where possible on common hardware.)

though the one-branch *if* is itself, technically, a Boolean induction).

Further Arithmetic Types: Other arithmetic types that frequently manifest as primitives in programming languages are fixed point numbers, rational numbers, floats and reals. Fixed point numbers are essentially no more than scaled bounded integers, and floating point numbers (while frequently and misleadingly referred to as "reals") are fixed point numbers where the scaling factor is itself represented by a bounded integer—taken as a power of 2, most often, to provide some representational purchase. While it is not especially clear that floating point numbers (especially hardware-supported floating point numbers) are of any great use for general purpose computing, nonetheless over half of a typical modern CPU is devoted to their manipulation, and one supposes that it is the language implementor's sacred trust to find something to do with all that silicon, or something.

Implementations of the rationals, **Q**, are typically much more thorough and, like the nonbounded integer, provide as much representational precision as can be fielded on the machine at runtime.

When it comes to **R**, the "real" reals, computers have a problem, because there is a serious mismatch between their *computational* properties and their *theoretical* ones, since any function manipulating a raw real value would appear to need to process an *infinite* amount of information, making it difficult (to say the least) to assure termination, even assuming that an appropriate representation is available. The price of implementing a useful form of real would seem to be the adoption of the framework of the constructive reals, and the consequent abandonment of total ordering: constructive reals can provide a semidecision on whether two numbers are within a given rational of each other, but no more. While this takes some getting used to, it is nonetheless the case that they are an appropriate type for those tasks where fixed precision is inadequate, may be amenable to tolerably efficient implementation

[LB90], and may be expected to become more common in the near future.

The designers of Haskell have, as it happens, further muddied the waters of numeric nomenclature by using Real as the name of the class covering all of the language's basic arithmetic types other than the Complex. This is unfortunate, since in consequence it provides a total order as one of its aspects—by inheritance from Ord—effectively barring implementations of the reals from membership in class Real.

Enumerations a Semi-Primitive: Many programming languages provide *finite enumerations* among their repertoire of data types. It is somewhat unclear whether they belong in the present section, since on the one hand they can be held to be (sometimes opaque) renamings of sets of small, fixed, cardinality—in which case those underlying sets are indeed primitive types in our sense; but on the other, syntax that introduces such a type can be seen as a constructor of an entirely new type, in which event they technically belong in section 4.6.1. We will take the very pragmatic view that they belong here for no better reason than that they are uninteresting: they have the same formal and representational properties as bounded subranges of the integers (though some languages weaken their semantics, providing no ordering or arithmetic over them).

The same cannot be said of free algebras, though they are a straightforward generalisation of enumerations; in fact, free algebras (along with other "sum" types) must await the introduction of dependent typing in section 4.6.

Characters: One type that is generally defined by enumeration but is nonetheless an indubitable semantic primitive is the character. Perhaps most programming languages provide a character type representing, unfortunately, not the glyphs of arbitrary human languages, nor even those of written English (one would have thought that æ and œ, at least, were unassailable, but it appears that œ was deliberately excluded from the most widely used standard for eight bit character encoding, the combination of ASCII and ISO Latin-1), but *some* set of (tokens for) linguistic glyphs, often containing at least the ASCII 7-bit coded character set.

Some languages also provide *strings* of characters as a primitive type, but there are many variants on this theme and for the purposes of this thesis they are all compound objects—modulo surface syntax (and perhaps the shortcomings of a particular language) they are subject to all the same considerations as arrays.

Atoms: The case of "atoms" (values which have no other properties than that they can be tested for equality and serve as a domain for partial functions) is interesting, for there are again two quite distinct interpretations available, depending in this case on the presence of a "gensym" operation that can produce new atoms at runtime. In the *absence* of such an operation, a type of atoms is no more than yet another enumeration, declared distributedly over, perhaps, several source modules. If gensym is available, however, the type of atoms is, like the true integers, of nonbounded cardinality.[30]

Either way, of course, the "traditional" implementation of atoms is as pointers to (or indices into) structures which contain information to accelerate the manipulation of sets they may belong to or relations they may enter into; a representation which shares nothing with enumerations but that a fixed-width field suffices to store a pointer (untrue in a model where memory is unbounded, but true enough of any contemporary hardware).[31]

A Menagerie: Programming languages, particularly special-purpose programming languages, provide in addition to the "usual" types above, a bewildering array of

[30]An intriguing hybrid possibility is to treat atoms according to the usual identity rules for local variables: that atoms with the same lexical name have distinct values at each activation of their context of definition, this still being understood as part of a single distributed type definition.

[31]But as noted by Wilson in [Wil91], Ralph Johnson of the University of Illinois at Urbana-Champaign has suggested an efficient approach to nonbounded (virtualised) reference spaces based on "pointer swizzling" (software pointer virtualisation supported by guard pages in virtual memory).

primitive types for particular applications. These could include, but are not limited to, files, formats, wombats, windows, pens, parsnips, patterns, colours, sounds, soups and such. Internally to a von-Neumann-hosted implementation they will all be bit-vectors (with the remote possibility of their being behind a layer of operating-system abstraction hiding their bit-vector-hood from the language); while from the outside their properties are thoroughly arbitrary.

4.4.2 Product types

Turning to operators for constructing *non*atomic types, we find that the most straightforward is the **product**, building types whose members are **tuples** (or, in computer-science-ese, **records**). In its most formal guise, the product constructor is a binary operator, $\times$, over types, and the elements of $\mathsf{T} \times \mathsf{U}$ are all pairs $\langle t, u\rangle$ with t in T and u in U. In most programming languages, and indeed *Exemplar*, the product type constructor is variadic, and the element tuples may, correspondingly, be of any length.[32]

In order for product types to be of practical use, it is necessary to be able to extract the elements of their members. There are two basic notational arrangements to this end. In one, the pattern-based approach, there is pattern syntax mirroring the data syntax in each product type—matching a tuple (of data) against a tuple (of patterns) resulting in bindings according to each successive pair of elements and effecting decomposition. In the other, **projection functions** may be associated with the type when it is constructed: these are functions that take elements of the product back into one or another of the original component values. The most significant difference between these approaches is in fact an engineering issue: the provision

[32]There is an important technical point here: that the *empty* product, () (to use *Exemplar* notation), must (if it be permitted) have precisely one element, []; it is isomorphic to Void. Singleton products are isomorphic to their argument types; whether they are interchangeable with them is essentially a notational issue (having mostly to do with the manner of invocation of the projection functions, described below: pattern-based notations are likely to make them identical, while notations with explicit projection operators will treat them as isomorphic copies, lest the projection function be semantically anomalous).

of separate projection functions requires that some syntactic method be found of *naming* them, while pattern matching can be purely positional (circumstances seem to determine which is best).

Back at the abstract level, then, a product constitutes no more and no less than a general, abstract method for taking objects of any (without loss of generality) two types; building from them a *single* value; and providing that that single value suffices for the reconstruction of either or both of the original items.[33]

There is a tradition in computer science of providing projection functions in particular with a distinguished syntax (often a tightly binding postfix notation), different from other notations for function application. Pragmatically this arises from the particularly efficient implementation that is available for these functions (in the obvious representation they amount to offsets in addressing modes, something even RISC architectures are happy to provide in one instruction), and from the fact that they are conventionally *mutably valued* functions, a situation which is often otherwise outlawed. If they *are* mutably valued, of course, this provides a secondary mechanism for building values in these types (by altering old instances), just as a pattern-matching syntax may be thought of as a secondary arrangement for elimination.

Two places where product types are very familiar but not often recognised are as **argument lists** and **declaration blocks** in Algol-family languages. Each of these introduces an anonymous state tuple with named fields which are accessed as variables within their scopes; the scoping rules of the language amount to rules for the reconstitution of the unwritten tuple names that qualify each use of a variable—the filling-in, that is, of the implicit argument to the projection function the identifier names. The very practical reality of this outlook will be of importance to us later, when we consider the actual application of **Visitors**.

[33]Given access to the underlying *representations* of two objects we can of course always build a special-purpose product—this is the tedious part of the familiar Gödel-numbering construction—and this is why products are generally implementable. That we are here generalising over types will ultimately, however, be crucial.

4.4.3 Functions

An important special case of the product is that in which the argument types are homogeneous: the product of n copies of some type T, written T^n, is no other than the familiar algebraic **vector** or the (unidimensional) **array** of computer-science. Here the elements—and hence the projection functions—are in one-to-one correspondence with some initial segment of the natural numbers, and it is convenient (and conventional) to provide a mapping from the naturals to the projection functions of the type (though once again the notation is apt to be specialised and the projection functions may not be usable in general "function" contexts).

Even if it is desired to be able to index the projection functions, it is not conceptually necessary that the indices be small natural numbers; the members of any specified finite type will do.[34] Programming languages in fact commonly provide arrays indexed by characters and other enumerated types. Let us write (*par abus de notation*) T^U for this more general case.

There are two superficially distinct constructions that can result in an array of more than one dimension: either a vector of vectors can be constructed (resulting in something of type $(\mathsf{T}^U)^V$); or a single array may be built whose index type is in turn a product (yielding some $\mathsf{T}^{U\times V}$). These two objects are clearly isomorphic: in either case, each distinct pair of indices maps to a unique array element of type T.

A convenient notation for array initialisation, and one more in consonance with the indexable projection functions that distinguish vectors from general products, is to use an explicit *algorithm* mapping the index values to the array's initial membership If the members are further immutable, then there is an obvious space optimisation: storage for the array proper need never be allocated, and the elements can be computed and recomputed from the initialisation code as needed—care being taken that

[34]Actually it is possible to view *arbitrary* products as being dependent maps (see section 4.6.2) from an enumeration of field tags to the field type(s). Meta, for one language, takes this approach.

enough original context is saved that this remains possible. Such a scheme is entirely practical, and is the conceptual foundation of the **functional** programming languages.

This leads in turn to the observation that, it not being necessary *ever* to represent components not actually visited by the programme—at most a finite number of values in the lifetime of the hardware—the index type need not be small or even *finite*—it can be any type at all. Repenting of our earlier notation-abuse, let us write this generalised type of **functions** from T to U thus: $\mathsf{T} \to \mathsf{U}$ (note that just as we restricted ourselves to considering the *constructive* real numbers when discussing primitive types, here we are not embedding the *classical,* set-theoretic functions in our language. Rather these are, again, constructive functions: ones that can be expressed by finite algorithm).

A variant on this strict interpretation of a function recognises explicitly that function bodies are re-evaluated whenever their "elements" are referenced, and permits algorithms to make reference to the dynamic state of the computation at the textual moment of reference (using the usual preörder traversal). Through the tender ironies of history, this is known as "strict" evaluation, and is still the standard way of doing things.

As before there are two alternate mechanisms to build what is commonly known as a function of more than one argument: we can either encode the argument list as a tuple (as already noted, conventional programming languages are perhaps best analysed as giving each function a *single,* tuple, argument), giving a type like $(\mathsf{T} \times \mathsf{U}) \to \mathsf{V}$; or we can use the **curried** representation[35], assigning the same function text (modulo, perhaps, some qualifiers on the argument names—depending on the details of the syntax for destructuring, in fact, as discussed above) the isomorphic type $\mathsf{T} \to (\mathsf{U} \to \mathsf{V})$. There is a complication: the curried type requires that functions be *first class values*—in particular, the application of a value in T to the curried function

[35]Currying is a manipulation named after Haskell B. Curry, the famous combinator, wherein functions in $(\mathsf{T}\times\mathsf{U})\to\mathsf{V}$ are transformed into (semantically equivalent) functions in $\mathsf{T}\to(\mathsf{U}\to\mathsf{V})$.

yields a *result* in type $U \rightarrow V$, and the implementation must provide for its reliable and accurate representation.

This highlights the crucial abstract property of these function types (of which arrays are but one variety): a function is an object *transforming* objects of its argument type to objects of its result (*i.e.* element) type, and the type of the function itself is, if you will, the contract under which this is performed: given an object of the argument type the function will not return anything but a member of the result type (and in some languages–such as Brouwer—cannot but return such a value).

This brings us to the oft-used concept of the *partial* function, a function with elements possibly missing from its domain (in the sense that no member of the range corresponds to them). These are used to represent various partially initialised objects, functions which are not computable for all values (and so sometimes fail to terminate), and functions which cannot complete for some pragmatic reason—opinions differ on where the line of pragma might be drawn.

In fact it will be observed that there are two entirely distinct ideas here: there is the function that does *not* return—whose distinct existence is of interest to a type theory *qua* logic, but not, to the author's mind, to a type theory *qua* computational engineering tool (unless, of course, the *logical* interpretation has been granted a very practical reality), if for no other reason than there is not much to choose between an unacceptably inefficient function and an infinitely inefficient one, when one has knocked and an answer is required [Mil26].[36]

Then, there is the function that on occasion successfully returns, with the wrong type of answer. With luck it returns an "exception value"; more often it prints a message and halts the programme—or the machine. This, on our analysis, is a misunderstanding, no more and no less: since this exceptional value (or message) ***was in fact returned*** from the function, it must *ipso facto* pertain to the function's

[36]In any event, if static determination of termination of every function is required, either the language must be severely restricted or the compiler must be permitted, in its turn, not to terminate.

range. *Exemplar* therefore provides the extension value mechanism and in principle that is that (in practise, of course, a non-locally-exiting exception handling mechanism is very useful—and the language from which *Exemplar* is subsetted calls for one—though thus far it would appear to lack sufficient theoretical grounding for us to address the matter here. As with many such things, **Visitors** should in fact apply ***mutatis mutandis*** to the underlying representations of programmes with fairly arbitrary exception handling facilities).

4.4.4 Sums are not so simple

The obvious next stop in a survey of standard types is the *sums:* types which can represent elements of any of a number of other types (again, without loss of generality, mathematical notation favours exactly two, while computer scientists favour free choice)—singly. An element of any one of the component types can be injected into the sum type, and subsequently extracted unchanged; but of course, elements of types *other* than that which was provided cannot be extracted, even if they number among the type's full list of options. It must also be possible—somehow—to determine which of the possibly representable types is in fact represented, for otherwise the extraction operation would itself not be reliable.

In fact we shall pass sums by for now, saying no more of them until section 4.6.1, since in our present minimal framework we cannot express the "contract" represented by these types: for what is "the type" of the value resulting from the obvious projection operation from a sum type back to its captured value?[37]

[37]Actually there are two workarounds in the present context, neither of particular interest to our development: we could, with some theoretical justification, argue that the elimination operator for a sum should *not* be presented as a projection function but as a function accepting a disjunctive object (an object drawn from a sum type, *i.e.*) and a sequence of functions each of which accepts an argument from one of the summands; then only the appropriately corresponding function is called, and it determines the result of the entire expression. This technique relies on the index types having (functional) +/—-position-dual rôles in sums and products. The second and far more down-to-earth workaround—nay, kludge—relies on knowing *some* element in each of the summand types, and representing every sum as a *product* over the same types—setting each "absent" or inappropriate field to the known (default) value. In a physical computer, of course, the resulting value is much larger than it needs be, and in the presence of equality the same excess of information content arises

4.4.5 *& Cetera*

Here we mention briefly a few more items that we would rather not, but without which the section would be incomplete. These are all items whose existence is a practical reality in programming languages, but which do not (yet) fit cleanly into the kind of pattern we have been describing.

Mutables: Most traditional programming languages provide a facility generally referred to as the *variable*—not the *functional* variable or the *logical* variable, but the garden-variety side-effectual *imperative* variable. These variables—to avoid old confusion (if only at the expense of creating new confusion) we shall refer to them as **mutables**—have the property that "from time to time"[38] they may change their values[39]—usually (if not always) on the express instructions of the programme. The mutable is typically modeled as something with a *name*—a variable (in the generic sense, now), a field in a record, or an element of an array (though not, for instance, the value of a general function). In Algol 68, mutables are identified with *references*—still "named" objects, but with names now themselves computational values and no mere textual identifiers (on which see the next subsection). COMMON LISP, generalising still further, sees a mutable as a "place"—just about anything, in fact, so long as setf works on it.[40]

In general, however, we can understand mutability as a (strange kind of) type constructor: for every type T we provide a type #T, a container for objects of T, that they can be assigned into and extracted from according to an appropriate "temporal"

as a need to be globally consistent in the choice of default. We only mention this somewhat silly second option because the author once worked for someone who swore up and down that it was good engineering practise in that the computer could then never get confused and return something undefined.

[38]The inverted commas stem from the fact that the author is by disposition unclear on what the intuition behind "time" is supposed to be.

[39]Or, as the romantically minded—but fumble-fingered—would have it, change their vales.

[40]An observation that is no longer circular since [Ste90] standardised—after CLOS—the defunability of (setf <*name*>).

discipline. For our present theory it suffices that:-

Values are defined: Whenever a mutable object is visible[41] it has a defined value,

Assignment is atomic: When an assignment is requested, there is no moment "while" it is happening, at which the container doesn't hold a well-formed value;[42] and

Not too much depends: Structural phenomena—like those determined by dependent types (as described below) are protected from the vagaries of mutating mutables.

One case of the constant definèdness condition in particular causes a certain amount of difficulty: some languages (notably Pascal and Modula-2) strictly separate the declaration of mutables from their initialisation. Thus there is an (arbitrarily long) period during which the programme has specified no value for them: they are then *declared* but *undefined*. The most straightforward solution to this problem is to design the language so that there is no such gap (this is the solution we have chosen for *Exemplar*); but equivalently we might have the compiler provide an initial value, possibly recovering some efficiency by forestalling object creation until its first explicit initialisation when (as is usually the case) this is possible.

Visible undefinèdness is only pernicious from the implementation's perspective (though in languages and in cases where it is not mandatory it is often indicative of a programming error) when the values of a type do not fully cover the representation space provided for it: for there is no knowing what the effects of manipulating a non-value "of" a type might be. Pointers especially are normally sparse even in their machine-level representation, and since they play a distinguished and very "semantic" rôle in the structure of data representations must be accorded special care.

[41]And in a garbage collected system this includes any moment at which a garbage collection can occur; it is not restricted to explicit references in the source programme.

[42]This may require nontrivial source language support in some circumstances, as, *e.g.*, the *monitors* of Concurrent Euclid.

Given the provisos above, mutables are *in any static snapshot* much like conventional values, and do not seem to require any special formal mechanics in the type system beyond exclusion from some rôles.[43]

In machine representation, of course, T and #T are (in naïve implementation) all but identical: the *mutable* is just the place where the actual data underlying an object are stored.[44]

The properties of mutables when taken in the presence of a subtyping relation are worthy of brief note. Since the *content* of a mutable is a conventional value, for purposes of *reading* we could permit $\#T \sqsubseteq \#U$ whenever $T \sqsubseteq U$. As an "lvalue,"[45] however, a mutable should only accept a new value that is of a *sub*type of its own type; which is to say that for purposes of *writing*, $\#T \sqsubseteq \#U$ when $T \sqsupseteq U$. In consequence, unless the abstract quality of "mutability" is separated into distinct properties of readability and writability, the # operator must be opaque to subtyping.[46]

Pointers: Another common computational type, and one which has significantly greater technical implications (in fact it is the root cause of the garbage collection problem, which provided the initial motivation for the present work) is the **pointer**. Here, the standard model is that to each type T there corresponds a pointer type %T,[47]

[43]Meanwhile, the provision of an appropriate temporal model seems to require all of denotational semantics· but that is a distinct problem, unless it is wished to weaken our conditions.

[44]Note that $\#T \not\preceq \#\#T$, though the latter is both meaningful and possible: assignment to the value of a variable of type ##T will not affect the value of the (original) value—an effect that is perhaps best accomplished by implementing mutable mutable types as pointers. This motivates Algol 68's collapse of mutables and references into the single notion *ref*, while our concern (in this thesis) with underlying representations prompts us to maintain the distinction.

[45]An **lvalue** is a value that may appear as the left-hand-side of an assignment; in *Exemplar*, as the first argument of := ;.

[46]There is another, technical, reason why it might be desirable *not* to make mutable types mutually comparable, namely that mutables are instance-sensitive (in that performing an operation on a copy of a mutable value does *not* have the same effect as performing it on the original) and the implementation of type conversions that require rerepresentation is made much harder.

[47]The fact that pointers are inherently—and *uniquely*—typed turns out to be crucial to the construction of **Visitors**. For other types we allow the "same" value to acquire different types (through the assignment of a value of a subtype to a variable of super-type), but we do not permit the conversion of a pointer to some type to a pointer to a supertype of the type (pointer types are, in other words, incomparable in the subtype relation). Theoretically the reason for this is that a change of type *may* always result in a change of representation, and hence a change of "identity," the property

and its interpretation is of an arrow in a graph of accessibility to various "copies" of the underlying T-object—isocopyhood being detectable either by operations especially provided for the purpose, or, if T or some transitive part of T (the definition of "part" being obvious for a product but, as we have seen, somewhat controversial for a function) is mutable, by causal time induction over side effects.

As with mutables, pointers are perhaps most easily understood operationally (they are, after all, a direct reflection of the characteristic operator distinguishing von Neumann machines from Turing machines): the usual implementation is that a pointer is the *address* of the object it makes reference to.

Patterns: At various points in our exposition we have made reference to *patterns:* more or less syntactic entities which are used for eliminating values previously built, and as the left-hand-sides of functions and equations. It is a pity that it is not presently understood how best they can be treated as values with types of their own, though it is amply clear that advantages would derive from solving this problem; perhaps the most difficult aspect of this undertaking is the separation of their name-binding behaviour from their semantics strictly understood. They would have to be provided with scoping rules that are in some sense "the opposite" of those of expressions, but the pinning down of in what sense proves challenging.

Type: Type theories are known in which TYPE is itself a type [Car]; for our present purposes—or rather, for those of section 4.6—it suffices (as, for instance, in Russell [DD85]) that types belong to some type-*like* domain within the language. The underlying representations they are assigned (and unlike some classical type systems it is clear that we *do* assign them representations) are their **Visitors** and any other

which pointer transmission preserves. The important operational consequence of this restriction is that the lattice representing the values that a (possible) pointer potentially refers to is trivial, radically simplifying the computation of reachability as needed for visitation (and in consequence any other fixed-point computation).

data that may be packaged with them to support genericity. We do not attempt here to provide them with any more formal grounding than this, other than to note that these too—like functions, mutables, the constructive reals and other "difficult" objects—have underlying representations that are of necessity perfectly well-behaved structures, and it is only the reflections of these representations back into the semantics that provides scope for "real mathematics" in their analysis.

4.5 Type Evasion

While sophisticated type systems are undoubtedly of value both for the purpose of organising (and maintaining coherence in) programmes, and for providing the compiler with information it needs to do a good job of code generation, there are circumstances in which type discipline does not seem to be on the programmer's side. In order to address this kind of situation, programming languages (and in some cases, programmer ingenuity) provide a veritable panoply of methods for evading the strictures of their type systems.

In this section we address a number of these thorny problems and their classical solutions, with an eye to distinguishing those that are perfectly legitimate operations *within* a strongly typed system, being no more than notational devices to reduce the burden on the programmer, from those that in fact step outside the type system and perform "truly untyped" operations (typically stepping outside of the intensional paradigm and providing extensional access to a datum); and further to identify which of this latter class are necessitated by remediable deficiencies of the language's type system, and which are truly inherent to the programmer's task. The astute reader will have anticipated our objective of establishing that this last class is very small indeed.

4.5.1 Motives

There is quite a range of reasons why a pr·grammer might wish to circumvent the type system of a programming language.

Genericity: In the programming language C, for instance (where argument type checking in function calls is optional, and has only become possible quite recently in the language's evolution—and where reïnterpretation of one integer-like type as another is easily accomplished), one of the main uses of type evasion has been to provide a kind of "poor man's genericity": one of the ways in which C scores over, say, Wirth's languages, is its ability to define (and for the user to define) functions like malloc() (which allocates dynamic memory space by *size*, and whose result is typically cast to whatever type is desired), printf() (which takes an arbitrary list of arguments of arbitrary type, and one format string that describes them and what is to be done with them) and sort() (which is in fact a conventional generic function, accepting arguments of "arbitrary" type—suitably cast to the most generic pointer type available—and the functions needed to manipulate them; and then operating on them as black boxes interpreted by those functional arguments).

All of these functions work because in C all pointers—regardless of referent—are more or less guaranteed to have compatible *representations,* and so "puns" on objects passed by reference are entirely feasible at the machine level.

Lest it be thought that such issues apply only to functions (in which case some form of macro-expansion approach—as is provided in Ada, for example—would be entirely adequate), let us observe that the need arises equally for generic *carrier types*—structures holding objects of arbitrary (though often homogeneous) type. A classic example of such a situation is in fact found among those functions specifically mentioned above: C's sort() takes an array of "generic" pointers[48] to be sorted, as

[48]In pre-ANSI C these are actually of type char *, but modern C compilers provide a type "void *" which is precisely a generic pointer, and represents an explicit embracing of the tactic.

one of its arguments.

This whole argument—that genericity is of prime importance, and type rules cannot be allowed to stand in its way—underlies the support that the so-called "untyped" programming languages (such as LISP and Smalltalk-80, and perhaps even FORTH) receive. In these languages it is possible to disregard the types of objects completely, except as they apply directly to the parts of those objects that are most directly manipulated by the programme: list manipulation code can treat a list as a *list*, regardless of what it might be a list *of*.[49]

Interfacing: An issue related to that of genericity (which can be understood as the problem of dealing with an object whose interface is dynamic relative to the type system of the language) is that of interfacing with objects whose behaviour, while fully static, is not compatible with the programming language's normal rules. There are two very common cases in which this arises: lateral interfaces, where it is necessary, for instance, to pass information between functions written in different programming languages; and downward interfaces between the application and the underlying (virtual or physical) machine.[50]

In either of these instances, since the form of the data structures to be manipulated was not under control of the language implementor, there can be arbitrary skew even between objects of equivalent semantics across such a boundary: the *representations* become—unfortunately—paramount. It can thus prove necessary to circumvent normal type-checking rules in order to access such data at all.

Another potentially difficult problem of interfacing arises when *implementing* facilities for the use of other, higher-level, software. In this case it may be desirable

[49]Smalltalk-80, of course, takes this one step further and, its adherents argue, enables one to manipulate an object without knowing anything about the object itself. This strikes the author as a most *peculiar* idea.

[50]In deference to the usual understanding of "layered" systems we assume that it is the downward, not the upward, hemiïnterface that poses the problem, though this is by no means always the best analysis.

to *create* what is conceptually an opaque type barrier while still permitting the implementation module the direct access that it needs. Since not all languages provide explicit facilities for this task, type evasion may be, in some contexts, the *sine qua non* of this desirable form of opacity.

Missing Operations: Akin to the usual motivation for calling an external function is a different reason to seek to evade the strictures of typing: that the programming language being used simply fails to provide a function necessary to the task at hand. Many languages, for instance, do not provide direct access to the separate exponent and mantissa fields of a floating point number,[51] despite its importance to a number of problems in numerical analysis; another example more common in the author's experience is the need to manipulate individual bits within a pointer, in face of the pointer's nominal opacity.[52] Here, particularly if one type provides more direct access to the underlying machine model than another, the ability to interpret one object as if it had (the "usual" extensional interpretation of) the type of another can make the desired operations possible.

Runtime Efficiency: Perhaps the most frequently cited motive for all dubious programming practise is runtime efficiency.[53] Type evasion can serve this end in both the spatial and temporal domains: spatially, by allowing the same storage to be shared by distinct objects (a potentially significant saving if the structure in question is large or replicated many times), or temporally, by specifying an implementation for an operation that is more direct than would otherwise be possib!~. The common

[51]To take an example belovèd of H. Rubin.

[52]Although the author in fact needed—or believed sie needed—to violate that opacity quite profoundly, there are fairly benign reasons to examine the internal structure of a pointer: such as constructing hash codes, at least in systems where objects' physical address do not spontaneously change.

[53]For some reason a claim of increased efficiency seems to justify almost any programming ill, even in the face of evidence that efficiency is not increased; while arguments to æsthetics, consistency or sound engineering polic, fall on deaf ears. It is beyond the author to understand this fact.

element in these cases is that the programmer knows of an optimisation that cannot be expressed in the source language strictly interpreted, and which the compiler cannot be expected to recognise on the programmer's behalf; type evasion is practised as a method of tweaking the compiler's type representation or code generation strategy.

Development Efficiency: Finally (and once again arguing in favour of unitypic languages), there is the problem of clumsy notation: how much detail is too much may be something of a matter of opinion, but there are certainly cases in which typed programming languages succeed in introducing enough notational overhead into a programme that the effort of coding is significantly increased while the clarity of the resulting source programme is impaired. Programming languages thus frequently provide mechanisms for ignoring the type system at the level of notation, accepting one type where another would strictly be required, or, in the extreme case, allowing the total omission of type information from the source code. The latter forces the language implementation to rederive and itself insert the type information it needs (whether dynamically or statically); while the former requires it to insert various type conversion operations on its own initiative.[54] It should be emphasised that these last "evasions" differ somewhat in kind from those we considered above, in that the language implementation must concern itself with the preservation of the intended semantics of the expressions involved, rather than taking the back seat to the programmer's whim: data are processed "hyperintensionally" and not, as in the preceding, "hypoïntensionally."

[54]The two phenomena are in some ways opposed, since the absence of explicit type information removes information that could be used to generate implicit coërcions; though from a formal point of view, "automatic conversion" is—or perhaps "should be"?—just an implementation technique for subtype-driven genericity.

4.5.2 Costs of evasion

Type evasion has a number of costs, some of them obvious, some of them less so. The most famous of these is the loss of static checking: wherever the type system is defeated, circumvented or tricked, or just looks the other way, an opportunity is passed up for the compiler to check the source programme for consistency, and the scope for human error is increased.[55] Almost as evident is that type-evading code (unless its mode of evasion is clearly sanctioned by the language) will tend to impair software portability: for two compilers for the same language on different machines are rarely bound to employ the same representations for data in each case.

More subtle, perhaps, is the fact that the employment of almost any such technique impairs the intensional *transparency* of the programme: what is an obligation on the compiler to look away while an operation is performed is an obligation on the programmer to think in terms of what the physical machine is doing in evaluating a certain expression, and constitutes a shift in semantic level for anyone trying to understand the programme on its *own* terms, not on the machine's; while an imposition on the programme to "fix" an incompatible data type by inserting conversion code, is just as much an imposition on the human reader (and maintainer) of the code.[56]

On a related theme, it is also true that, just as defeating structured code format with a "judicious" goto is more likely to impair software performance than enhance it, confusing the compiler and causing it to disable optimisation in favour of a literalistic, blow-by-blow translation of the source, so presenting the compiler with a "type pun" will in general force it to retreat to a more transparent treatment of storage structure

[55]Advocates of dynamic typing (our *unitypicity*) sometimes argue that static checking is unnecessary in the presence of adequate support for interactive debugging. Clearly, however, this argument is specious, for while there it is a valid point that only oft-executed code needs to be tuned for speed, it is hardly the case that only frequently used execution paths need be correct!

[56]The author in fact favours automatic conversions *as part of* a rich type system that provides more substance to the practise than just minimising keyboarding effort: in mathematics a good *abus de notation* is one in which blurring a semantic distinction increases the clarity of the exposition, and so it should be with software.

than might otherwise be necessary: at the bluntest, it will be seen that an object declared as one type and then interpreted as another "behind the compiler's back" *must literally exist in memory,* even when a less "clever" description of the task to be performed would have allowed for it (or its parts) to be dissociated, loaded into registers, or folded into the object code in some manner (as discussed in section 4.2.6).

Finally, there are a number of language facilities possible which rely on inviolable typing to work. Among them is garbage collection; but in general any operation that relies on the global integrity of data structures is likely to be sensitive to type errors, and anything relying on *asynchronous* integrity is absolutely dependent on type correctness, as are such manipulations as formal verification which require *static* typèdness. In each of these cases the very existence of a loophole compromises the facility as a whole, and the entire debate is elevated to the level of language design criteria.

4.5.3 Analysis

Of the various phenomena we have categorised as type evasion, some quite clearly have the status of legitimate operations *within* the type system: the surface notation may play fast and loose with the types of expressions but the language semantics is entirely clear on the meaning of the operation, and specifies that meaning in terms of well-typed (lower-level) operations. In this class of type-legitimate operations we find both semantically-motivated static coërcions (as, perhaps, when an integer is implicitly converted into a floating point number, or a string may be written in a context where an array of characters is "expected") and the dynamic automatic conversions and "typeless" features that are common in such languages as LISP. In all these cases, certain operations are unambiguously required by the language in the interpretation of certain expressions, and the fact that the notation used to request them is null changes nothing—except possibly the clarity of the notation, something that must be weighed carefully by the language designer.[57] In these operations, the intent is that intension be preserved.

A second class of operations are those that we may consider fully legitimisable: in the use of C pointers as generic object handles or the exploitation of unchecked argument lists to provide more flexible operations, nothing is being done other than the working around of a semantic shortcoming in the language's type system. A more powerful and more flexible typing scheme—in particular, one providing for polymorphism or dependent typing—would render such gyrations entirely unnecessary.[58]

Next comes a class of operations that are legitimated by fiat of the programming language: they do not make much sense from a type theoretical perspective (and will have an inherent tendency to compromise the possibilities for positive benefit from the type system, as noted above), but the language provides the operations in an "official" way and with appropriate disclaimers about programme behaviour in their presence, and the programmer is trusted to use them well.

The features in this class are in general those intended specifically for performing low-level tasks like interfacing and storage management: there is a sense in which they are better interpreted as providing fairly normal access to the underlying machine representations (in some slightly extended version of the language) and then arranging for these low level objects to be reflected up to the "normal" user level.[59] In whatever guise, these operations are somewhat dangerous and liable to confuse the compiler enough to induce hypercautious translation; but they certainly have valid application—if not so much as is often supposed—in systems programming and in the construction of the libraries that support the basic vocabulary of operations that

[57]Witness Algol 68, of which it can easily be said that there are a few too many such implicit conversions.

[58]The reader unfamiliar with such type systems will find a discussion in the next section of this chapter—though in this thesis we use explicit parametric typing rather than unconstrained polymorphism, on the grounds that the former provides finer semantic control, while the latter must in any case be implemented in terms of it.

[59]A number of programming languages, including Mary2 employ this model explicitly.

normal language users see.

The remaining operations, essentially those not sanctioned by the language but employed because the programmer has discovered that they "work," are probably not found within the normal toolkit of the wise programmer, and the programming language designer or implementor might consider removing them altogether from existence. They operate by the exploitation of operations whose validity the compiler does not or cannot check: mismatched or underspecified function call interfaces, unchecked "variant records," creative pointer arithmetic or array indexing, and so forth.[60] Such operations not only carry with them all the deleterious effects of the previous class, but as "unofficial" loopholes in the language they are entirely likely to provoke unexpected results even when the compiler has noted their presence and taken steps to defend itself: they step clearly out of the bounds of what the programming system can reasonably cope with, and leave the programme at the mercy of the raw underlying hardware. In a sense, these operations *have* no semantics, and even system implementors might be well advised to find another way.

As a final comment, let us note that genericity is not alone in being a need that can be filled through more sophisticated language technology: matters of both efficiency and external interface might well be addressed by a compiler architecture that permitted the programmer more insight into and more control over the internal transformations that are traditionally seen to constitute optimisation. This might be accomplished at the source level through extended declarations and annotations (as, for instance, C's register declaration and field trims); but in the interests of portability, clear source code, and not tying the implementation's figurative hands, it is possible that—at least in a system that places less stress on the concept of the integral, printable, source file—a more appropriate level for this kind of work could

[60]Not, it should be noted, that *any* of these isn't sanctioned in some way or another by *some* language—though in the process they shade into the previous class of dangerous but explicitly legitimated practise.

be found, one talking more directly to the compiler's middle- and back- "ends."

4.6 Dependent Types

While we have been arguing for the desirability of a static typing scheme, it should be clear (if only from the discussion in the preceding section) that a type system in which each expression can be assigned a unique underlying type *regardless of its context* would be overconstrained in many applications.[61]

Fortunately, there is a broad middle ground available in the form of **dependent types**. Strictly speaking, an expression has a dependent type if its type is itself an expression containing (in general non-manifest) free variables, though the term is used more loosely to refer to types composed from these. Care must, of course, be taken, in specifying where the variables free in such a type are ultimately bound.[62]

Ultimately, dependent types (in this loose sense) may differ in both their compositional structure and the binding points of their variables, though for historical and notational reasons these often covary. This section is organised so as to examine a number of the classical dependent types separately, though the notation employed is clearly compatible with the full generality suggested in our definition (modulo a particular set of scoping rules).

4.6.1 Dependent product

The most familiar of the dependent types is without a doubt the *sum* or *discriminated union*, which we analyze as a (finite) dependent product.

The **sum** of types T and U, T + U, is a type whose members may derive from

[61]The situation is in fact very similar to that of pure functional programming: the argument for side-effect free code should certainly not be taken as an argument against parameterisation! In fact, we shall see that one possible condition for the conceptual well-formèdness of a dependent type is precisely the functional "purity" requirement: that the information needed to freeze the dependency is safe from potential assignment.

[62]In particular, all the usual arguments against dynamic scoping for the free variables of functions apply with redoubled force for types: a type *must* have an assignable meaning that does not change with its context, else what of the objects pertaining to it?

either of the two argument types, and whose provenance is recorded. Just as records generalised product types to an arbitrary number of operands, the **discriminated union** generalises the sum; and, type evasion aside, this generalised form is, for example, the only dependent type provided by Pascal, where it appears as the record...case...:...of...end construct. Likewise, in our preferred analysis, the only type that LISP provides at all is a member of this class (it is the discriminated grand union of atoms, cons cells, integers, strings, ...).

If we choose to write a (particular) discriminated union thus[63]:

```
( {tapir, cuttlefish, stone} kind
, if kind
  is tapir      then Distance noseLength
  is cuttlefish then Distance meanTentacleLength
  is stone      then N cucumberFrameMishapCount
  end
)
```

it becomes clear that the discriminated union is simply a pair, a two-field record structure in which the type of the second field is an enumerated function of the value of the first. (Equivalently, and more in the tradition of the block-structured languages—though perhaps not of their type systems—we generalise multifield record constructions to permit arbitrary dependencies of field types on the values of preceding non-mutable fields[64]—this was in fact the paradigm of chapter 3.) We can generalise this straightforwardly to a dependent version of the Cartesian product operator.

Building on the usual Cartesian product notation, in which the type of pairs with elements drawn from T and U, respectively, is written $T \times U$, and employing

[63]Note that unlike the very similar InterestingThing type employed in chapter 3, kind is here a separate field, not embedded in the condition. The semantics is in fact the same; but the objective in this chapter is to focus on the logical structure of the type rather than its relationship to Algol-like control structures.

[64]Note that access to fields (and variables) in outer scopes is subject to precisely the same issues as access to outer-scope variables in functions: provision must in all cases be made for the value of the outer variable to be available. In fact, as has been mentioned elsewhere, a variable frame is precisely a stack-allocated structure, and the situations are for our present purposes isomorphic. In the presence of a garbage collector, access to outer scope objects must be possible at all times, and not just at points where textual references occur. Fortunately this need not be the concern of the programmer, though later we will introduce restrictions on the complexity of type expressions.

the substitution notation $e\overset{x}{a}$ to mean e with all free occurrences of x replaced by a (under suitable renaming of variables to avoid name capture), we write the *dependent* product of T and U (where U is now an expression possibly containing free x in T) $\mathsf{T} \dot{\times}\ t.\mathsf{U}\overset{x}{t}$. The "." is intended to recall the variable introduction dot of λ-notation, though for us it is here part of the dependent product operator itself.

Thus we can define $\mathsf{T} + \mathsf{U}$ to be $\mathbf{2} \dot{\times}\ i.\mathrm{if}_{i=1}\mathsf{T}\ \ \mathsf{U}$ (where $\mathrm{if}_p xy$ is x if p, and y otherwise).

This generalisation both permits us to define "counted" strings (for which no enumerated union is adequate), for instance, as:

```
( N length
, Vector[Character, length] data
)
```

and allows us to introduce more arbitrary (and potentially more practical) patterns of dependency.

Finally, if the language provides some version of a type of types, it is possible to define objects such as those found in Smalltalk-80, which share a common type even though all variant representations are not statically known (types being dynamically computable); which is to say that we can define a form of **universal type**, formally equivalent to the single type shared by all LISP objects:

```
( TYPE Type
, Type value
)
```

Despite all of the preceding it should be stressed that the system discussed here *cannot* represent the C union construction, although it clearly provides for an alternation of types. The reason for this is that in C the discriminating variable (the tag, *i.e.*) is bound only at the point of *use* of the field, rather than in one of the other fields of some containing structure. In fact, since points of use are not statically associated with objects, this is one of several respects in which C fails to be strongly typed.

On the other hand, the Algol 68 union is (isomorphic to) a very simple instance of the class of dependent products, where the left argument of the $\grave{\times}$ operator is restricted to be a particular finite enumeration of names of types, and the right argument the canonical mapping from names to types.

4.6.2 Dependent functions

Once dependent types have been admitted to a programming language, it is essential that it be possible to write functions that manipulate them. So long as they are restricted to a finitely enumerated set of variants this does not require any conceptual extension to the naïve arrangement in which each expression has a closed type, since a restriction (for example) of the form that variant fields are only visible within case statements dispatching on the discriminant field would suffice.[65] In the presence of the kind of unrestricted dependency described in the preceding section, however, it is clear that some capacity must be provided to manipulate data of essentially *arbitrary* type.[66]

One framework for doing this is that of purely generic functions, ones that are able to operate over arguments of completely unspecified type, as provided, for instance, in ML (a familiar example from the older expression languages is the usual *if*...*then*...*else*...*end* operation, if it be understood as a true function with conditional branches enthunked). Aside from the technical observation that this source construct in fact begs the implementational question and must underlyingly be handled in the manner of the dependent maps we shall describe shortly, this mechanism will be seen to be too weak to fill our needs, since in order to be generic over *all* types a function must presumably make *no* assumptions about how its arguments can be manipulated, and is thus not in a position to manipulate them in other than a com-

[65]As in Pascal and Mary2. In fact it is presumably the intent of the very restricted variant record structure in Pascal to make this syntactically checkable.

[66]Actually, in the presence of type *comparison*, one can get by, howeverso clumsily, with the universal union structure—in fact, LISP, of course, does. We might, however, not wish to admit type comparison, for any number of technical reasons.

binatorial manner (which, while computationally complete, is less than satisfactory from a software engineering viewpoint).

A significant generalisation of this approach is found in languages like Haskell, where generic types can (optionally) be restricted to support named operations.[67] This extends genericity to cover the objected-oriented concept of type: parameters are allowed to belong to any type nominally satisfying a particular input-output protocol; but it does so by sacrificing the ability to manipulate (uniformly, at least) objects of types that differ in this broader (signature-based) sense.[68]

The problem can be addressed straightforwardly through **dependent function types** directly analogous to the dependent products described in the previous section. We can introduce, for types T and $\mathsf{U}\overset{x}{x}$, a type $\mathsf{T} \overset{\cdot}{\rightarrow} t.\mathsf{U}\overset{x}{t}$ of functions returning values of type $\mathsf{U}\overset{x}{t}$, depending on the *value* of the argument t (the notation is strictly analogous to that introduced for dependent products, in section 4.6.1, above). This is at its most obviously useful in the case of functions returning fairly uniform structures depending on some parameter (as values in some range or arrays of variable dimension); but, in the presence of a type of types, it suffices to represent both freely generic (strongly typed) functions and operations over "carrier types" where argument and result types are both parameterised by some type which is passed in as (or perhaps computed from, though there are practical reasons to place restrictions on this, as we shall see) an earlier argument.

At first blush it might appear that the ability to make the value *returned* by a

[67]Actually, as far as is known to the author, all such approaches to date appear to be seriously flawed in terms of the scoping rules they are forced to employ in order to coordinate the supported operations with those available at the callpoint; they must either be severely restricted or ultimately amount to a dynamic scoping scheme. A better solution would probably be to associate the operations dynamically with the statically located type itself, yielding a structure that is at once object-oriented and lexically scoped: types are then values with distributedly declared structure and an implicit substructuring relation.

[68]The signature-restricted generic mechanism is clearly different from the case of Ada generics, which, while similar in style, are designed to be implementable by macro expansion; although they are *syntactically* generic they are in fact only capable of being instantiated over a finite collection of (super-) types (and it was apparently several years before the first true shared-code implementation appeared).

function depend on its argument is insufficient to our goals, since it as important that the *arguments* be generic as it is that that the result so be. This difficulty can be circumvented straightforwardly by treating an entire argument list as a single cartesian product, or, if a function of multiple arguments be understood to be implicitly curried, by making the value returned from application of the early arguments be a function now specialised to particular argument and result types.[69]

(It should perhaps further be pointed out that the value of dependent type returned by such a function is no more "orphaned" or untypable than is the right member of a dependent pair, since the arguments to a function (insofar as it is applied) are *a fortiori* available at its callpoint (and sufficiently evaluated to determine the type of the result whenever the result's value is required). In fact, the type of the field reference operator, considered as a function from field tags to values, itself has a dependent map type. Making the field reference operator typable is a major motivation for dependent product types in its own right.)

4.6.3 Functional types

Throughout this section we have been discussing dependent *types* as if types in fact they were. In actuality, these formal entities are no more types than $n + 7$ is a number: they are expressions over the domain of types, they evaluate to particular types, but they—in light of their particular identifying characteristic of having free variables—are not even objects in the language.

The formal notations we have been using for dependent products and dependent maps, however, have been chosen for their suggestivity of the classical λ-abstraction notation:[70] we write $\mathsf{T} \xrightarrow{\cdot} t.\mathsf{U}$ in recognition of the relationship between this construc-

[69]That these two solutions are as practically as theoretically equivalent becomes very clear on examination of the current implementation of Lazy ML.

[70]The idea behind λ-calculus is this: that if functions are true values, they can be manipulated straightforwardly and to advantage; and that this can be achieved by the reïnterpretation of the expression $f : x \mapsto e \overset{\blacksquare}{x}$ as being an *equality* (with nonstandard equals-sign ':'), relating f to a *function* "$x \mapsto e$"—the whole being re-notated (for historical reasons) $f = \lambda x.e$.

tion and $\lambda t.u$; we take '.' to denote (textual) abstraction, designation and binding of a variable, with $\overset{.}{\rightarrow}$ and λ being operators on these formal entities.

Let us, then, introduce a new such abstraction-legitimating operator, Λ, which differs from λ in that it builds abstractions over *types* rather than *values*.[71] We can use it to build dependent types as objects in their own right. In fact, objects of this kind are already familiar: they are the type *constructors* like Vector and #, objects that accept parameters and evaluate to types.

The difference, of course, between these **functional types** and the dependent types we addressed earlier in this section is just that the dependency is made explicit in an argument list, rather than by default in the *failure* of a type expression to bind some variable (above all others this is the reason that functional abstraction is a powerful engineering tool, and it naturally applies to other entities besides values).

4.6.4 Other dependency patterns

The varieties of dependent type just described would seem to suffice for most practical purposes, but it seems worth mentioning a few of the more pathological structures from extant programming languages which can be handled (or at least, have clear analogues) within the framework of this thesis. Since we are dealing with structures from languages with weaker type systems than those which we propose, we will find ourselves referring ever more frequently to physical representations. It will become apparent, furthermore, on reading this section, that here we reach and, indeed, exceed the limit of the descriptive capacities of *Exemplar*; we consider the development of adequate source-level notations in which to describe these more intricate data structures to be an important area for future research. As will be discussed below, however, the *techniques* of this thesis apply straightforwardly in most of these cases: they are far more easily processed than defined (a fact which should not surprise us,

[71]It is a function of the particular type theory whether λ and Λ are identical. Systems exist in which they are, but it is a weaker assumption that they are distinct. *Exemplar* does not assume their identity, although our *implementations* of the two operators are very similar and tightly interwoven.

given that all are practical examples). Furthermore, although we are concerned here with objects that are generally conceptualised in terms of their underlying structures, it will be seen that we *are* able to provide reconstructions—as *Exemplar types*—that capture most of the characteristic properties of these objects.

C Strings: One of the superficially less well motivated aspects of conventional data structure lore is the *guard value:* a value that "cannot occur in practise" is selected from a domain for use as a delimiter or sentinel in some data structure, in order to reduce the complexity of boundary checking code and the amount of state the programme must maintain. Unfortunately, it tends to accomplish this *en dissimulant* the relationship between the data and their structure, increasing the likelihood of programming error; while the assumption that any value of a type will in fact not occur is considerably less justified than one might hope.[72]

One language in which the guard value plays a prominent and integral rôle is C, where 0—besides being an integral value—plays a special part both as a "pointer" (in which context it is understood to be an acceptable member of any pointer type, though it specifically fails of reference) and as an element of a character string (which, despite its otherwise being an acceptable character, it serves to delimit).[73] Thus, in the C string (which is essentially no other than a pointer to a 0-terminated array of characters), we find three separate pathologies from the perspective of strong typing: the pointer that may not point, the character that may not figure (or worse, whose characterhood simply cannot be decided), and an array-like structure whose bounds are determined by the very data it contains. Nonetheless we can provide a secure reconstruction of this type within the present framework, at the cost, admittedly, of

[72]While no *formal* type evasion is involved, it will be seen that this is another instance of the programmer's concealing the intended interpretation of the data from the compiler.

[73]The pointer value is commonly written NULL and the character '\0'. The author's reading of the defining standard ([ansi89]), however, is that they are intended to be truly identical: it is completely acceptable to assign, say, 9 − 9 to a pointer—though the same clearly does not go for any other integral value, which must be cast, and *caveat castor*.

requiring a great deal of compiler sophistication to recover the full efficiency of the very straightforward C equivalent.

In terms of our language *Exemplar*, the declaration of such an object would run something like this:[74]

```
  TYPE CChar : #Character{nil}
; TYPE NullTerminatedString : for N i while CChar [i] :: c; c ≠nil end
; TYPE CString : #(%NullTerminatedString){nil}
; ...
```

Here, the uncertain nature of the character and pointer values themselves is captured by the extension-element mechanism (as described in section 2.4 and below). The more interesting point for our present purposes, however, is the appearance of a *while*-loop in the declaration of NullTerminatedString. This allows us to describe the kind of serial dependency between elements of the string that is characteristic of such an ensentinelled representation. It does this, in our notation at least, by mirroring the structure of the code that processes the elements of the string in order—and of the **Visitor** that the compiler would generate from the declaration.

The dependency pattern thus introduced, where the *presence* of each element is a function of the *value* of its predecessor (in an unbounded fashion) is in fact a peculiarity and the cause of some difficulty. In a strongly typed language that wished to provide such types, bounds checking would have to be performed by scanning

[74]This declaration actually fails to capture two interesting properties of C strings: first, that it is possible to remove a prefix from a string while maintaining sharing, simply by incrementing the pointer; and second, that since they are invariably maintained within buffers, it may be possible to perform meaningful manipulations on strings by overwriting characters within them with null values and vice versa. In fact, there is nothing in the nature of the present work to prohibit the exploitation of either of these properties in user code, though it is hard to see how they could be introduced into the source language in a clean manner. The difficulty is that the validity of pointer arithmetic here depends on the ability of the compiler to perform minor theorem proving (since the fact that the pointer still points to an object of the same type after incrementation—assuming it has not been stepped past a null character, which must be checked—rests on a nontrivial induction); while the correctness of splitting and merging CStrings depends on their being located within a buffer,

```
for N i to bufferSize do CChar [i] end
```

and we have not provided any mechanism in the *Exemplar* of this thesis to describe such multiply interpreted objects.

the string to locate its end (or at least, to demonstrate that its end occurred before the element indexed) before each reference (though an optimiser could easily remove most such operations from any given block). The ensuing loss of efficiency is fairly great, but is perhaps justified by the observation that a significant proportion of the notorious security problems of the Unix operating system stem from problems in the management of null-terminated strings.

Algol 68 Arrays: One interesting data structure provided by the venerable Algol 68 is its particular rendition of the familiar array. Arrays in Algol 68 have, as in C, more the nature of pointers than values—which is to say, that the values associated with the names of arrays are in fact references to the elements of the array, rather than the value-sequences themselves (this distinction becomes clear when they are used as arguments to functions, for instance: although in both these languages parameters are passed by value, transmitting an array results in the elements of the array being shared while the value of the array object—the reference—is copied). Unlike C, bowever, where the characterisation of an array as a simple pointer to its elements verges on literal truth—the only qualification being that the *type* of a C array can (in some limited circumstances) bear information about the static size of the overall array as well as of the elements and the pointer—the Algol 68 array carries full and dynamic dimensioning information in the form of a descriptor whose format depends on the array's rank.

This representation of an array as a complex reference to a complex object permits much richer semantics than other models of arrays; in particular, it is possible—through the manipulation of the descriptor—to provide efficient implementations of a number of bulk operations over the entire array. Among these, for instance, are the ability to extract slices through multidimensional arrays at certain indices, and, more generally, the ability to trim dimensions (deleting leading and trailing elements along one or more axes) and rebias indices.[75]

The fact that these operations are implemented on descriptors and not on elements has implications beyond those of efficiency. From the user's perspective, it provides a further respect in which arrays are visibly reference-like and not value-like: the results of these operations end up referring to the *same* values as do the arguments, a fact of potential algorithmic use (particularly since copying can be forced when desired). Far more significant for our purposes is the structure implied for, or induced on, the data components—the actual *arrays*—of arrays.

On a basic level it should be apparent that the implementation of these arrays involves iterated use of functional types. As with conventional fixed arrays, Algol68Array (as described below) is actually a function from an element type and a set of bounds (in this case, just the rank of the array, since the dimensions are potentially variable and thus not part of the type of the array *per se*); but the element type and the dimensions stored in the descriptor are further passed through to the underlying array of values. In *Exemplar* we are able to describe this arrangement explicitly (and the availability of Visitors, of course, follows directly); we might write something like

[75]These operations amount to minor changes in the index types as interpreted as integral subranges—in local scopes they could typically be optimised so as to involve no runtime cost at all. One could, however, further imagine operations picking out triangular or trapezoidal subarrays, performing in-place transposes, slicing out diagonals, or various kinds of variable-step or irregular slicing (akin, perhaps, to APL's selection functions). Vector-register supercomputers often provide instructions to implement this kind of operation directly. The enthusiastic reader is invited to bear these generalisations in mind as we examine the complexities arising from even the standard facilities in the remainder of this section.

this:[76]

```
TYPE :< Algol68Array @ (TYPE t, N dim) >:
  with N n from 1
    for N i to dimension
    then (%Vector[T, n]) values
  as
    N (lowerBound[i]) :: l
  ; N (upperBound[i]) :: u
  ; n + u * l
  end
; ...
```

When considered in more detail, these objects turn out to have some unusual properties, at least as regards their physical representation in memory. If we examine the pattern of multiple referencing that arises as these arrays are sliced (something that *Exemplar* as we have presented it is unfortunately not suited to represent di-

[76]Actually, the simple version presented does not take care of the mechanics of indexing: the user is still required to index the field "values" in arcane fashion. One solution to this problem would be to use an auxiliary function for indexing. Assuming, however, that facilities are available for vector arithmetic and preörder traversal of vector spaces, we can handle indexing *within* the type specification in the following fashion (though it would require an impressive compiler indeed to implement this with full efficiency):

```
TYPE :< Algol68Array @ (TYPE T, N dim) >:
  Vector[N, dim] lowerBound
, Vector[N, dim] upperBound
, (%Vector[T, reduce[1, (*), lowerBound - upperBound]]) _ :: v
; Vector[N, dim] stride :
    [ with N x from 1
        for N i to dim
        do x
      as
        note lowerBound[i] ≤ upperBound[i]
        in upperBound[i] − lowerBound[i] * x
        end
      end
    ]
; for Vector[N, dim] i from lowerBound to upperBound
  do T [i] : v[i − lowerBound * stride]
  end
; ...
```

rectly) we see that the (simple) array object referenced above by the field values is often not held fully in common with other Algol 68 arrays seeing (some of) the same values; in fact, once a computation has progressed through several slicing and trimming operations, no single Algol 68 array that is still in use may reference all the elements that were originally introduced in any given block. Furthermore the patterns of elements that are still referenced may be somewhat irregular (a phenomenon that will be of relevance in chapter 7, where we discuss the application of **Visitors** to garbage collection).

Let us turn first to the simplest case, in which a multidimensional array is sliced so as to yield an array of lower dimension, and where it is the fastest-varying coördinates that are uncollapsed. The result of this operation will be a new array descriptor, the elements relevant to which will be a contiguous subrange of those in the original underlying array, and disjoint with any other set of elements arrived at through a similar operation. A very slightly more complex pattern arises if we take such (underlyingly) contiguous arrays and trim their slowest-changing index; this will result in new and still contiguous arrays, but now with the property that they may overlap, partially or wholly.

If instead we examine slices along higher-indexed axes, or trims along lower-indexed ones, we find that the sets of elements accessible through a *single* array descriptor form a *dis*contiguous set: there will be a regular alternation of accessible and inaccessible ranges of elements.

Consider finally the (still far from maximally co...plex) collection of elements we obtain from the union of two (untrimmed) slices taken along disjoint sets of axes. (They will not constitute a single array, but might well both be accessible at a single point in the programme.) They will have a common element (more generally, a common subarray) at the point where the specified coördinates of both slices are satisfied; and the other accessible elements will lead off from that point in both directions, in a regular manner. The overall pattern will be cruciform in the original coördinate space, since the periods of the included elements contributed by the slices will be different.

This kind of complex pattern of accessible substructure is essential to the Algol 68 array (at least relative to von Neumann computer technology) and not merely an artifact of our particular approach, since the patterns of sharing of elements are a crucial part of the semantics of the language, and greatly restricts the possibilities for placement (or motion) of the elements involved. (There is a sense in which these structures are practically implementable only because they are computed analytically by the dissection of extant, nonoverlapping arrays; rather than synthetically, by desired pattern of alias.[77])

Other structure restrictions: The most interesting characteristic of the complex arrays discussed in the last section is the way in which the various *access paths* to the (presumed) regular, linear, underlying array interact to produce a patchy pattern of accessibility. Recalling the straightforward relationship between arrays and structures, it should not be surprising that a parallel phenomenon arises in product types, in some implementations of some languages.

The most straightforward such case is when a language provides for the construction of references to substructures of objects: it is easy to see how a succession of such operations can result in various *parts* of the same objects being visible from different places; and the best implementations of (object) *inheritance* have the same effect (though without an explicit record of the fact in the source code: the compiler initiates such sharing silently). In the most extreme case [DD85] we find proposals for bidirectional structure layout (fields are allocated at both positive and negative

[77]In fact, this suggests a very practical, if suboptimal, technique, whereby an array descriptor contains not only a programmer-visible description of the elements of the array described, but also invisible fields detailing the array from which the present array was ultimately derived. In a tagged architecture this might indeed be the only acceptable option.

offsets from the base pointer, that is), where objects may (in complex cases) further exhibit "holes," areas of memory which they surround but do not contain (but which some other object in the inheritance graph is presumably using in its objects, for its purposes).

Such almost completely arbitrary patterns of storage location do not necessarily pertain to functional types (which is to say that it is not necessarily the static type of an object that resolves its structure); it is equally plausible that such a type be a dependent product, with one or more control fields serving to determine which other fields are present, and their types. Worst of all, when this is taken in conjunction with two-dimensional structure layout, it results in physical object representations that are *not* resolvable in a single left-to-right pass (and thus this is the first structure we have seen that requires an extension to the **Visitor** mechanism, albeit a comparatively trivial one to implement: the **Visitor** must now be authorised to perform pointer arithmetic beyond mere rounding[78]).

A Strange Thing: As mentioned in footnote 15 of section 3.1.2, there is one additional peculiar case that can arise in systems with inheritance or automatic conversion: two types with different *and incompatible* dependencies between their fields may have non-empty type intersection under certain rules. For consider the types:

(N a, N b, Vector[N, a])

and

(N a, N b, Vector[N, b])

When a = b these structures are, as it were, *coincidentally equivalent.* At least in the presence of a distinguished equality relation, this could form the basis of a system

[78]Conventional sum types when *contained* in other structures can also result in representations with holes when the alternatives are of differing lengths; this, too, must be handled either with pointer arithmetic or by permitting the visitation of "nothing"; in that case, however, the problem can be avoided by transformation of the data structure without (in most cases) crippling loss of efficiency. The same is not true of the bidirectional structures which already serve to address a severe performance bottleneck.

of type *intersection* whose ramifications are sure to be complex—and which must remain a topic for further research.

Hash Tables[79]: There do exist practical (and desirable) data structures in which the pattern of reference and determination between the various components is, from the perspective of low-level structures like the **Visitor**, completely opaque. One that concerned the author for some time and influenced the early work on **Visitors** (and on generic garbage collection in general) in the direction of increased support for *ad hoc*kery, is a particular form of hash table[80] contrived by the author for the LISP dialect Lithp [BS85]. The interesting feature of Lithp tables is their storage retention behaviour: they permit arbitrary associations between LISP objects on the basis of address (in COMMON LISPese, they are :test #'eq tables), but the retention of table entries through a garbage collection depends on the accessibility of the *keys* as well as of the table itself.[81] While it is clearly practical for the **Visitor** to pay attention to only the (quite straightforward) physical representation of these objects (thereby failing to capture the subtlety about accessibility of table items), the only method we have yet found of implementing the intended retention semantics of this structure does involve global reachability analysis of the heap, and thus special treatment under visitation. This structure, by virtue of its global-minded semantics, therefore marks one of the limits of the methods here presented.

[79]The reader unfamiliar with garbage collection technology would perhaps be well advised to postpone the reading of this section until *after* chapter 7, since the particular technicality here considered is intimately bound up with the garbage collection problem itself.

[80]The **hash table** is a particular implementation technology for functions from some (arbitrary) type T to #U{_}, that works by doing numerology on the particular arguments, t in T. The most familiar application of hashing is the *horoscope* (in which T is typically DateOfBirth and U is SuitablyVaguePrediction), with the symbol tables of compilation lore a distant second.

[81]This is correct and expected behaviour for such an associative structure if and only if there is no maphash-like operation that dumps the entire table. The absence of such an operation was a deliberate decision made on the basis of software engineering principles; in particular we saw them as a structured alternative to the conventional LISP *property list*, rather than as a generalisation of sequences.

The Worst Case: One possibly useful (if only as a space optimisation) data structure that we have come across has even worse characteristics than the hash tables just described: there is a technique (used at one point by DEC) in which two-way linked lists are represented with *single word* link fields, containing the bitwise exclusive-or of the component pointers. References into the list are, compensatorily, not one but two words wide: they hold the addresses of *adjacent pairs* of list entries. The original up- and down- links of the list can be reconstructed by exclusive-oring the address of *one* of a cell's neighbours into its link field; the result is the address of its neighbour in the *other* direction. This structure is, in itself, not a problem: it amounts to no more than a mildly nonstandard pointer representation (and thus a minor special case, techniques for normalising which on the fly have been developed).

Consider, however, the consequence of storing the two pointers that constitute the list handle *separately*. Reconstruction of the contents of the list now requires unbounded computation: it is not in general possible to determine *which* pointers will eventually come together to form an indexing pair without running the programme experimentally.

Even in principle, there is nothing that can be done to establish the coherence of a particular such data structure, and as it is the best example the author has yet discovered of a data structure (as opposed to a programme: there are many subtle graph manipulation algorithms that rely crucially on properties not amenable to analysis by the runtime system, for instance) which challenges the notion of necessary representational typèdness—though, to be sure, it is unclear that such a structure is of any practical use, and one can coherently recommend against its use.[82]

[82]The technique generalises to the use of arbitrarily *encrypted* pointers, which is one of the methods suggested for the implementation of *capabilities* (particularly in the absence of secure address space protection hardware or the presence of insecure processing nodes [TvRvS+90]). Such provision of security *from* the compiler is beyond the scope of this thesis, as is the entire problem of distributed garbage collection, raised by the use of capabilities of almost any description.

4.7 Mutable Object Format

Thus far we have prohibited the dependence of a type on values that are either impure or naughty (though they may naturally have these properties themselves by virtue of mutable non-control fields). Such dependencies are not, however, un-heard-of in the real world, and this motivates some minor relaxation of the restriction (though it will turn out to be in some sense non-substantive). Two fairly typical examples of this kind of dependency of types on mutable objects are the classical Pascal variant record, and the extension to Pascal provided by Turbo Pascal strings.

Variant Records: In the case of the variant record, the presence of the variant fields depends on the actual value of the discriminant, which can be changed dynamically over the lifetime of the object; changing the value of the discriminant field causes the original variant fields to be replaced by (undefined) instances of the fields associated with the new value. (In practice the bitpattern in the storage underlying the variant part is not changed by this operation, and the wily programmer not infrequently takes advantage of this loophole in the type system.) In practical terms, other than the sudden undefinèdness of the values of the variant fields (not an anomaly peculiar to this situation, since Pascal is notable for its uniform *dis*couragement of the idea that mutable objects should have defined values throughout their lives), the main consequence is that enough space must be allocated to permit the representation of the largest of the possible variants, whether or not this is the variant represented by the present value of the record. In fact, the necessity of determining this value is one of the reasons that variants fields are so commonly constrained to be enumerated.

Turbo Pascal Strings: Character strings in Turbo Pascal are characterised by two parameters: a *present* length and a *maximum* length. Of these, the latter is static and forms part of the object's type, while the former is dynamic, and precedes a buffer whose size is equal to the maximum length. Assignment to a string variable results in the characters of the string being written into the buffer area, and their number (which must be less than the destination string's maximum length) into the present length field; buffer positions indexed by values in excess of the present length lack definition, whatever their relationship to the maximum. In *Exemplar* we could express this arrangement thus:

```
  TYPE :< TurboPascalString @ {0-255} limit >:
     (#{0-limit}) size
   , Vector[#Character, limit] _ :: v
   ; for N i to +size
        do Character [i] : v[i] end
; ...
```

Each of the above is, of course, a special case of a dependent product in which the dependent type is *im*pure.

4.8 Software Engineering Issues

4.8.1 Pros and cons of strong typing

The most frequently noted advantage of strong typing is that since it computes fairly detailed information about each value in a programme (and imposes on the programmer to make this information available) it permits much more detailed static analysis of the source programme, yielding benefits in both reliability and efficiency.

From the reliability perspective, the advantage of strong static typing is that programming errors are flushed at compile time that would otherwise not be detected until runtime. The obvious benefit is that bugs are detected sooner (lest the reader be uncertain of the truth of this statement in the context of a modern integrated development environment, in which editor, compiler and runtime system are simultaneously active on the programmer's behalf, be it noted that finding a bug by executing a programme can involve a wait of hours or days while the necessary conditions for manifestation establish themselves; in fact, in the case that the bug resides in code whose purpose is to enhance fault-tolerance, disaster recovery or portability—or any such application where the importance of the function justifies code whose overall probability of eventual execution is less than one—the burden imposed on the programmer by the dynamic testing approach may be infinite). Furthermore, they are often automatically localisable to specific interfaces, not leaving the programmer with the uncomfortable situation of knowing that a bad value got loose in a data structure "somehow," but with no clear idea of quite how. The crucial observation is that strong typing—especially with a powerful type system capable of fine distinctions—makes available a flavour of automatic software validation that is independent of execution path.

Efficiency is potentially enhanced by any technique that makes better information, especially better global information, available to the code generator. Type information is particularly valuable in this regard since it is fundamentally structure, type's runtime correlate, that drives instruction selection in conventional architectures. Information can similarly be propagated in the reverse direction, from code to data, resulting in better data structure selection; and the effect is synergistic—loop optimisations and array re-representation can, for example, be carried out in parallel.

Less obvious is that (as indeed is the substance of this paper) coarser-grained operations can be supported in strongly typed languages than would otherwise be available, by exploitation of the kind of big-picture analysis that strong typing permits. In particular, strong typing makes possible certain simple inductive arguments about the possible states of a programme that would otherwise (in the light of computability results) not be available.

But what of the disadvantages? Often cited is that strong typing increases the verbosity of a language, entailing greater programmer overhead in writing code. On the assumption that type information is not in fact useful to the programmer or necessary to the translation and execution of the programme, this is a valid criticism; but it seems more reasonable to couch it as a question: in a given language proposal, does the extra effort incurred in dealing with the type system actually pay for itself, or not?

Proponents of strong typing frequently argue that, as with descriptive identifiers, typing pays for itself (at least in the case of code that must be maintained and reused) by dint of code self-documentation alone: a type declaration on an interface is a much clearer guide and a much firmer guarantee that a function behaves in a certain way than is likely to be provided by any practical naming convention in isolation.[83] If that "overhead" is actually of use to the programmer in merely reading the text of the programme, the argument from verbosity is difficult to make.

Another criticism is that strongly typed languages provide much weaker *practical* support for common operations such as input/output and for unavoidable debugging activities. In this thesis we argue that in fact this is a cost incurred not by strong but by *weak* typing (in point of actual fact the type system of LISP is *stronger* if less rich than that of C or Pascal, each of which sabotages any potential for generic mechanisms that the language might otherwise have had by providing trapdoors out of the type system—and, at least in the case of C, actively requiring their use), as techniques are presented for providing these kinds of facilities at reasonable cost in a strongly and richly typed context.

Semantic paucity has also been claimed of strongly typed systems, on both theoretical and practical grounds. On the theoretical level it is claimed that there are many programmes that simply cannot be expressed in strongly typed languages. While often this criticism derives from an excessively limited view of the possibilities of type systems (see section 4.6 above), it does in fact hit home in two areas. The first, and weaker, of these is that the programmer might wish to write a programme that is not

[83]It is also worth noting that it is not necessarily the case that strong typing be manifest in the source code at all [DM82].

conceptually typed—presumably something that by its very nature uses data in a way that does not correspond to that in which it was generated. This argument is weak because it requires that the programmer intend an interpretation of the data that is not among the interpretations that the data had when generated, yet the data were (presumably) generated on behalf and under the instructions of that programmer. In short, the programmer is hacking: attempting to extract information from the language system that, whether for reasons of correctness or efficiency, the abstraction provided by the system normally conceals. It is not clear that it is desirable, at least from a software engineering perspective, to support this particular undertaking.

The second theoretical argument is of a mathematical nature: what if there are programmes which while conceptually well-formed, are not provably well-typed within whatever system has been selected for the language. This argument is thoroughly valid, examples in the author's experience having included the **Y** combinator, various parts of the visitation system, and a software verifier [Dev84] that is (correctly!) theoretically unable to verify itself. The expectation is, however, that such programmes are rare, and amenable in practise to any number of trapdoor mechanisms (whereby, for example, the consistency of a recursive nest of type definitions is established by programmer assertion and not by mechanical proof).[84]

A more pragmatically-minded argument for the semantic paucity of strongly typed languages is that they tend to lack such facilities as alternate representations for, for example, "long" and "short" integers. But as we have shown, this is in no way a necessary feature of typed languages; in fact, it is the very typèdness of a language that makes such facilities possible. If it were not for the fact that each datum *has* a guarantee of interpretability associated with it, it would not be possible to provide automatic selection of the appropriate operations for the representation that happens

[84]Note that those programmes with subtle correctness proofs not expressible in the type system pose no problem, in that they can still be represented in the language at the expense only of safety; unlike the more pernicious forms of type evasion discussed earlier, such programmes involve fibs not of *commission* but of *omission*.

to be in local use.

One final point that has been made a number of times recently is that systems are increasingly arising which must run perpetually, in spite of constant upgrades and occasional profound changes in their internal workings. It has been argued that strong typing greatly complicates this manner of undertaking by making the reconnection of functioning components much more difficult. While it is true that such systems are in need of polymorphism and other such flexibility in their type systems, it is not at all clear that strong typing is a disadvantage. In fact, it would seem to take a great deal of daring to reconfigure running systems at all, and doing it in the absence of adequate mechanical assurances about the compatibility of the components with their predecessors and "interfacemates" would seem downright foolhardy.

Perhaps this feeling of the inappropriateness of strongly typed languages to dynamic reconfiguration is an artifact of the association between these languages and traditional implementation techniques, in which compilation is a once-only process. In fact, in the presence of the mechanism described in this thesis, it should be comparatively straightforward to implement systems in which compiling and recompiling modules of a running programme is a fairly straightforward task, without incurring any data coherence problems that cannot be solved mechanically, though we have not explored this issue in any depth.

To summarise, it seems that most of the arguments usually levelled against strongly typed languages are in fact arguments against *weak* typing, and the corresponding advantages that are derived by users of the unitypic languages like LISP and Smalltalk-80 are in fact due not to the *typelessness* of these languages, but to the fact that their type systems, if not rich, are inviolable. True strongly typed languages have the potential to reap the traditional rewards of both camps.

4.8.2 Impact on modularity

Another important question to ask about strong typing is how it interacts with the ideal of software modularity. One rather facile response is that without strong interface checking, modularity, at least modularity in the large, is an administrative impossibility. Unfortunately, traditional approaches to the topic bring with them a significant performance penalty when information needed by the implementation is hidden at module boundaries. Furthermore, the drive to modularity can be self-undermining when the domains of modular control are restricted to coïncide with the "natural objects" of the language: modules end up being constructed that cross the inherent semantic and administrative boundaries of a project, because they are required to gather together information deriving from or needed by the different modular domains, even though these parts have no significant internal interaction.

Types, Modules and Compilation Units: The efficiency concern is clearly not much of a problem where modules form part of a single compilation unit: the fact that opacity is enforced at the level of the source language need not constrain the implementation in its optir˙˙ation, since the safeguards against illegal and undesirable interactions have already been applied by the compiler front end. Between compilation units the classical solution has been to ignore the problem and require that all interfaces on the surface of the compilation unit observe standard calling conventions, a practise that is tolerably efficient in conventional strongly typed languages (since the calling conventions will presumably be well-matched with the data representations in any case). Unfortunately, on the one hand, once transparency of data representation has been abandoned, literality no longer constitutes a straightforward solution; and on the other, the adoption of a type system sufficiently flexible that maintaining its inviolability is not a liability makes inter-modular representation optimisation a necessity[85].

The same techniques that serve to recover intermodular efficiency across compilation unit boundaries, however, will solve this problem directly. In systems (such as Mary2 [pen83][86]) in which code generation is deferred until link time, data representation selection can be treated similarly, being based on information collected from all the parts of a compilation. Similarly, a system based on maintaining a common database of modules in various stages of compilation and optimisation should experience no difficulty in supporting sophisticated data representation techniques intermodularly.

The conflict between problem-oriented and language-oriented modularity constraints is also exacerbated somewhat by treating types as "real" objects with intelligent implementations and a certain level of semantic sophistication: it is no longer adequate to consider types as "mere" parts of the interface language, but things with nontrivial implementations in their own right. In fact, it is not really the case that such issues do not arise in systems that place less emphasis on types, but it *is* there less clear that the representation of data is one of the loci of concern.

The same constructions that alleviate the problem of poor scope boundaries for code structures can perform the same service for types at the module level: as functions can be provided with clausal definition mechanisms[87], so it should be possible, by extending the familiar concept of structures with separate public and private parts[88] to create type definition mechanisms that are likewise distributed and clausal,

[85]Consider for example the implementation of universally polymorphic functions, whether this is accomplished with (as in *Exemplar*) or without an explicit type parameter.

[86]Or Mythos, an experimental operating system (relying heavily on the techniques developed in this thesis) presently being designed by the author, which takes the even more radical step of considering code generation to be a reversible optimisation step, akin to process migration or data caching.

[87]This facility for the spatiotemporally distributed definition of abstract functions, to my mind, rather than the extreme-late-binding execution model, is the fundamental advantage of Smalltalk-class languages; and it is one that they share with many contemporary functional programming languages.

[88]as in C++.

whether the clauses be associated by sum, product, or (possibly) restriction.

A formal reconstruction of this policy in the case of functions (and data objects: consider, for instance, a slight extension to C which would permit the programmer to introduce a value 0 into an arbitrary novel type) can be provided in terms of objects with polymorphic types being granted overloaded implementations. In the domain of types themselves, the correlate of polymorphism is underspecification (typically in the form of the declaration of a subtype or a supertype—either can be appropriate, depending on whether they underspecified type is destined for use in negative or positive contexts). Types could be introduced in a partially specified form and refined independently in distributed scopes. The effect would be that various aspects of the structure of a type could be isolated within separate modules.

In order to maximise its utility, the practical application of this technique would require careful consideration of issues such as default values, subtyping, and visibility rules, but should pose no insurmountable difficulties; it would pay off in providing considerably better support of modular objectives in the strongly typed context than is possible in weakly typed or unitypic languages, and with no loss in safety. This entire approach, of course, is predicated on an adequate solution (of the extreme case) of the separate compilation issue, as discussed in the preceding paragraphs.

Levels of Type Abstraction: Another issue of relevance to the interaction of strong typing and modularity is the extent to which a strongly typed language can support changes in the apparent linguistic repertoire. It should be clear that although it has been conventional to provide specialised notational facilities for the types built in to the language, this is no more a necessary characteristic of strongly typed languages than it is of weakly typed ones (perhaps the less so, in that the type mechanism itself exists to mediate the allocation of new denotations). Similarly, in the absence of a well-developed type system, issues of representational opacity cannot well arise: if there are not types the details of whose structure can be withheld, then representational opacity is at best a matter of programming discipline. Finally, the presence of a type system provides an intensional basis on which operator overloading can be resolved, potentially increasing the level of conceptual abstraction of the source programme.

An important characteristic of (intensional) type systems, then, is not that they permit abstraction—any programming language provides, after all, an abstraction of the underlying machine—but that they permit and encourage *layered* abstraction. In particular, since a change of type need not imply a change of representation (something which cannot be the case for extensionally motivated systems), this layering of abstractions can be accomplished at low—and often zero—cost. It is the value of **Visitors** that they permit controlled unraveling of such abstraction without doing violence to the layering itself: from each point of observation within the programme the composition of a datum will be coherent with the virtual type system visible at that point.

Chapter 5

Representing Types

To this point we have been more or less relying on the transparency of the relationship between a nonatomic type and its natural representation to facilitate our discussion of the type (and "type") schemes of extant languages, and to carry the weight of our assertion that **Visitors** can manipulate objects of arbitrary *type*, not merely arbitrary *structure*. In practise (as we have noted more than once) the relationship between a type and its *actual* representation may be far from transparent (in the worst case, involving an arbitrary transformation function—see section 4.2.2). In particular, wherever **deferred evaluation** occurs—because a type is explicitly *lazy* [Gro86, RC86], because an object is (as yet) underparameterised (as with a function value), or because of the normal evaluation sequence of the language (as in a normal reduction order language or one, like ICON, providing streamed expressions)—the data structure underlying an object will be bound up as much with the execution mechanism of the host machine as with the actual datum eventually to be delivered.

In fact, the entire range of such phenomena turns out to be strikingly easy to handle, once noted that the realities of von Neumann architecture[1] strongly restrict the forms of data structure that are sufficiently efficient to be chosen as underlying representations, even of—perhaps especially of—system-related objects.

In this chapter we at last consider in detail the actual relationships that are to be

[1]As it presently sees implementation, at any rate.

found between types and their representations in the host computer, both in terms of the structures that represent them directly and the **Visitors** that we provide for their manipulation.

5.1 Primitives

Turning first to the question of the structures that implement data types, it will be convenient (as it was for types in their turn) to divide them into atomic (or, rather, basic) and compound entities. Though we argued in section 4.3.6 that conventional hardware, being possessed only of "bits," is best analysed as being strongly but trivially typed, it will suit us for the moment to treat pointers as a distinct case[2] (these two cases will shortly be reïntegrated to some extent, in section 5.3).

Having considered the atomic data structures, we will turn to the different underlyingly compound objects, working our way eventually through to the runtime representations of types themselves.

The number of primitive structures that a machine represents is in some sense a matter of interpretation, and in another a point of no moment. Certainly no harm would be done—except possibly in terms of performance and code complexity—if a large number of primitive structures were recognised, covering all the different interpretations the hardware might place on a bitstring. Or again, a very practical choice might be to distinguish among different primitive structures on the basis of *storage class:* information in the stack being handled differently from information in a garbage collected heap, and differently again from statically allocated objects with indefinite lifetime.

The weakest (and for many purposes, the most satisfactory) distinction we can draw is based on the *syntactic* application of the data (in a sense to be explored more

[2]A distinction that has physical correlate, of course, in that pointers are ultimately emitted on the address bus: if the CPU is seen as an autonomous black box, pointers are more clearly distinguished from other data in manner of processing than even instructions.

fully in section 5.3, below). For the usual run of languages and machines, the memory representation of an object can be divided into: **raw bits,** whose interpretation is determined solely by the eventual use to which the programme puts them, and whose only inherent syntactic relationship is adiacency; and **pointers,** which have the additional property of referring to—and equally important, *transmitting type* to—remote storage locations.[3] For our purposes, then, we will treat all objects as ultimately decomposable into these parts. Any objects "atomic" at the higher level—such as Boolean values, references or floating point numbers—are in this stratum composed of sequences of pointers and raw bits.

5.2 Simple Composition

As has already been suggested, the very simplest composite types, the products and sums, are amenable to direct translation into natural structures. The product $\mathsf{T} \times \mathsf{U}$ can be represented straightforwardly as the linear juxtaposition of the representations of T and U; similarly, $\mathsf{T} + \mathsf{U}$ (since it is, in our terms, *if* Boolean t, t *then* T *else* U *end*, or equivalently $2 \grave{\times} i.\mathrm{if}_{i=1} \mathsf{T}\ \mathsf{U}$) becomes a Boolean structure followed immediately by the representation of either T or U, depending as it reads true or false.

Extracting the first member of a product is then very straightforward: it is accomplished by simply ignoring the second part of the product's representation (in fact it has already been noted that this is the basis of a straightforward form of type single inheritance). Since we treat the sum as underlyingly a product, the same observation applies to examining its discriminant field.

[3]In support of the technical device we proposed in section 3.1.2 for the representation of data with complex control expressions, we add to these two a third class of cache bits, which represents information redundantly with a computation over (the preceding part of) the represented object and thus has no "true" significance for the object's syntax, while still being crucial for its practical representation. (Note one subtlety, though: if the device is used to store locally computed information redundantly, the associated storage is indeed composed of cache bits; but if it is used to close over nonlocal or mutable data this is, rather, a true closure with semantic interpretation and composed of raw bits and, presumably, pointers.) The distinction between cache and raw bits is a bookkeeping device of use, for instance, if it is desired to make external representations of objects as transparent as possible; or in making certain time/space tradeoffs. See further the discussion in section 6.2.1.

Obtaining the value field of a sum is scarcely more complicated, for it immediately follows the discriminant, at a fixed location (we can assume the discriminant is of invariant form); thus it can be extracted by ignoring the *first* part of the representation. That the appropriate variant (that is: the one corresponding to the actual setting of the control field) of the sum is selected—that the correct type is employed in examining the storage following the discriminant—should be assured by the overall type correctness of the source programme and the formal rules that define it.

A difficulty now arises, however, in that there is no requirement that the representations of the two variants have the same physical size (since T and U are, after all, arbitrary types); even to stipulate[4] that they are "padded out" to the same size is inadequate if one of the variants is, for instance, of unbounded maximum length. This is particularly irksome if such a sum appears as the *first* field of a product: for now the *second* field is no longer at a fixed offset, but is simply found immediately after the first, a location that is, in general, arbitrarily hard to find. If we are to keep the cost of the second projection function for the product reasonable (and for such a primitive operation, a very *small* constant cost is really desired), something must be done to keep the locations of the constituents more predictable.

One approach to the solution of this problem would be to precede the first object by its size; but this fails to provide *direct* access to fields after the first, as will be evident upon consideration of the process of finding the n^{th} field of an array encoded with this technique. The general solution is to replace at least a variably-sized first field of a product (and perhaps every first field of a product, or every variably-sized field, or every field that can possibly be more than one word in length) by a *pointer* to the field's representation elsewhere in memory. (This must be either transparent to the programmer, or something that one is *required to request,* of course.) Since pointers are of fixed size, both fields can now be located in constant time—one with

[4]As does C in this situation.

a dereference, one with an offset.[5]

A less general but nonetheless interesting alternative is to take advantage of the isomorphism[6] between $\mathsf{T} \times \mathsf{U}$ and $\mathsf{U} \times \mathsf{T}$ to move a variably-sized field of a product to the end of the physical representation, where it will cause no such trouble.[7] This transformation has several conditions: there can be no more than one variably sized field (two, if negative offsets are permitted and various further conditions are met); the distributions of places in which the involved products are eventually used must be such that no confusion arises (in conjunction with associativity, this transformation can result in some of the projection functions requiring the copying of their results or producing discontiguous representations which must be handled appropriately); and the physical sequence of the fields must be either invisible or irrelevant (which is essentially a condition on **Visitor** construction and use).

5.3 Data Syntax

The comments of the previous section prompt us to a more general discussion of the *syntactic* properties of in-core data representations. We shall be picking up on a number of comments that we have made in the preceding.

Although it is clear that memory representations are subject to grammatical phenomena, their grammars are rather different from familiar string grammars. A first model (acceptable if only types so far discussed in this chapter are involved, and there are no pointers) might be that a data structure is just a self-terminating bit string, on

[5]The restriction we have imposed on pointers—that their type be uniquely determined—may require that a certain amount of "factoring" be done on fields before this transformation is applied, and ultimately limits its utility somewhat: there are cases where the output of the process is not terribly felicitous. This aside, no typing problem can arise since the type of the moved field is transmitted through the pointer.

[6]Care must here be exercised: these two types are only isomorphic under appropriate renaming in the introduction and elimination operations; and we are not in a position to do anything of this sort for the *dependent* product (though the reader wishing exercise is assured that there are things that can be done).

[7]This representation optimisation corresponds directly to reorganising a function in order to permit tail-recursion and subsequent tail-recursion elimination.

which an LL(1) syntax is imposed by the generating structure declarations. In fact, however, there are a number of complications. First, a data structure consists potentially of *two* bit strings,[8] progressing out in both directions (towards increasingly positive and negative offsets, that is) from an origin at the object's nominal address (objects in registers or other disjoint address spaces are easily handled by extension of the basic address). Second, the structure of the underlying alphabet of these strings is complicated (and not merely extended) by the existence of pointers: for certain contiguous "words" of bits can be grouped together into pointers *which are not distinct from the underlying binary symbols.*[9] Third, since we have been permitting *dependent* types, items appearing in the right-hand-sides of productions (corresponding to fields in a structure description) are permitted to have arbitrary functional dependencies on (what amount to attributes computed in a single left-to-right pass over) material appearing to their left in that production. Finally, just as, given that structures are (at least potentially) bidirectional, it is possible to use information alternately from each of two data sources, it is also possible to look ahead over any *fixed length* substructure—or gap in structure—and thus visit fields in noncanonical sequence or support discontiguous representations.[10]

As has already been observed, the presence of pointers permits us to "step around" variably, even arbitrarily, sized objects (subject to the condition—if we are resource conscious—that they do not provide information that controls the rightward interpretation of the structure); but in fact they also allow for arbitrary structure sharing (or "DAGginess") and even circularities in data structures (which in turn can be

[8]Actually, there are cases of practical address spaces of higher dimensionality than one. An object oriented system might exhibit a microsegmented address space; and in general, any paged or segmented system could assign semantics to, *e.g.*, particular segmental offsets, allowing indexing through multiple pointer subfields. Although we are interested in such schemes (see, for instance, our discussion of "zones" in footnote 15 of section 7.2.2), we do not yet have a unified approach to offer for this more general case.

[9]Although imprecise garbage collection schemes—discussed in section 7.1 below—in fact employ heuristics in an attempt to finesse this problem(!).

[10]Actually, this last *can* be done in LL(1) through an arbitrarily expensive rewrite of the grammar.

used to represent formally infinite objects—even nontrivial ones, if information flows "down" the cycle). (This has profound implications for **Visitors**, which we shall discuss in section 5.6.4, below.)

On the other hand, there are a number of *soft* constraints placed on "good" data structures over and above those mandated by the logical structure just described, by the nature of their computational context.[11] First among these is that because it is undesirable to commit arbitrary machine resources to the mere examination of internal data structures,[12] representations are typically chosen to involve *at most* some subset of the resources of the CPU itself (and no additional RAM) in their analysis. Technically, this is but a finite amount of state and object syntax is thus *a fortiori* restricted to a regular grammar (but then the amount of addressable RAM is likewise finite). More realistically (or rather, more in line with the usual approximations), we find that regular syntax is still supplemented with integer arithmetic of single word range (in order to provide arrays, if for no other reason), and no more; since the maximum length of a physical array cannot normally exceed the maximum span of a pointer, this still fits the practical bounds.

In order to meet this practical complexity constraint, two techniques, dual in sense (and both have which we have already discussed) are employed: the moving of complex substructures out-of-line with pointers; and the precomputing of complex control information, to be placed in cache fields within the structure for ready access. These same techniques find application in the satisfaction of another soft constraint, the result of continuing the tradition that the primitive structure access functions of a programming language (corresponding to the eliminators of the type) have constant cost, ideally on the order of one or two additions. Aggressive application of our dual simplifying transformations can often, if not invariably, permit the allocation of fields

[11]Note that these are the *same* constraints that we previously observed to limit typical language *designs;* here we consider them as they impact all language *implementations.*

[12]Particularly since this *will* occur during garbage collection, when resources are at their most scarce.

to fixed locations within the structure, or at worst fixed within a single additive parameter.

Finally, because of difficulties in managing allocation, it turns out to be impractical to manipulate structures that are of potentially unbounded length in *both* directions, and inefficient if both halves are merely of variable length. In fact, the only applications for bidirectional *data* structures we have run across in the literature have been in the service of the goal of fixed field offsets.[13]

The practical constraints just mentioned apply with redoubled force to the host machine itself: in interpreting its instruction stream, at least, and possibly in locating the arguments to its instructions, a typical processor architecture abides strictly (modulo virtual memory!) by the dictates of constant cost operations.

One final, minor complication to the syntax of data structures (though a linguist would merely look upon it as bearing the hallmark of a morphological process[14]) arises when two (typically atomic, for reasons of simplicity of interpretation) substructures of a structure are assigned *overlapping* storage. This is distinct from mere "packing," in which substructures are aligned tightly, without regard for word boundaries, and which merely inhibits the straightforward construction of pointers to the entities involved; here we are concerned with the case that the *same bits* potentially have more than one interpretation. The most common case is probably that in which a pointer field has a nonpointer interpretation that is active in the case that the contents of the fields are transparently inappropriate as a pointer. This *might* be no more than a sum structure packed into a single word by the coöption of an "unused" bit to hold

[13]It is interesting to observe that bidirectionality has occasionally been suggested for object code—one application might be to ensure that even instructions with full-width operands can be of effectively fixed width in the instruction stream, and/or that "advanced notice" can be given of upcoming address literals without compromising normal instruction fetch mechanisms. It is perhaps also worth observing that there is an alternate interpretation of the contents of a control stack—at least during return—as being a secondary instruction stream consisting of calls in good old-fashioned following-argument format (a format which has by now been essentially abandoned as requiring writable code segments; but writability is in any event a *sine qua non* of stackhood). This is a particularly good fit to a hardware-threaded machine architecture.

[14]"We needn't worry about that, it happens in the lexicon!"

the discriminant field; but it is just as likely that there are particular values for the "pointer" that signify the second interpretation.[15]

5.4 Functions, Closures and Activation Records

Although a conventional array is a straightforward finite cartesian product, and is normally represented as a linear sequence of (by hook or by crook) equally-sized element structures, more general map types exhibit a wide variety of internal structure. Associative tables become complex combinations of underlying linear arrays and linked structures, while general (computational) functions (and various other objects for which a delayed evaluation mechanism is selected or required) have representations ultimately involving native instruction sequences for the host machine.

5.4.1 Functions

The simplest of the representations involving machine code is that of a closed, effectively *asynthetic* function not employing visitation and requiring only a bounded (and reasonable) amount of local storage.[16] These correspond precisely to the functions that are provided by C or Modula-2: they are typically represented by a simple pointer to an unadorned code sequence. Looking at the representation from the point of view of a garbage collector, at least, such a code sequence is simple binary storage; the **Visitor** need know no more than the function's address and possibly its length (if it is garbage-collectible). Thus the function as a whole is very much like a homogeneously stored counted string; though it will usually not be possible to store the

[15] *Exemplar's* "extension" values (see section 2.4) are intended to formalise just this practise.

[16] A function is **analytic** (not, of course, in the sense of mathematical analysis) if it involves elimination rules, and **synthetic** if it involves introduction; effectively so if the property is externally detectable. Note unanalyticity corresponds immediately to purity and asyntheticity to niceness when the latter are understood denotationally. For this very reason it is possible for an implementation to trade off one against the other; in particular a nice synthetic function can often be converted into an implementational form that is asynthetic but naughty, reusing the same (mutable) space to hold its result on each successive invocation. This is in fact how we analyse the use of processor registers, both as temporaries and for argument passing.

length of the code segment at offset zero (since this will be the entry point itself in most architectures), it can be stored conveniently at a fixed negative offset.

The simplest deviation from this basic case is when the function can return constants: while these can be evaluated at compile time,[17] the fact that they can appear in the function's output implies that they must lead a "public" life; in particular, it must (in general) be possible for them to live on even after the function returning them has been garbage collected. Under certain circumstances simply embedding such constants in the object code will suffice (small integers are often representable directly within an instruction, for instance, and since they are typically no larger than pointers, there is very little point in trying to avoid copy proliferation). Frequently, though, it will be necessary to grant such constants first-class status, and make them accessible to the **Visitor** for the function.

Any of four approaches to code visitation could be employed. The first and most traditional (it has frequently been used in implementations of both LISP and Smalltalk) is to segregate the constants into a separate **constant table**, typically either prefixed or suffixed to the code segment proper (since many processor architectures provide instruction-pointer-relative addressing for constants this is as convenient from a code-generation perspective as immediate data, if not quite as efficient in terms of space or bus bandwidth); data stored in this manner must needs be tagged as to their types, whether by universal convention of the implementation, or specifically in this application. Somewhat higher performance may be possible by storing data as immediate operands in instructions, while retaining the constant table with *indirect* references back to them (and retaining, of course, the tags). This second approach (aside from being particular to a function) is no different from a table-driven approach to data structure analysis, and immediately suggests a third encoding in which we

[17]Unless their semantics demands that they exist in a new *copy* on each scope entry, in which case we treat them as variable; this would include mutables and the scoped-distributed version of freely generated types that subsumes "atoms."

permit code and data to be intermixed quite arbitrarily, but provide for each function a private **Visitor** specialised to that function's particular layout (and capturing in code the information that the second approach placed in its table). Aside from the probable negative offset of the locator for this **Visitor**, this is essentially equivalent to the encoding we proposed for the "universal" type: a type descriptor with **Visitor** prefixed to the datum proper.[18] These second and third approaches are most clearly advantageous when it is considered that (in most implementations) every function is a constant, and the targets of call instructions are in general among the things that the first approach requires be placed in an external table.

The fourth, final, and technically most humorous approach to code segment visitation is write a **Visitor** directly for object code itself. In the the most general case this is not possible, since disassembly is in principle harder than execution, but *locally* instruction sets are very well-behaved—necessarily so, since contemporary architectures interpret straight-line code with a very simple finite state method. Very little additional information would need to be present for the successful decoding of the output of a typical compiler. Though the computational overhead of using this method would be substantial and it is unlikely to be practical for the present undertaking (in particular, note that information deriving from the return type of the function must be propagated *backwards* from the return point), it has real potential to reduce the amount of auxiliary data that must be stored in order to handle activation records correctly.

Dual representation of functions: In the foregoing we have been making a silent assumption of immense proportions, namely that the representation of a function is expected to be utterly opaque. In fact, there are a number of applications where this

[18]There is a subtle point here: this last representation results in a **Visitor** being embedded directly in an object, and since **Visitors** are themselves *code* objects, it will require a **Visitor** of its own. This is the concrete manifestation of the type-of-types problem, and explicit steps must be taken to ensure that the recursion terminates; see section 5.6.6.

is unsatisfactory. In the cases of debugging and (coresident) compilation for instance, it is necessary to actively decompose a function into its semantic parts, rather than treating it as an uncomplicated block of storage. More striking yet is the case of process migration (something that may be as unremarkable as an attempt to print out the value of a partially evaluated function, incidentally), where it is necessary to derive an external representation for a computation in the absence of any guarantee that it will be "resurrected" on a processor of similar architecture.

This difficulty can realistically be addressed by treating the native object code of a function as no more than an optimised form of some more portable—and more decomposable—representation.[19,20] The native code itself is then stored as *cache bits*, with the more abstract code being visited according to the type appropriate to its representation.

5.4.2 Activation records

In conventional applicative-order programming languages, functions exist in two distinct states: active and inactive. While active (while the flow of execution is transitively "inside" them), they require additional storage to hold their internal state. Sometimes (though in most implementations only for "leaf functions" at the fringe of the call-graph,[21] since the number of processor registers is quite limited) it may be possible to store this state entirely within the CPU, but in general it must be saved externally in an "activation record."[22]

[19]And by providing fairly conventional debugging information to relate the two views, though admittedly a roll-back debugger would make for a less constrained implementation.

[20]The author's operating system design, Mythos, rests on the notion that compilation is never more than an optimisation for anything outside a sub-kernel, and in fact integrates compilation, physical memory management and disk management into a unified caching protocol.

[21]Recall that such leafness is *not* unambiguously defined by the language but is itself an implementation-dependent property, since, as we have already observed, the representation of functions is exceptionally subject to transformation and optimisation—if only because this has been the traditional focus of work on optimising compilers.

[22]What manner of storage regime is appropriate to the activation record depends on the function in question, and how it is used. In the simplest case of a non-recursive function in a single-threaded language, its storage may be allocated statically from a pool, in such a manner that it possibly overlaps only the static storage associated with other functions that cannot be active at the same

While certain mechanisms for the provision of parallelism make objects of process type available to the programmer as data (and the application of visitation to one of these would presumably permit the indirect inspection of that process's current activation record), and while a debugger would certainly provide this facility, it is not usual for activation records to become visible to the programmer. Wherever a function transitively allocates storage, however, the garbage collector may come to be activated to recycle allocated but no longer needed memory; and since it must identify all the storage that *is* in use it must examine the activation records of all presently suspended functions, so **Visitors** for them must be provided. Interestingly enough, it need *only* examine suspended records, since the recycler itself is the active function and its transient state is not among the structures that must be preserved through garbage collection.[23]

The activation record contains a number of distinct data (a list that is, perhaps unsurprisingly, very similar to the contents of a typical process descriptor, though lighter on the synchronisation support): the explicit local context (local variables and so forth); a parameter list (part of the linkage *between* activation records, that can be accounted either **upwards**, to the callee, or **downwards**, to the caller); an instruction pointer (again accounted either upwards as a **return address** or downwards as an **instruction pointer**); possibly a **static link** to the lexical environment (see the sections on closure, following); and a **dynamic link** to the calling (and thus dynamically enclosing) activation record.[24]

moment. Recursive functions, since they may have more than one simultaneous activation, need their activation records to be allocated on at least a stack. If parallelism is introduced, the stack must fork, becoming a forest or tree of stacks. In the most general case (as in Smalltalk-80, or Lazy ML), the activation record must be a heap-allocated object like any other, because its lifetime becomes completely unpredictable.

Which of these cases obtains is not of great concern to us, however, since the visitation mechanism will, by construction, successfully reach all relevant instances, as we shall see.

[23]Though its *static* structures are, at least as they pertain to types that are still somewhere in use (and this is the crux of another subtle issue discussed in detail in [Spa86]).

[24]This last may be implicit if activation records have known (or represented) size and are stored (almost everywhere) stackwise.

Since the local context is composed of declared variables (and auxiliary variables introduced in compilation), we can infer that it has a **Visitor** obtained from the structure declaration that results from abstracting just the declarations themselves from the function to which the activation pertains. These can depend on the arguments, (the object at the other end of) the static link, and possibly the dynamic link (depending on the semantics of the language[25]). Most especially it depends on the saved *instruction pointer,* since the location in the owning function at which execution was suspended in general determines which declarations are in scope.

The argument list is, from the callee's perspective, part of its static type and thus depends on at most its static link. From the caller's perspective it is a static property of the point of call, and thus depends on the instruction pointer and all the other factors that we just saw influence the local context. From either of these two angles, however, it is another simple structure (in *Exemplar* it is even notated this way).

The static link is discussed in more detail below, but it suffices for now to observe that its format is (typically) fully determined by the function with which it is associated, and while this must be determined it depends on none of the other, more transient data in the activation record.

The dynamic link, and the instruction pointer or return address, are themselves simply pointers; and the referent of the dynamic link must have a format sufficiently well agreed-upon that it can be used to restore the old context. Thus, unless explicit disambiguating information (presumably classed as cache bits) is inserted into the activation record at a pre-arranged location, with concomitant runtime cost, the saved instruction pointer must serve to ground the interpretation of the whole activation record.

[25]Note that if the language does provide some form of "deep binding" dynamic scoping—meaning that the dynamic link must be traversed in the normal course of function execution (before, that is, the final return)—nonleaf activation records are in general constrained to a physical format with an invariant component holding this information, while otherwise—and if the correct Visitor can be located—idiosyncratic layouts are possible.

(Even in the presence of explicitly cached information about the activation record it is not safe to treat the instruction pointer as opaque, since it asserts the reachability of the object code of the function. It cannot be assumed that this is independently accessible from the static form of the caller—as would be superficially convenient if only garbage collection were at issue—since the activation record may represent an invocation of a variable function; nor can it be assumed to be available via an activation record elsewhere in the stack, since the originator of the function—and even its caller, in the presence of the all-important tail-recursion optimisation—may no longer be extant.)

The most straightforward method of performing the feat of mapping the return address to a **Visitor** for the activation record is to precede the return address's target (presumably an instruction starting the code for the continuation of the suspended computation) with a descriptor not only for the code found there but also for its anticipated dynamic state. This information, being associated directly with the code, is (relatively) static and involves no direct overhead at runtime; but it has the indirect consequence of making it impossible to use "conventional" subroutine linkage instructions in a straightforward manner, since now each callpoint must have a fixed data field associated with it, between the point of call and the point of return[26,27]

A somewhat less intrusive method that works for functions known as constants (and so represented by calls to fixed addresses) is to place the administrative information in front of the *entry point* to the function (instead of at its call point). This can be located by standardising the calling instruction sequence, locating it from the saved instruction pointer, and decomposing it to produce the target address. Since this fails for functions invoked anonymously, it must be backed by another (possibly

[26]Much in the manner of archaic PDP-11 calling conventions, in fact.

[27]For some instruction sets there is not an execution time penalty associated with using nonstandard linkage conventions adapted to this requirement and completely avoiding the call instruction, except possibly in terms of second-order effects relating to instruction cache residency. Such an efficient non-standard call sequence is exhibited (in this case for the Motorola MC68000) in [Spa86].

less efficient) method, perhaps working by associative lookup on the possible return addresses into functions suspected of use as anonymous values (though as functional programming gains in popularity the frequency of this case can be expected to rise sharply).

Finally, it may be feasible to gamble on there being relatively few activation records (rendering a more expensive solution acceptable) and store the requisite information in a separate table; this is certainly a coherent fallback position if it is impossible to associate data with a callpoint.

In any event, traversal of the stack (or whatever other structure the activation records might be organised into) is always possible by the same argument that supports **Visitors** in general: since the machine must interpret the stack frames (or discard them), they are all interpretable (or ignorable). Particular architectures may favour root-to-leaf or leaf-to-root traversal (depending in part on the "accounting method" that assigns components of the activation record to the caller or the callee, in part on matters of optimisation as touched upon shortly following, and in part on the level of stylisation of the stack itself[28]) and will certainly have profound impact on the best method of attaching hints to the programme for the minimisation of visitation costs. The range of options is sufficiently wide, however, that a good compromise seems to be consistently available in practise.

For a more detailed discussion of the data structures necessary for the efficient traversal of the control stack, and the visitation and analysis of activation records in the context of garbage collection, the interested reader is referred to [Spa86] and [Gol91].

Activation Records of the Optimised: Two complications to the above analysis arise in the presence of competent object code optimisation. The first is relatively

[28]Both the SPARC and the VAX (at least under the grossly inefficient CALL/RET discipline) are notable for their extremely stylised stack formats.

trivial: interprocedural analysis often produces *specialised entry points* for functions, in which arguments are passed by compile-time negotiated once-off conventions (possibly even being suppressed if their values are known ahead of time) or where certain nominally necessary integrity checks or case analyses are dispensed with for privileged customers that are (statically) known to meet appropriate criteria. These nonstandard entry points may be present *in addition* to entry points representing the function "as declared."

That a function should have more than one entry point is a minor thing, easily taken care of by replicating any per-entry-point administrative data; and any nonstandard calling conventions need not concern us in that by the time the called function *itself* makes a call (thereby laying its state open to inspection) the situation will have normalised itself (possibly on the understanding that the argument list and the local state must in some way be merged into a single data structure).

More gruelling is the fact that the moving of values into registers is the single most direct way of improving code quality for conventional architectures. If registers are managed on a *caller saves* basis, this presents no difficulty at all (in fact it is isomorphic to the case where no registering has occurred but the values are "normally" stored where register save happens to leave them); but where the *callee* saves, register contents can be dislocated arbitrarily far up the stack from their contexts of definition—for nothing would be gained by a callee-saves convention in which such registers were saved regardless of whether they are actually clobbered.

The only evident solution to this problem involves maintaining a model of the register contents as the stack is scanned: if the scan is root-to-leaf it carries type information up to where the values are eventually stored, while a leaf-to-root scan must carry the values down to their types. Fortunately, in either case, the set of registers is normally quite small, and the only significant overheads are (1) that the physical implementation of Visitors must make provision to pass enough state between them to perform this task (which may in turn have impact on their optimisability), and (2) that information about where registers were spilled must itself be recorded in the static descriptions of linkages.

5.4.3 Closures

When we discussed the representation of functions, above, we restricted ourselves to the consideration of those that appear at the *outer level.* Lacking any defining context that is not fully static and available at compile time, the "static link" in their activation records is truly static, and so construction of an activation record purely on the basis of the code address of the function is possible.

Functions in statically scoped programming languages, however, depend crucially on the bindings in effect at the moment of function creation, and in internal scopes (where functions are effectively *computed* values) this postdates the start of execution. When a function is created the current bindings (at least insofar as the function actually depends on them) must be saved as part of the function value itself. The combination of an executable component and an environment is known as a **closure** (because the free variables of the function text are *closed* by the environment structure, which is formally a substitution list).

The simplest method of doing this is to represent general functions as **duplex pointers**, pairs referring separately to the function's entry point and to its environment, the activation record of the creating function (this latter pointer is the proto-static-link that will be employed when the function is eventually invoked). Once again, the value of the entry point reference suffices to determine the type of the environment pointer's referent. Some care must be exercised: if the creating function can exit while the created function is still in use, the environment must be copied to[29] a place where it can outlive the function's activation proper. For the sake of consistency, outer functions can of course be provided with dummy environment pointers.

[29]Or in general, if it contains directly mutable values, have been built in.

An equivalent technique (with different performance characteristics, and not possible on every architecture) is the **autoclosure**: rather than unifying every function value under a duplex representation, a common simplex representation is employed, and conventional closures are replaced by pointers to *dynamically* generated code segments. This method is practical (and in particular does not involve the full force of a conventional compiler at runtime), because the constructed function need only perform two tasks: it must create the static link from the (to it, constant) environment pointer and transfer to a code address common to all instances of the textual function it derives from.[30] The scope for sophisticated optimisation optimisations is, however, very broad.

Closed Data: Sometimes objects other than functions are in need of closure. These include: *lazy lists* and other explicitly lazy structures found in otherwise strict languages; values whose context dependencies are easiest left in symbolic form until they are finally eliminated (type values being an example); and the part-code, part-data structures that might result from the interprocedural optimisation of specialised recursive nests.[31] Such structures typically have multiple code pointers sharing a common environment; but the overall effect is no more than the application of the tail-recursion and common subexpression elimination optimisations to a type with function components.

As with simple functions, both closure and autoclosure approaches are feasible; in particular the representation of *all* data as autoclosures with the code address

[30]In typical architectures, and if environments are *copied* into heap objects when functions are constructed, this is the work of a single instruction: a conventional subroutine call pushes the address following itself onto the stack and transfers to a fixed target. If the return address is popped into whatever location is used for the environment pointer by the target of the call, the same effect will be achieved (modulo a constant offset in the environment) as if the "trampoline" sequence had consisted of a register load and a jump.

[31]The state of the art in code generation is not yet such that the optimisations can be detected and applied automatically. Later, however, we will see just such a system of specialised functions resulting from the optimisation of Visitors for use in garbage collection, and we see no reason in principle why the process should not be automated.

doubling as a type tag appears to be a powerful representation method for use in graph reduction interpreters (since in this representation it may be as fast to attempt normalisation of an object as to check whether it has yet been normalised). This technique is described in [KL89] and has been employed by the author in the Jeter implementation of the functional database language Alfonso.

5.5 Types and Genericity

In an ideal world, types (which capture information about how values are used) would be information *about* programmes, not information *within* programmes, and would not need to be manipulated—or explicitly represented—at runtime. Even where the language permits types to be computed, these computations are in principle the concern of the compiler rather than the runtime system, for at runtime the data stand on their own.

Unfortunately, practical resource constraints (and such human activities as debugging) intrude upon this idealisation in a number of ways, ways which have the joint effect that data are in practise examined asynchronously to the manipulations the programmer requested explicitly, and type information must therefore be made present to the runtime system.

Leaving aside externally motivated tasks (like debugging), there are two ways in which the finitude of resources can have impact on the programme. The first is the very empirical matter of resource *exhaustion:* in particular, as a computation progresses and memory is used to hold intermediate results, the space available on the computer may run out. This necessitates a *garbage collection* in which values that are still to be used are distinguished from those that are now finished with and can be discarded, and in turn this requires (in general) a traversal of all the storage that is still in use, to make that discrimination. The types of the objects being traversed must be manifest to the garbage collector so that this traversal is possible.[32]

The second (and more profound) respect in which resources are limited is that the von Neumann machine has finite words and is in this respect not a faithful mirror of the computational domains of denotational semantics. While memory cells may *refer* to values of arbitrary size anywhere in memory, they cannot *contain* them, and in consequence even the trivial manipulation of an "arbitrary" value is a potentially nontrivial operation. Thus, where generic functions are to be implemented (even where the language's notion of "type" does not carry the additional burden of providing the versions of operations appropriate to its members—as it does, for instance, in Russell [DD85]—and even if the surface notation does not mention types, leaving their determination, as in ML [MTH90], to an inference mechanism), the types of generic arguments must normally be passed explicitly in the implementation.

How, then, do we represent a type?

As already noted, in some languages a type bears explicit runtime information in the form of a tuple of functions implementing the primitive operations on that type,[33] but this is in principle a semantic argument implicitly determined by the type, rather than the type itself. The *actual* type must capture the information necessary to manipulate members of the type in general, global ways: in particular, to traverse it and locate its constituent parts—to identify the information it holds. In combination with the appropriate arguments, this is precisely the function performed by the type's **Visitor**, and so our preferred representation for a type is in fact a (possibly optimised) closure containing at least the **Visitor** and its argument packet.

In the absence of **Visitors**, the only obvious options for direct type representation are a symbolic representation of the type declaration itself, to be manipulated

[32]The same observation applies *mutatis mutandis* to a virtual memory system, incidentally, an observation that applies nontrivially if the type system of the memory manger has not been—as in virtual object stores it has not been—reduced to involving only "pages" in fixed virtual pageframes.

[33]The type of this tuple is presumably fixed at compile time by the type or types of which the given type is a subtype, for otherwise this representation of the type itself would be uninterpretable.

interpretively; or the avoidance of the issue by reducing the information content of the type to zero.[34]

5.6 Visitors

Crucial, then, to our approach to the representation of types is the physical implementation of **Visitors** themselves. Just as we relied on the naïve assumption that instances of types could (at worst) be implemented by their natural representations to carry us through to the present chapter, so we have implied that, since **Visitors** are executable, they are representable in the same manner as other functions. While this is (modulo issues of typability at the source level) strictly speaking true, it is also a gross oversimplification, in that in many (perhaps most) of their anticipated applications they must operate in unusually constrained environments and with the greatest efficiency possible. At the same time, and in compensation, they are quite restricted in form and thus amenable to the application of specialised implementation techniques.

5.6.1 Mode objects, $\mathcal{M}$

In section 3.3 we presented a formulation of the **Visitor** founded in mutual recursion through a finite **mode** function of type $\mathcal{M}$, whose domain is the enumerated type $\mathcal{T}$ of type constructor names and which typically maps almost every constructor to its **Visitor**. The utility of this arrangement arises from the ability to update the initial (purely visiting) mode function pointwise to yield modes tailored to a particular

[34]There are two methods of achieving informationless type: the first is that employed by LISP, in which all data are explicitly tagged and there *is* only one type; the second is that of Modula-2, in which opaque types are pointers with the information about their referents *erased*, rendering them effectively unmanipulable. The former approach, as we have seen, has negative (but bounded) performance impact; while the latter makes certain services, such as automatic storage management, impossible.*

*Although the current literature of garbage collection includes schemes for *imprecise* garbage collection which are claimed to work for languages of this class (such as C++), these by no means amount to an automatic storage reclamation system, since they (so far, at least) only work in cases where garbage-collected types are partitioned from each other in the heap [BW88].

application.

Although $\mathcal{T}$ is finite and is typically statically determined,[2] it may in practise be beyond the capacity of the compiler to enumerate. In particular, the merging of information from separately compiled modules is difficult or even impossible for the linkers provided by a number of common operating systems. Thus the association of names to values that embodies a mode function is in actuality very similar to the abstract structure represented by the variable location mechanism of a dynamic-binding language like classical LISP. In particular, a deep-binding implementation (wherein the function is essentially an association list) is correct, but inefficient. A shallow-binding approach, wherein each type (or, more precisely, type constructor name) is an actual structure with a field representing its interpretation under the "current" environment is potentially usable, but requires careful management if the source programme is not such that only one visitation is in progress at a time: visitations must not see each other's pointwise updates.[3]

In practical application it appears that the number of simultaneously active visitations is often bounded; the majority of the applications thus far implemented have used visitation non-reëntrantly and with an almost entirely static *mode*. It would be possible, if not entirely desirable, to impose this manner of use as a constraint on the programmer (or at least to rely on it as the "common case"), in which event the best representation is the inverted one, the type containing a linear array of mode function domain entries, indexed by a representation of the function of the moment.

This certainly does not constitute a general solution; indeed, the ideal approach is probably to take the metaphorical bull by its conceptual horns and represent the

[2]In conventional programming languages it is limited by the number of type declarations in the source text and the resourcefulness of the compiler in producing divergent implementations; in those with first-class types it is dynamic and unbounded but still constrained by the constructivity of computation.

[3]The problem is even exacerbated slightly by the fact that the points of update—the relevant domain elements—may not even be mutually visible under the applicable type-name-scoping discipline and modularisation constraints.

mode directly as a single, contiguous linear array, even when this involves taking a survey of the dynamic extension of the type system at the moment of invocation. This is an extremely aggressive undertaking, but there are evidently opportunities for incrementalisation of the computation and the reüse of old mode values in light of their great mutual redundancy.

5.6.2 Physical representation of the Visitors of types

Laying aside for the moment the issue of optimisation, we recall that **Visitors**, the automatically provided domain-members of the mode functions and the actual entry points to the visitation system, are themselves at base just functions. In principle they diverge from run-of-the-mill functions only in that they are in some applications invoked by the implementation on its own behalf as well as by the programmer, much in the manner of assignment operators in conventional languages. This has the effect that a number of system services that are atomic from the programmer's perspective (and from the viewpoint of typical programmer-supplied code) may interact with them strongly. In particular, the basic data coherence assumptions that we have enforced in user code by type discipline and restrictions on mutable objects do *not* in general apply during the execution of **Visitors**, and the overall correctness of the computation must be safeguarded by careful consideration of the detailed behaviour of their object code.

The difficulties occasioned can be roughly partitioned into three groups: those that arise from resource scarcity (because **Visitors** are employed in the virtualisation facilities that provide the illusion of boundless resources to more "normal" code); those that arise from the detectable nonatomicity of object creation (when the **Visitors** themselves are involved in this genesis); and those that concern nonstandard transient data representations (as when **Visitors** are being employed to perform some global low-level data transformation).

Resource Scarcity: Because the visitation system may be employed in resource virtualisation (in their earliest incarnation **Visitors** formed the kernel of a garbage collector, the paradigm case of virtualisation-critical application; these same issues arise, however, when using them to perform bulk transput—as when saving a process image—or in support of virtual memory systems optimised by type sensitivity). The precise restrictions that such use will impose depend on the application (not a problem in that these particular applications typically form static parts of the language-implementation), but they can be characterised roughly as a requirement that the spatial resources locally consumed in the execution of the **Visitor** be bounded by a static constant, while their runtime be linear in the size of the object visited;[4] this will suffice for the purposes of the implementation's static "accounting" (for the active depth[5] and breadth of the heap are themselves limited directly by available physical resources).

As already noted, there are two ways of looking at this restriction: from a formal perspective, we can observe that we are restricting the **Visitor** to be (discounting any **Visitors** it may transitively invoke) a finite state machine and thus the objects it visits to be regular (as previously observed, this is almost always the case with data structures, anyway); pragmatically, we want to restrict the **Visitor**'s object code to use approximately the static resources of the CPU itself, eschewing the stack but for calls accountable to distinct objects.

Given the possibility we have granted ourselves of resorting to *cache bits* to record the results of computations on which layout may depend, this does not appear to present a substantive difficulty.

[4]And the constants should be measured in small numbers of bytes and cycles!

[5]Recall that we require that a pointer assign a *unique* type to its referent. In fact, even in the absence of this restriction we can still derive a bound on effective heap depth after cycle-breaking (using the lattice induced by the subtyping relation and a pigeonhole argument over the finite domain of addresses), but it is unacceptably large for practical application. Whether such pathologies would be common if the restriction were lifted is unknown.

Atomicity Failure: The second, formally straightforward but technically more troubling, respect in which care must be taken in selecting a physical representation for **Visitors** is that they are subject to failures of the usual creation (and update) atomicity of data. Since **Visitors** themselves may be employed in the creation, input, or motion of objects in memory, it is vital that they examine the data under their interpretation for semantic purposes only *after* they "visit" it by passing it to their mode argument for declaration.

The operational difficulty is that there is an orthogonal practical respect in which objects may be nonatomic at the level of their implementation: certain (very efficient and desirable) memory allocation methods use segment boundaries, invalid pages or their equivalents to delimit available storage, relying on an addressing fault to trigger the appropriate virtualisation strategy. More likely than not, such a fault will partition an object, and (if it is what triggers the fault) the execution of its **Visitor**.

The obvious consequence is that if the **Visitor** does not traverse the object's representation in *memory order* (as induced by the facility used to trigger the fault), a faulting object will not merely be incompletely initialised, but discontiguously so. This situation is easily defused by requiring that further manipulation of this object (up until the faulting operation is restarted) be carried out by the *same* **Visitor** and so meet the same fate.[6] More wearisome is that this technique for passively bounding allocation domains requires special preparation of any functions that might be in the position of triggering such a fault, so that the recovery process can reconstruct the formal state of the computation at that moment.[7] Fortunately, **Visitors** are typically very stylised in form and arranging for the necessary conformance at implementation

[6]This does actually entail a restriction that *all* **Visitors** of a given object employ the same sequence of visitation, which could in principle cause trouble; consider the (admittedly implausible) case of a bidirectional string terminated with NULs at each end, and known by different addresses within. Such troubles are avoided if partially instantiated objects are forbidden to have more than one distinct outstanding reference outside the **Visitor** itself.

[7]This is nothing but the familiar *imprecise interrupt* problem of pipelined hardware remanifesting itself in software.

design time should not prove excessively difficult; the required state description is similar to a **Visitor**, and the situation is akin to the provision of **Visitor** visitors described below—only simpler.

Representation Failure: Particularly while data are in transit between two points, there are other respects in which they may fail to live up to the usual constraints on their form. In particular, we find that pointers are unreliable: if their referents have already been processed they may point to what is now an "old" incarnation (and one that may already have suffered collateral damage of some kind); and values in general and pointers in particular may be otherwise "marked up" or temporarily mutated to facilitate the operation at hand. The information content of the original value must somehow, somewhere, be preserved; but a layer of interpretation must be inserted between such fields and their use by the **Visitor**, or some secondary representation mechanism must be provided whereby separate "runtime" and visitation copies of the data are provided (one, perhaps, concealed as cache bits). Then the problem is reduced to that of reïnitialising the copies from each other as appropriate.

This trouble is especially spectacular in the case of **Visitors** themselves: for **Visitors** may well be active *while* they are being relocated, thus they may encounter themselves in a transitory invalid state. In part this can be addressed by the two-copy stratagem outlined above; but because of the inadvisability of invoking the entire visitation system recursively on its own *dynamic* state (a process that would not converge), steps must be taken to assure the inviolability of the *initial* **Visitors**, even at the sacrifice of some efficiency, at least until the entire system visitation completes.[8] This does not prove impractical.

[8]This does *not* apply to visitation occurring as part of "normal" programme execution and suspended by a system-level process; its state can be visited as can that of any programme, given that the appropriate activation record descriptors are available; we have chosen to permit exactly two levels of recursion in the system, corresponding to objects and to types. Formally we could extend this to any fixed bound, but the particular ends of resource virtualisation are best served by setting the limit to two.

5.6.3 The base cases: pointers and bits

The function of most of the **Visitors** in the system is the decomposition of types, whether defined by the user or the language, into simpler components. Ultimately, however, this decomposition must terminate, reaching the actual physical representation of the data. On our present analysis the process grounds out in the invocation of one of three most primitive **Visitors**: those for pointers, data bits and cache bits. Anything that does not reach the point of analysis by one of these functions was either empty, not part of the original object, or has had its analysis preëmpted by the replacement of the **Visitor** for some type of which it appears as part by a (non-visiting) user function.

The implementation of the primitive **Visitors** for both data and cache bits is completely trivial: they immediately return a pointer just past the end of the storage area described to them, performing no semantic action at all. Notification has been made that these storage areas exist, but, in effect, noöne cares. The case of the **Visitor** for pointers is scarcely more complex; to a first approximation (an approximation we shall sharpen in the next section) the **Visitor** of pointers simply invokes that of the object referred to (which it is passed as a parameter, "pointer" being for us a type-parameterised constructor rather than a type in its own right) and, on completion of this subvisitation, returns the address following the pointer. This ensures that the nonlocally "contained" value is visited at the appropriate point in the overall traversal.

The utility of these functions is, of course, in their very absence of effect: they make it possible to locate data storage while ignoring pointer structure; and conversely, to locate pointers while just "stepping over" the binary storage in between; it is necessary only to replace the **Visitor** for the component of interest by a particular semantic routine, and let the others take care of themselves. The burden of providing a complete cover of the local system is lifted. In the case of a very literalisticallyimplemented type (with untransformed **Visitors**)[9] there is even clear cause for a visitation in which *none* of the three primitive **Visitors** is overridden: overriding a set of specifiable types would have the effect of reaching all and only the subobjects having the overridden types, with any remaining structure bypassed.[10] In many other applications, of course, both pointer and data bit visitors (perhaps also that of cache bits for system-level operations) will be overridden, either to assure complete coverage or as an error-detection measure.

5.6.4 Pointers, circularity and fixed points

The somewhat simplistic version of the basic **Visitor** for pointer types just described in fact fails for one important class of data structures: those that are not (instantaneously) well-founded in their tree interpretation, even though they are *a fortiori* finitely represented.[11] These include **cyclic** structures[12] which refer (transitively) to themselves. It might seem that such structures as the infinite list of natural numbers (readily definable in languages sporting lazy evaluation) would also appear in this category, but since we here examine the *representations* of objects such an entity will have an unevaluated tail which is a cyclic code object essentially after the first style. Obviously, the application of a **Visitor** that immediately visits the referent to a pointer forming part of a representational cycle will result in a computation that only terminates when some resource limit (in all probability, that on stack depth) is

[9]In a sophisticated system the very fact that this style of visitation is attempted would be considered an indication to the compiler that such a representation should be selected.

[10]From the point of view of modularity this is an *a priori* somewhat doubt-inspiring operation and it is naturally the programmer's responsibility to ensure that this is in fact the behaviour desired.

[11]There is another sense in which all the data in a practical programme will be well-founded: the algorithmic manner of their construction assures that they are not only finitely represented but also finitely historied. We cannot, or rather would not wish to, take advantage of this fact, because it is not (generally) desirable for history to be represented explicitly. There is, after all, a very real sense in which computation advances through the loss of irrelevant information. Debuggers and fast synchronisation methods employing execution reversal or "rollback" form only a minor exception to this rule, in that even they are unlikely to record anything approaching a full history of an execution.

[12]Though, note, not all cyclic *types:* while the **Visitor** of a recursive type will be self-referring it is possible—as with the simple linear linked list—for a recursive type to sport only well-founded instances.

reached.

Given our insistence on the constant definèdness of values, such a cyclic structure must arise either as the result of an effectively atomic creation operation or through the modification of a mutable value; and in the case of such a side-effectual origin must involve precisely a cycle in the type. It is thus feasible, even straightforward, to outlaw the occasion of this case in the design of the source language.[13]

Such a dictatorial approach may not, however, be desirable. Given then that nontermination is not it, we must ask what a sensible semantics for the visitation of a circular structure might be: what is the result of visiting something that is "infinite," and how might it be computed? The practical solutions, it turns out, will also address another problem, less theoretically profound but almost as troublesome from the perspective of practical performance: that of handling objects with *shared* structure with acceptable efficiency (for in principle at least shared substructures can result in a term whose tree interpretation is exponential in the size of the physical representation).

The intuitively desirable method of handling cyclic structures is somehow to compute an appropriate fixed point of the client computation. For completely general applications of visitation there is of course no reason to believe that such a value exists; nor are we in a position to compute it.[14] Since the locations of potential cyclicity are exposed through visitation, however, through the visitation of types that the user knows to be cyclic and, ultimately through the visitation of pointers (which indeed conveniently declare their types), we can leave the solution to the concerned client. Occasionally, in fact, special semantics are attributed to shared objects, and leaving

[13]The effect is that achieved in LISP if not setf but setq is employed.

[14]An apparent exception is when the visitation is constructing an image of the visited value that mirrors the cycle, whether in code or in data. While the fact that this is the usual (and easy) case serves to motivate our more pragmatic solution below, we cannot employ it directly unless we provide a special facility for requesting this interpretation *at the source level;* for otherwise we are left with the task of determining when a programme exhibits the requisite behaviour, which is clearly infeasible.

their handling to the client may be preferable *in principle*.

PRINT-CIRCLE: One familiar application that runs into trouble with cyclic data structures is the generation of printed output. In COMMON LISP we find that the standard print routine (ultimately write) provides for this through the *print-circle* global control mutable: when *print-circle* is true, a careful printing algorithm is used which prefixes the output representation of all shared or cyclically referenced data with #n= (for some unique n) on the first textual occurrence, and then replaces subsequent occurrences with #n#, allowing the reader to reconstruct the original value. This is a very clear case in which the application has its own idea of how the problem should be approached. While there are several possible implementations of the technique, among the most straightforward we find a double pass over the term to be printed, with output occurring on the second. During the first pass we simply enter every object to be output into a table; excepting that if we find ourselves about to put something in the table that is already listed, we mark the entry as twice-reached and forbear to traverse it again. On the second pass we check the table before printing each object: if the current value is marked as doubly reached we output a tag, record the tag in the table and proceed; but if the value has a tag already we print it in place of the object itself.

Object-Marking: In principle, the same effect can be obtained in a single pass by tagging *every* object output, in case subsequent reference is required—in fact the tags could then be suppressed if they are assumed to start from some fixed value and increment with each object written—but the result is considerably less useful to the human reader (it does, however, form the basis of useful machine-readable data representations). Stripped of the mechanics of printing *per se*, we see that the essence of the mechanism for dealing with cyclicity is the marking of the objects that have already been reached. For a broad class of problems where the overall result can be computed on the basis of a single traversal of a spanning tree of the object graph (and notification of where the truncations that generate this tree are located), this would seem to be a good approach.

In particular, if the computation being performed is idempotent in the objects being processed—as when computing the maximum of a set of values stored in some data structure, perhaps, or when computing the amount of storage consumed by an object's physical representation—the calculation proceeds with minimal redundancy. Further efficiency is in some cases obtainable by marking objects as (previously) reached in the objects themselves, rather than in a separate table; in particular, if objects have a uniform tagged representation, then a bit or two in the tag might well be allocated to this end.[15] In those cases where it is permissible to destroy the object while traversing it (of which traversals intended to make new copies of the object elsewhere are often examples), it may even be desirable to store the administrative information necessary for the reïdentification of an object with multiple attachments *in place* of the original value.

The basic assumption of object marking, however, is that objects may be identified with their addresses. Unless data are tagged, this assumption may not be sound; in particular, it fails for the more general data representations discussed elsewhere in this chapter.

Traversal Marking: In fact, since the same address (or range of addresses) can in general be reached under any of a number of different types (something that was clearly impossible in our original COMMON LISP example), a full solution requires that (though *sharing* must still be reported whenever an overlapping address range has been seen) traversal must continue unless the (representational) type of the present visitation is dominated by the *join* of those of previous visits.

[15]Since this amounts to a static allocation there is a reëntrancy problem introduced. Either a static proof of nonreëntrancy or a straightforward fallback implementation would suffice to surmount this difficulty.

We can simplify this enormously, however, under our assumption that pointers transmit *unique* type. If the record of prior traversal is associated not with the object but with the reference to the object, then traversal will terminate after at worst reprocessing the immediate target of a pointer (since any transitive targets are either already marked or not yet traversed). In principle this results in a polynomial increase in complexity over the object-marking approach (when it applies), but in practise most data structures seem to be *almost* arbiform.

Once again (and particularly since the typical machine pointer representation has one or more bits that bear no information, being fixed as a matter of course) we might consider the collapse of the prior-traversal-recording data structure into the objects themselves, this time into the pointer values. Since it renders a fairly sizable external associative structure unnecessary this can result in a fair performance improvement.[16]

Implications for Visitors: The special handling of cycles and sharing in general, and the ideas of recruiting informationally redundant bits from pointer representations or of overwriting them or their referents with administrative data in particular, have direct implications for the implementation of **Visitors**; for they imply that at least during *some* visitations, pointers may be stored with other than their conventional formats. In other cases where the values of data were subject to the process of visitation we suggested that this would be an appropriate application of cache bits; but clearly we cannot predereference each pointer and store its referent explicitly.

Consequently, we are in need of a mechanism for examining the referent of a pointer in the face of these (information-preserving) disruptions. It is partly for this reason that we introduced the dereferencing functions δ_T in chapter 3.3, laying on their

[16]In fact we speculate that by arranging (or rearranging) objects within memory according to their use of pointers and the possibility of cyclicity, and by informing **Visitors** of these same facts, it might be practical to use a sparse associative structure to good effect, and with much reduced performance impact; this would solve the reëntrancy problem directly. But this is a matter for future investigation; at present the best solution would appear to be the use of bits in the pointers until they are exhausted (with priority going to system tasks such as garbage collection), and external structures thereafter.

shoulders the burden of interpreting whatever representation might be employed at visitation time.

The overall implication, then, is that a client of visitation wishing to visit cyclic structures must in general look to its own semantics in this case; but that the implementation can provide a reasonably efficient mechanism for detecting and resolving this situation in favour of the spanning tree approach. In particular, this is precisely what the system does on its own behalf when visitation is employed in such low-level undertakings as garbage collection.

5.6.5 Optimisation

Given our attitude that **Visitors** could be made fundamental to a wide range of facilities, some of them at very low levels in the language system, efficiency is obviously a matter of some concern. While ultimately we might rely on the steadily improving quality of code generators to do a good job of visitor compilation, here we comment on a number of optimisations which exploit the straightforward structure and limited scope of **Visitors**, and on the use of which a compiler might be coached.[17]

Tail-Call Optimisation: From the perspective of the formal analysis of **Visitor** performance, the single most important optimisation is the most conventional of those we shall consider. In the presence of the so-called **tail call** optimisation (in which the sequence call *target*; ret becomes a simple jump *target*), it can be guaranteed that no stack frame is buried under an arbitrary traversal unless it also accounts for the processing of an increment of the heap storage, effectively bounding stack consumption at a linear function of heap size. From a more practical standpoint, the result is a nontrivial performance improvement, for depending on the type structure

[17]In the case of a simple compiler wishing to exploit these techniques, it might prove more practical to perform code generation of **Visitors** directly from type descriptions (perhaps in more than one copy, with different optimisations applied) so as to exploit more of the potential for optimisation than would otherwise be available in the absence of a middle-end optimiser.

of the programme untransformed tail calls could easily amount for as much as ten percent of the cost of visitor execution.

Register Allocation: A second important optimisation in the compilation of **Visitors** is the appropriate global allocation of registers. Since the basic visitation mode of a programme's type system is a recursive nest of functions with completely homogeneous calling conventions and largely trivial bodies, it makes sense to allocate all of the arguments in registers. In particular, examination of the semantics of the **Visitor** indicates that the only argument that (after recovery of imperative semantics) is updated is the running pointer whose initial value is the object being visited and whose final value is the overall result of the visit. In the case of conventional product types this running pointer is never examined within the **Visitor** proper (except possibly for purposes of alignment adjustment). In sums, we find that previous values of this variable are used to access control fields, but (because of the desirability of constant-format initial structure segments, as discussed above) these are normally constant negative offsets from the present value and require the use of a temporary register on few architectures. Thus, if this argument is allocated to the very register in which it will eventually be returned as result, it too need (almost) never be saved or restored.

Besides the arguments proper, it turns out that for many architectures (having a plentiful register supply and lacking a branch acceleration mechanism applicable to variable targets) performance is enhanced by the (redundant) allocation of registers to each of the primitive, base case **Visitors**: those for pointers and linear storage. This is particularly effective in conjunction with the flattening optimisation described below (which significantly increases the ratio of primitive calls to visitations of user types).

Both these optimisations and that following are externally visible as deviations from the standard calling convention and require the coöperation both of the caller and of any routines passed into the **Visitor** as arguments for transitive call. This is a point of some moment, since conventional inter-procedural linkage optimisations are (for obvious reasons) suppressed—or at least relegated to a secondary entry point—in the case of non-manifest functions. The visitation system, however, constitutes an identifiable recursive nest within which *all* linkages are indirect (unless specifically optimised). While this poses no particular difficulty for the compiler beyond a certain amount of additional bookkeeping, it is worth noting that any confusion might be avoided in a fully mechanical manner, since the type signatures of **Visitors** involve $\mathcal{P}$, a type that should not otherwise appear at any level in a well-typed programme. In conjunction with the usual type consistency rules this alone should suffice to enforce the appropriate pragmatic divide.

Common Environment: **Visitors** themselves, at least for anything like a standard type system (and certainly for the type systems discussed in this thesis), turn out to be pure, nice, environmentless functions: *all* of their dynamic context is received as parameters.[18] This strongly suggests the abbreviation of their representation to simple pointers (rather than the more common duplex pointer representation of closures). Ordinarily this would have to be accompanied by the use of autoclosures for the *overridden* range members of a mode function (see section 5.4.3 above), since it approaches certainty that these will manipulate state.

In a significant proportion of applications, however, and in particular in those low-level cases where performance is presumably most important, we find that all the overridden range elements (attainably) originate in the same scope. When this is the case it seems to make sense to allocate their common environment pointer to a register before entry to the recursive visitation nest, thus saving the effort of reloading it repeatedly.

[18]They are not combinators, however, since they do bind constants.

Flattening: As we have seen, there are two distinct classes of application of visitation: those in which the objective is simply unbiased coverage of the storage structures chosen to represent types, used (in a broad sense) internally to the system; and those in which it is used as a user-level facility for data decomposition. While in the latter case the formal internal structure of the data must be preserved in the interests of correctness, in the former there is opportunity for the optimisation, not only of the *representation* of the data, but of its *description*.

Clearly, as with any use of functions, a certain amount of inline expansion of **Visitors** may be desirable. The optimisation is particularly favoured in this case because the function bodies are often very short, and the spatial cost of inlining comparatively minimal. The **flattening** optimisation amounts to no more than hyperaggressive inlining: the replacement of an indirect call to a **Visitor** (as all internal visitation invocations are indirect), possibly even in the case where it is not provable that the particular element has not been updated. This move has the twin objectives of reducing runtime for the visitors and possibly eliminating some elements entirely from the mode function's domain, thereby reclaiming some or all of the space expenditure.

Under normal circumstances this "optimisation" would clearly be unsound, since the observable behaviour of the programme is modified. In the case that the language processor does *not* make guarantees of representational transparency, however, it is correct, because the semantic alteration is not canonical.[19] Note, however, that it specifically must *not* be applied to the base cases, pointers and bits, lest the caller lose the ability to perform a useful computation.[20]

[19]And caveat hacker!

[20]Actually, and fortunately, the distinction between upper- and lower- level visitation is nothing like as clear-cut as we suggest. On the one hand, as earlier pointed out, positive practical benefit might derive from the refusal to guarantee the transparency of the decomposition provided by visitation (just as unordered parameter evaluation may be considered a feature rather than a bug of a language design); and on the other, since the set of pointwise updates on a visitation mode will—at least under our recommendation for the handling of type name scoping—be statically bounded above, it will almost certainly be practical to compute a subdomain of the basic visitation system over which this optimisation can be applied in total safety. Indeed, the optimisation can be applied independently in different cases as a simple case of partial evaluation. How many distinct instanti-

Pre-marking: Finally, although it is not directly consistent with the presentation of **Visitors** we have made in the above, it is worth noting that in applications using traversal marking, the performance of visitation of cyclic objects, both spatially and temporally, is significantly improved by *pre-marking:* scanning each object twice, first to schedule visitation and mark the embedded pointers as attended to, and then again to perform actual transitive visitation. This has the effect of preventing the same "object" from appearing more than once in the stack.

5.6.6 Visitor visitors

Just as normal functions themselves have representations that need describing through **Visitors**, so it is with **Visitors** themselves; and here find concrete manifestation of the paradoxicality of a naïve type of types. The particular difficulties are, first, that the process of visitation (and the progress of any computation so mediated) must in no way incapacitate them,[21] and second, since **Visitors** exhibit the same variety of internally idiosyncratic structure as other object code, some means of visiting it must be found that does *not* require an infinite progression of visitor-visitor-visitors.

Three approaches to this difficulty are known. The first is simply to forbid the involvement of **Visitors** in any visitation process, allocating them in static storage and accessing them only through the intermediary of the type of type names, $\mathcal{T}$. This is completely satisfactory but for the case where types are truly dynamic—though unfortunately, polymorphic binary transp'ıt falls in this class.

Secondly, we might (as already suggested) classify the actual *executable* aspect of **Visitors** as cache bits, now to be regenerated from a formal, abstract description of the code periodically (in particular, whenever it is believed that the cached version has become invalid), care being taken to avoid any deadlock situation in which the

ations of the visitation system are desirable is of course a conventional issue of time/space tradeoff in compilation.

[21]This includes ensuring that **Visitors** are located and preserved through garbage collection, an important issue that we shall not be able to explore here.

cached version is invalidated while it is still in use (or in which it itself is involved in the revalidation process). This need not be anywhere near as expensive as full code generation, of course, since in the usual case the old, invalid version of the function persists as a template for an update function.

Finally, if the host instruction set architecture permits, we might restrict the code generation of **Visitors** so that they never contain any embedded data, being pure binary storage, relocatable and dynamically linked. This forces the locus of the difficulty back to the controlling mode object.

In any event, we find that ultimately there must be a cap to the directly-executed part of the type system, in the form of a fixed virtual machine distinct from the host processor that manipulates data interpretively. The function of **Visitors** is to push this back from the level of the individual data to the types themselves, where the computational overhead is much lower. That it cannot be removed completely is no surprise: something, somewhere, must perform virtualisation, and it must do so from "outside."

5.6.7 Sample physical translation

For the sake of concreteness, let us finally consider the physical implementation of the **Visitor** for the type Node:

```
  TYPE N32 : 0 .. (2 ^ 32) - 1
; TYPE Key : N32
; TYPE Geom : {leaf, internal}

; TYPE :< Node @ (TYPE T, N32 size) >:
    if  Geom geom
          .. (Assume 3 bytes of align padding follow this control field)
    is  internal
    then
          %Node[T, size] left
        , Key guard
        , %Node[T, size] right
    is  leaf
    then
        Vector[Key, size] keys
      , Vector[T, size] values
    end
;
```

The algorithm of section 3.3.1 yields *Exemplar* code similar to the following:

```
.. Raw visitor ..

  𝒫 :< $\beta_{\mathrm{N32}}$ @ (𝒫 p0, Void a, ℳ m) >:
    m[binary][p0, 1.word, m]                    .. N32 is just 1 word long
; 𝒫 :< $\beta_{\mathrm{Key}}$ @ (𝒫 p0, Void a, ℳ m) >:
    m[⌜N32⌝][p0, _, m]
; 𝒫 :< $\beta_{\mathrm{Geom}}$ @ (𝒫 p0, Void a, ℳ m) >:
    m[binary][p0, 1.byte, m]

; 𝒫 :< $\beta_{\mathrm{Node}}$ @ (𝒫 p0, ((𝒯 t, 𝒜[t] a) t, N32 size) a, ℳ m) >:
    𝒫 p1 : m[⌜Geom⌝][p0, _, m] + (1.word − 1.byte)
  ; if  $\delta_{\mathrm{Geom}}$[p0]
    is  internal
    then
        𝒫 p2 : m[⌜%⌝][p1, [⌜Node⌝, [a t t, a t a, a size]], m]
      ; 𝒫 p3 : m[⌜Key⌝][p2, _, m]
      ; m[⌜%⌝][p3, [⌜Node⌝, [a t t, a t a, a size]], m]
    is  leaf
    then
        𝒫 p2 : m[⌜Vector⌝][p1, [⌜Key⌝, _, a size], m]
      ; m[⌜Vector⌝][p2, [a t t, a t a, a size], m]
    end
;
```

We now consider the compilation of the function β_{Node} for a generic 68000-like processor.[22] Naïvely (not in terms of local code generation, but relative to the

[22]By a "generic 68000-like processor" we intend something with the following characteristics:

- a dozen or more geneneral-purpose registers;
- an orthogonal $1\frac{1}{2}$-or-more-address instruction format; and
- a hardware supported call stack on which data may be saved and parameters passed.

Visitor-optimisation issues discussed above), we might generate the following object code.

```
; Naïve translation of $\beta_{\text{Node}}$

; Register assignments:
;
;               rT      Argument passing to deref functions
;               rE      Environment (static link)
;               rS      Stack pointer
;
;               rP      Running pointer, Pi        : $\mathcal{P}$
;               rA      Argument, a                : $\mathcal{A}[\ulcorner T \urcorner]$
;               rM      Mode, m                    : $\mathcal{M}$

vNode           push    rA
                push    rP
                move    [rM + 16 * tGeom + 4], rE
                call    [rM + 16 * tGeom]
                add     rP, 3, rP
                pop     rT
                move    [rM + 16 * tGeom + 12], rE
                call    [rM + 16 * tGeom + 8]
                bran    rT ≠ k_internal, c_leaf

c_internal      move    [rS], rA
                push    [rA + 8]                   ; a size
                push    [rA + 4]                   ; a t a
                push    [rA]                       ; a t t
                push    rS
                push    tNode
                move    rS, rA
                move    [rM + 16 * tPointer + 4], rE
                call    [rM + 16 * tPointer]
                disc    5
                move    [rM + 16 * tKey + 4], rE
                call    [rM + 16 * tKey]
                move    [rS], rA
                push    [rA + 8]                   ; a size
                push    [rA + 4]                   ; a t a
                push    [rA]                       ; a t t
```

The processor of the example actually differs crucially from the 68000 in that immediate-mode-convention branch target addressing is used: the operands of jump, call and branch instructions are assumed to give the *addresses* of target routines, not the instruction sequences themselves.

```
                push    rS
                push    tNode
                move    rS, rA
                move    [rM + 16 * tPointer + 4], rE
                call    [rM + 16 * tPointer]
                jump    c_end

c_leaf          move    [rS], rA
                push    [rA + 8]                ; a size
                allo    1                       ; vKey ignores rA
                push    tKey
                move    rS, rA
                move    [rM + 16 * tVector + 4], rE
                call    [rM + 16 * tVector]
                disc    3
                move    [rS], rA
                push    [rA + 8]                ; a size
                push    [rA + 4]                ; a t a
                push    [rA]                    ; a t t
                move    rS, rA
                move    [rM + 16 * tVector + 4], rE
                call    [rM + 16 * tVector]
                disc    3

c_end           disc    1
                retu
```

Substantial improvements are possible, however. Applying the optimisations we have discussed and allocating only one more register, we can halve the size of vNode, reduce its nominal memory reference rate by a further factor of two, and (on the assumption that the global contextual condition is met that the **Visitors** for Key and Geom are overridden nowhere in the application), eliminate its reliance on sub-**Visitors** and the associated linkage costs.

```
; After the following optimisations:
;
;                Closure elimination
;                Tail call elimination
;                Inlining (only valid in certain application contexts)
;                Fixed-pattern global register allocation
;                 Encoding ⌜T⌝ directly as β_T

; Register assignments:
;
;                 rT       Deref, vPointer argument passing
;                 rS       Stack pointer
;
;                 rP       Running pointer, Pi        : 𝒫
;                 rA       Argument, a                : 𝒜[⌜T⌝]
;                 rM       Mode, m                    : ℳ
;
;                 rBinary  Constant                   = m[binary]
;                 rPointer Constant                   = m[pointer]

vNode             push     rA
                  move     1, rA
                  call     rBinary
                  add      rP, 3, rP
                  move     [rP - 4], rT
                  bran     rT ≠ k_internal, c_leaf

c_internal        move     [rS], rT
                  move     tNode, rA
                  call     rPointer
                  move     4, rA
                  call     rBinary
                  pop      rT
                  move     tNode, rA
                  jump     rPointer

c_leaf            move     [rS], rA
                  move     [rA + 8], rA
                  mult     rA, 4, rA
                  call     rBinary
                  pop      rA
                  jump     vVector
```

In this example we acheive the following local code quality improvements (in fact we do considerably better if we take account of the total elimination of calls to β_{Geom}, β_{Pointer}, β_{Key} and β_{N32}).

Old/Optimised	internal	leaf	static
instructions	33/14	27/12	49/20
data bus transactions	37/7	27/7	54/11

5.7 Interfaces

At various points in this thesis we have made the observation that the representation of a type as a structure in memory is not unique: not only may the programmer encode the same idea in an infinitude of different ways, ways that the unassisted compiler may well be at a loss to recognise as related, but as with any part of its task efficiency is best served by allowing the compiler itself to choose whatever representation it will for an object described. Before concluding this chapter, let us investigate, briefly, some of the implications—and some of the limits—of this observation.

The present section is entitled "interfaces," and the reason is as follows: that the representations of objects are only visible at the **boundaries** of modules. Of course, this is somewhat tautological, since the notion of a "boundary" is imprecise and can only really be understood in terms of the places where objects become visible to external systems; in particular, the same system may have or lack a particular boundary depending on its actual circumstances. When run under a debugger, for instance, a programme acquires an interface between almost every part of itself and the outside world: all that is normally private is brought out for the programmer's inspection. The consequences are both familiar and extreme—unless a programme has been specifically prepared for debugging, most source-level debuggers are severely constrained; and unless the compiler was instructed to be especially simpleminded (through the disabling of "optimisation") most debuggers can become impressively confused. In short, if every source statement and source type must be considered to be at an interface, the compiler must take special pains to follow its directions to the letter, not pandering to its own perceptions of intent.

5.7.1 User interface: print forms

One of the respects in which richly typed languages and unitypic languages in the mould of LISP typically differ is that the former (not having complete *a priori* information about the range of types they might be required to manipulate) leave much more of the task of providing meaningful, humanly interpretable output to the programmer; while the unitypic languages provide printable forms for many if not all varieties of data.[23]

Such print forms, intended primarily for human consumption, are not necessarily exact representations of the data from which they are generated, but must be sufficiently suggestive that they are of practical use. Conversely they must have broad enough range (though in principle just the ability to put out an arbitrary, specified character will suffice[24]) to represent whatever information the user does in fact need; the exact facility provided impacts little besides programmer convenience and the abstract ends of good engineering. Similar observations apply to input.[25]

Wherever automatic formatting of data for transput is provided, this makes an interface between the user and (the type of) the object transput: the object is laid open for inspection. Except where there is explicit provision for indeterminacy of format, the results must be predictable; in particular, though the language processor has free choice of internal data representation, whatever choices are made at that level must be "normalised out" during output formatting and similarly compensated for during input.

Since all interaction with the user (even string transput, if the process be examined in detail) will involve rerepresentation, this presents no large technical difficulty; it

[23]In this section we discuss *textual* transput representations, based on the traditional media of the keyboard and paper- or screen- imaged text; but all our observations apply *mutatis mutandis* to the full range of human interface technology. This should not be regarded as evidence of textism on the author's part, a failing of which we are, the reader is assured, unquestionably innocent.

[24]This character presumably being drawn from whatever character set the host system holds most dear.

[25]It is often said that input is "untyped" because there is no control over what the user may type. Of course, this is quite untrue: since the user is provided with only a keyboard (or a mouse, if the software cares to interrogate it—the set of input devices is somewhat arbitrary but strictly finite), input has the type of key-press-events *ab initio* and this type will be restricted further as the data pass through validation and re-representation code. The only form of input type error that a user can really commit in a system that is type-secure internally is of the stuff of humour: reprogramming the computer with a very large axe [Ada79], perhaps, or spooning ice cream right into the slots [Jac91, *s.v* delusions, 15-point]....

means only that the details of the process must depend on the actual *structure* of the data at the point where transput is requested, not merely on its *type*. This, in turn, may be accomplished either by specialising the transput code or by *prenormalising* the data to some standard form (perhaps the natural representation—the single most common nontrivial restriction on what compound objects are automatically formatted for output being essentially the existence of a natural representation). The former mirrors what a programmer might do in support of a particular, explicitly chosen representation of some type; the latter is more in line with the approaches we will shortly suggest for interfaces to other software. One might say that the distinction conceptually rests on whether the transput system is regarded as within or without the language itself. But in either event, the user sees a uniform external representation, independent of the underlying details.

5.7.2 Lateral interfaces

Where the textual transput system is viewed, not as part of the language system to be adapted to the programme as necessary, but as an autonomous entity to which some standard face must be presented, we have an example of a **lateral interface**: an interface between two peer subsystems of the electronic world.

Since software lacks the adaptability of a human user, lateral interfaces must be even more tightly specified than print forms for data; but because they often operate at higher bandwidth and connect entities with greater patience for "administrivia" there is (theoretically, at least) more scope for negotiation in their implementation. They can be roughly classified according to the time at which such negotiation takes place: it is another of the *binding time* issues so common in the analysis of programming language.

The earliest possible binding time for the "notation" of a lateral interface results in the situation most similar to that of interface with a human user: the interface here constitutes a prior standard to which both parties must adhere. Similar in nature is the situation of the "*de facto* standard," wherein the interface is designed solely for the convenience of whichever component was written first, its counterpart conforming to whatever that might be (which, all too commonly, is specified nowhere but in the behaviour of the first partner).

These situations share with human interface a rigidity in their implementation: since at least one of the partners in the interface was not party to the original negotiation, a fixed the external format is simply computed from the in-memory representation and employed directly. The common format can be expected to be fairly arbitrary, but since it implements the same type as the internal representations, appropriate mappings should exist. Whether an implementation can derive them automatically is another question: in fact, as with data formats for human use, it is to be expected that, unless the representation is derived from a broader standard [niso91], code will have to be written for the purpose. One exception is when what is convenient for one party is convenient for the other: this is the case when a traditional programming language that uses very transparent data representations is used to implement both sides of a suitably intimate connection.

The situation is different when the interface is arrived at jointly by the two partners. This normally happens, for instance, between functions in the same compilation unit: to a certain extent, each can accommodate the other since they are compiled more or less at once. It is less obvious that this is an option even between source languages or across module boundaries, but in fact a sufficiently aggressive implementation of a compiler suite sharing a common back end can maintain a database of data formats which is updated constantly by the compilations contributing to an overall application or environment. In this context we now expect the whole burden of generation of appropriate interface code to be assumed by the compiler, since it has been pushed back beyond the start of compilation and down below the level of source code.

The extreme case of late binding is where the format in which data are exchanged is established dynamically. It is obviously not possible for such a thing to be done in the complete prior absence of shared information, since the two partners must at least share a common language of negotiation; in particular there must be a preëstablished notation in which information about type[26] and implementation can be conveyed. But if such common meta-ground exists, it provides the possibility of communication between potentially disparate representations of the same type while providing asymptotically optimal performance in the case where the representations used by the partners chance to be well-matched (an important property in operating systems applications such as remote procedure call). (On the author's suggestion, Samuel A. Rebelsky of the University of Chicago has adopted just such a scheme for his lazy term transmission protocol [Reb91].) The high asymptotic performance can be achieved, of course, because once an interface *has* been negotiated it can be adhered to implicitly for the remainder of the connection's existence. Such a system is described at greater length in the next chapter.

5.7.3 Downward interfaces

The final direction in which interfaces must be made is **downward** to the underlying system. Communication with the underlying hardware or to a "hostile" host operating system is characterised by a *hyperprecise* interface in which data representations and protocols are specified in more detail than the source language is likely to permit of expression.[27] Although some systems programming languages get by on the basis of making explicit promises about the representations they employ for their various types, it is ultimately necessary either to provide an extensive facility for the description of physical format, or to attack the problem on a completely different

[26]Or metatype or metametatype or....

[27]In fact, this same problem can arise even when communicating with the user when, for instance, the evaluation order of the programming language is underspecified.

level.

The classical alternate attack on the problem is to "escape" from the source language into assembly code, programming the necessary interfaces explicitly, and reducing the interfacing problem from the main source language's perspective to that of providing a well-documented lateral interface to this other source language (the assembler), and in a manner that it is free to dictate. It is then essentially bound only to be consistent; all further difficulties are laid as a burden on the programmer.

A hypothetical but more sophisticated approach might be to permit the programmer to intervene in the compiler's normal selection of data representations, by communicating with the middle- or back-end directly. This is similar to suggesting that the programmer adopt the compiler's intermediate code as the assembly language of choice for the lateralising strategy; the difference, of course, is that there is more opportunity to enlist the compiler's coöperation, and less work may have to be done by the human programmer. In principle the compiler can even retain the ability to type-check the programme as a whole (something that is essentially never possible when part of it is written in assembly code). The price is paid in flexibility: if the target architecture has features that are not accessible through the code generator, for instance, the strategy may fail. While interesting and promising on paper, it may be that the additional demands this scheme places on the internal structure of the compiler are unsatisfiable.

In any event, where such explicit low-level interfaces occur, it is often the case that data must be represented in some declared manner, not influenced by the compiler's preference. To return to our theme of visitation, **Visitors** may still be generated for such objects, identifying fields and visiting each in turn, but a number of restrictions are likely to apply. The first is that the requisite physical representations even of subfields may diverge from the standards of the language to the extent that they defy description except as raw storage. Thus even superficially decomposable objects may be easiest reported as as no more than data bits. Second, any pointers may themselves be strangely represented or subject to unusual restrictions (such as an absolute prohibition against "stealing" one of their bits as a traversal marker lest something untoward and hardware-like occur). Either of these irregularities may in principle be mitigated by the same strategies that we proposed for **Visitors** themselves: removing them from the domain of visitation or providing that the **Visitor** see a regularised image of the structure of, shall we say, differing canonicity. A third trouble may be that there is, indeed, no place to *put* any cache information that may be needed in the interpretation of the object or as part of a forced regularisation strategy, since one of the main ways in which an object may have a restricted representation is having a static (and "magic") address. In the simple case a duplex pointer will assuage this concern, one branch designating the real object and one its formal image, but if the object's address is itself derived from an object of stated representation we end up with an inductive structure completely parallel to the **Visitor** itself. The type system has effectively ended up with an additional layer, but so long as the appropriate steps are taken to keep everything in synchronisation (which may be nontrivial, but not impossible) visitation still works as usual.

5.8 Conclusion

Our basic position is that strong, static typing is essential to the construction of reliable software of any scale greater than that attainable by a single programmer in a single session. But as we have seen in this chapter, a firm commitment to strong typing pays its dues at compiler development time; if the structures described above are compared with more "usual" runtime representations it will be seen that they show clear advantages in both time and space. It is a well-established fact that the runtime efficiency of code is in the common case enhanced by the additional information that a type system makes available to the compiler, and a more sophisticated approach to typing makes more aggressive optimisations practical. At the same time, as we have tried to show, strong typing need not result in any semantic impoverishment of the language: not only are the facilities normally associated with unitypic languages potentially available within a strongly and statically typed framework, but types, as a device for making facts about intended redundancy explicit to the translator, can serve to *reduce* external notational overhead in the mapping between the problem domain and the final programme, in the same way that it can often eliminate the internal representational overhead of explicitly tagged data.

In the next two chapters we exhibit essential programming facilities that can be provided gratis in the typed context, but which cannot otherwise be obtained without significant programmer investment. Through the mechanism of **Visitors**, furthermore, they will be able to attain the performance levels of compiled code, in contrast to the interpretive behaviour of the ancestors of these facilities in unitypic languages.

In this manner we hope to demonstrate that the cost of adoption of a strongly, statically and expressively typed system is, in the present state of the art, a decision with negative expected cost.

Chapter 6

Applications

Having now examined **Visitors** in both their practical and theoretical aspects, we turn our attention to their application. Some uses apply fully at the user level, while others are transparent to the programmer; most lie in between, producing conventional and familiar effects in a semi-automatic and "tedium-reduced" fashion. As will be seen, however, they share a common theme: they provide facilities that are traditionally provided for built-in, "system" types, but indiscriminately to those types provided by the system and those defined by the user, in many cases at a level of efficiency that reaches or exceeds that of conventional dedicated strategies.

Not all of the applications described in this chapter have actually been implemented; several of them are derived from the author's **Mythos** operating system project, which is still in the paper design phase. Among the the main intellectual objectives of **Mythos** are the identification of places where higher levels of abstraction in design and implementation, can result in improved practical performance. More practically, we hope to demonstrate that aggressive dynamic architecture independence can be accomplished at negligible overall cost. **Visitors** are a cornerstone of this undertaking.

6.1 Garbage Collection

The application that originally motivated the development of **Visitors** is *garbage collection:* the provision of unbounded virtual space to software with limited momentwise storage occupancy through transparent analysis of actual memory use (and possible reconfiguration of the computation within its address space).

This task is by definition invisible to the user programme,[1] and the function of the **Visitor** in this application is twofold: to increase the modularity of the garbage collection code and to provide a general and efficient mechanism for supporting *tagless* data representations in the heap. Tagless representations are almost always more compact than their tagged counterparts[2] and permit straightforward implementation of richer object semantics, but their types can only be established by the kind of inductive argument that **Visitors** encapsulate.

Since this application of visitation was the first and remains the most heavily investigated, we have chosen it as the subject of a more detailed investigation; this is presented (along with the requisite background material about garbage collection in general) in the next chapter.

6.2 Binary Transput

Intimately related to the garbage collection problem is **binary transput**, the storage and transmission of computational terms in a format that is both efficiently processed by the machine and as semantically complete as possible. The similarity to garbage collection rests in the requirement of identifying (and in this case moving) all of the

[1]There are slight exceptions to this in the case of data types that have intimate connections with physical addressing; or which are capable of adjusting their space requirements in response to overall memory availability; but these are the funky exception rather than the rule, and while we have some initial experience with them, we have not yet devised a clean, general interface for their support.

[2]Specifically, they consume space proportional to the number of distinct representations rather than to the total number of objects, and an advantage is thus to be expected where object layouts are multiply instantiated. Typically, this condition only fails to obtain for structures—like code—explicitly provided by the programmer.

storage associated with a value while entraining as little extraneous data as possible; and of processing it without semantic damage.

As we shall see, **Visitors** permit us to perform that task without requiring the coding of transput routines for any more than a covering set of types that are atomic for the purposes of the transmission channel; while actually increasing the flexibility and generality of the system as a whole. Best of all, **Visitors** are independently motivated and can be generated automatically.

6.2.1 Naïve bulk transmission

To a first approximation, then, the task of performing binary output is just that of traversing the term, writing its leaves serially to some external medium; a task to which the **Visitor** is ideally adapted. The root **Visitor** for data bits is replaced by a binary output routine, while pointer visitation is either left untouched and transparent (if the data to be output are known to be tree-structured and free from circularity—graphiness—and other sharing—DAGginess), or by a function that detects sharing and for each pointer traversed writes a mark indicating whether to refer to some other object in the file (and which) or whether new data follow.

Input is handled similarly, by a visitation of the area to hold the input term with the most primitive **Visitors** replaced by binary *input* routines. Since we take care (as previously noted) in the construction of **Visitors** to delay the examination of controlling fields until *after* their traversal, they will already have been read in and installed, and structural analysis of the partly transferred datum will be accurate. Traversal of pointers must of course be made to allocate space for the referent in the case that it is not identified as a pre-existing object (and as is necessarily the case when it has been decided not to represent pointers externally).

One point worthy of slight note is the handling of *cache bits:* since, as we have seen, cache bits are used to avoid the performance of arbitrary computations during structure manipulation while avoiding substantive restrictions on their nominal types, the most straightforward approach to binary transput requires that they be treated as any other data. This yields good throughput and minimises (computational) resource consumption in the **Visitor**, but it does so at the expense of space: the resulting external representation will not be the densest possible. If denser representation is a goal, one could imagine a slight modification of the visitation system set forth in the bulk of this thesis, in which the argument packet for the cache bits **Visitor** takes as an additional argument the actual code which performs the calculation redundant with the content of the cache bits field. This could then be invoked by a consumer routine to reïnitialise the cached field, lifting from the producer the burden of saving it and from the channel that of its transmission. In fact, this is our preferred approach.

6.2.2 Typed external representation

Several degrees of precision of binary representation are possible: it can be arranged that a term written to a file (or other channel) be interpretable later in the *same* execution of the writing programme; that it be interpretable by a *later* execution of the same programme; that it be interpretable by another programme having a prior *contract* with the term's writer (in the form of a shared type definition for the term, perhaps) and running under the *same* language processor, on the same machine; similarly, but for a *different* physical node of the same architecture; similarly again, but with thorough independence from architecture; or, finally, without even the constraint of a prior contract. A dogmatic (and, we would argue, justified) interpretation of the ideals of strong typing and well-defined interface would in principle support all options but the last; indeed, in the absence of a prior contract it is hard to see how even a thoroughly self-describing term could be transmitted, for how then would the description be interpreted? We will therefore assume that at least the *channel* is typed.

On the above scale of possibilities, each entry is, from a technical perspective, successively harder to achieve. Traditionally, each step up the scale has been tackled by moving to a more symbolic representation for the data, replacing objects lacking stable external representations with symbolic references to peer structures, ultimately standardising external representations of atomic data across all platforms and times and tagging them explicitly where options are available (the clearest example of this is perhaps ASN.1 [iso87]). This is manifestly far from optimal in any empirical respect.

If exact **Visitors** are employed—ones that have not undergone structure-modifying transformational optimisations (and that correspond to similarly transparently represented types)—and if the chosen external data representation corresponds to some straightforward traversal of the term, visitation can similarly be used in service of these more abstract binary representations; it is now necessary to provide a semantic routine for each type for which a platform-independent representation is mandated, and to ensure that these cover at least those parts of the structure whose preservation is required.

We can improve on the standard approach significantly, however, in three respects. The first is that (at least if we adopt the earlier suggestion of treating the generation of native code from abstract, machine-independent intermediate code as a dynamic optimisation process resulting in cache bits rather than data) we are in the position of being able to provide stable external representations for a far higher proportion of objects than more conventional approaches would permit, including, in particular, executable code.

Secondly, **Visitors** themselves constitute a type description format that (given only transput of code!) is itself conveniently transmissible in like manner.[3] This vastly

[3]The all-crucial linkage with peer structures in the receiver requires only the establishment of "pretransmitted" references to the basic underlying type vocabulary (though difficulties can still be expected with more pragmatically motivated types, as the lore of floating point cross-compilation well attests). In general, however, such identification of homologous data is the limiting factor on the accuracy of external representations, since it is not clear that such identifications can extend beyond the lifetime of the nametable in which they are recorded.

improves the external representability of generic data: where conventional schemes require either very rigid frameworks or item-by-item tagged data, if they can handle the situation at all, we can transmit the **Visitors** (in their capacity as runtime type representations for such data) as they are needed, with subsequent reference to them being handled in the same manner as other shared data. In fact, close examination of the mechanism given above will reveal that this behaviour arises automatically in the course of visitation, without any particular additional effort on the part of the implementor (when once the overall format of **Visitors** has been designed correctly).

Third, we can in fact circumvent the need for a universal standard external data format to a certain extent. One of the primary sources of inefficiency in highly portable external data formats is the cost of data rerepresentation into a standard, linear, platform-independent external format, and its subsequent unpacking by the final addressee—the so-called "marshalling" and "unmarshalling" costs. Since the *usual* case for binary transput is that the receiving process is running on a machine that is at least similar (if not identical) to that under the sender, it is a coherent optimisation to defer such format conversion, putting it off until the binding of the destination (not immediate if the transmission medium is passive, of course), if not until actual reception.[4] When the system is coded in terms of **Visitors**, this requires only that a (covering) set of input routines be generated on the receiver's side using the the *inverse* of the foreign conventions when accessing memory. These can plausibly be generated automatically from the same intermediate code as is used more straightforwardly for the homogeneous case, by making small adjustments to the parameters of a suitable retargetable code generator.[5] Such a technique is (once again) operable because the examination of stored control fields *follows* their restoration to

[4]Thus a small amount of additional *type* information is associated with the channel: it encodes such architectural parameters as bytesex, wordsize, characterset and floating point format. Obviously strong typing is essential if this information is to be coherently applied.

[5]This motivates our characterisation of the operation as the use of an *inverse* transformation in memory access rather than a *direct* transformation in the physical input operation: the latter would be considerably harder to implement automatically.

the appropriate internal format.

In sum, we find that not only do **Visitors** provide a far more *automatic* approach to the construction of binary transput facilities than has hitherto been usual, but they in fact permit a broader range of capabilities, with potential for higher (and more tunable) performance along a number of axes.

6.2.3 Lazy term transit and remote procedure call

Fully transparent remote procedure call across heterogeneous platforms has not hitherto been possible, because of the absence of a sufficiently general technique for parameter handling. One aspect of this difficulty is the *description* of parameters in a portable manner; **Visitors** in their strict interpretation (when applied to untransformed representations, that is, and under the assumption that any code visited is seen in its abstract form rather than the locally compiled object) go a long way towards addressing this—for a strongly typed *operating system*, achieving it completely. In fact, since (on a theoretical analysis if not in the notation of many programming languages) both argument lists and return values (return lists, in the rare case where these are provided) are single data structures, it is only the mechanics of the channel that distinguish this case from that of binary transput as just described.

Equally pressing, however, is the need for efficient transmission of large structures between remote address spaces; indeed, it might be argued that traditional inattention to mechanisms for describing arbitrary parameter types to remote procedures has been *motivated* by the absence of an efficient means of communicating them, were they describable. For consider the case of a linked list passed as an argument to a remote function: we may well be able to employ visitation to transmit (a copy of) the entire term to the remote process; but this involves unbounded communication, and it is not hard to imagine operations in which the majority of this effort is wasted (as when the remote function need only examine the first element of the list to complete).

While this could be addressed through modification of the protocol or annotation of the interface, these increase the level of human intervention necessary to derive the interface specification, and decrease the opacity of the remote service.

A partial solution is offered by the idea of lazy term transit [Reb91]: data can be passed a node at a time, with pointers being replaced on the receiving side by *promises* of data (ideally transparently through the manipulation of the virtual memory map). With this idea we can combine the advantages of remote shared virtual memory [Li88] on an *object* level and the tagless typèdness of visitational binary transput. Now, though, we suffer from the inverse difficulty: the data to be transmitted have been fragmented into their component nodes, which while of very small average size are nonetheless potentially unbounded (as in the case of a monolithic array, for instance). Clearly, some happy medium is needed.

Such a compromise is to be found in the observation (commonly made in operating system design) that there is usually an *a priori* optimum length for data packets in (any implementation of) a protocol, often one determined by the characteristics of the underlying trans: ·t system. An amount of *read-ahead* that will fill out the packet (or the page) is thus for practical purposes free (similarly, large objects can be truncated at such boundaries—being effectively paged—provided the necessary adjustments are made between architectures whose page boundaries cannot be made to coïncide).[6] Experimental results from the garbage collection and virtual memory literature [Wil91] suggest that an appropriate traversal order is breadth-first within each packet (and directly demand-driven thereafter). While this is not the natural traversal generated by visitation, the buffering of subvisitation demands in a queue can clearly be applied through a small modification to the usual handling of pointers;

[6]This approach originally arose in conversation with Samuel Rebelsky. Since Rebelsky's *term tour interface* [Reb91] also serves as a gateway between disparate execution domains (in particular interfacing eager and lazy reduction order processors) it does not appear that this optimisation can be applied without explicit casewise justification by static strictness analysis or dynamic checks for subterm grounding.

and this can be done in the client code, without change to the visitation system itself.

The single greatest novel complication introduced by the above (relative to conventional remote procedure call facilities) is that we are here attempting to pass data to a remote procedure by *reference* rather than just by value. Superficially, the implementation just described will actually provide an erratic hybrid between copying and non-copying semantics (though this will be detectable only if the caller is multithreaded or the remote service is provided with remote callbacks). This can be tightened by defensively invalidating local pages containing objects actually transmitted to the remote procedure and applying whatever synchronisation or atomicity policy has been specified for the remote interface as a whole at the moment that a potential conflict is detected. At the worst, absolute synchrony can be enforced by reclaiming the transmitted data to its page of origin on the calling side. This amounts to making the lazy remote copying mechanism simultaneously bidirectional, with the advantage that it unifies the transfer of arguments and results (though it must be admitted that a difficult distributed garbage collection problem ensues). As always with remote invocation, it is essential to performance to choose the weakest synchronisation model consistent with the application.

6.2.4 Typed memory virtualisation

Several advantages potentially devolve from making type information available to the virtual memory system. One is that quite aggressive compression can be employed in paging at comparatively low cost when type information is available [Wil91]; clearly we can do this as more or less a direct application of the technique discussed in the previous section—for a backing store may have an unusual call interface, being invoked implicitly on the process's behalf by a page fault handler, but it is easily seen as one of the most trivial of possible remote services.

Another conceptually straightforward possibility is *inter-architectural* remote shared memory, wherein both format conversion and data transmission are handled transparently through visitation and and an underlying communication channel. In fact this is no more than a special case of the remote procedure call interface sketched above: in a typed system, virtual shared memory is effectively identical with the combination of strict synchronisation across the call interface, a multithreaded client, and parameter passing by reference.

6.3 Formatted Transput

While the bulk of this thesis deals with data in one or another machine-readable representation, their most familiar form (at least to computing "civilians") is certainly their external, *printable* representation. In the previous chapter we discussed the problem of ensuring that data to be read or written were converted to a canonical *shape* for communication with the user, but did not address how **Visitors** apply to the actual parsing and imaging of data once a standard form is reached.

Much of the effort of communicating with the user is the inalienable responsibility of special-purpose code that knows the printable representation of the various data considered atomic by the language. Even in the case of compound data, the reading and writing of the appropriate brackets and separators must often remain the domain of the application. **Visitors** assist the undertaking, however, in three regards.

First, transput of data structures generally involves a structural induction, and it is in the mechanics of induction that **Visitors** shine. Under certain assumptions about the manner in which visitation is made available to the programmer, much of the work of case analysis, traversal, and the handling of circularities stands to be automated through the installation of the various transput primitives as entries in a visitation mode.

Second, the handling of "semistandard" types (such as might be informally described as "a kind of vector" or "a linked list of stuff") can be simplified, because the externalisation of the dispatch mechanism and the inherent polymorphism of visitation permit the ready installation of specialisations of generic handlers for such cases in the transput modes.

Finally, when it comes to the printing of the unprintable, the simple provision of a covering set of external notations for the primitive varieties of storage ensures that *some* kind of external representation for data can be cooked up automatically, and if necessary it can be made a faithful representation of the value in memory.

6.4 Debugging and Abstract Editing

A more exciting application of the same technique of using visitation to assist in the construction of human-readable output is to debugging and abstract editing. Where it is necessary to manipulate the "official" source representation of data, the **Visitor** is essentially restricted to providing infrastructural support, but when it comes to providing *some* kind of representation for *arbitrary* data it is far more visibly useful.

In particular, we observe that **Visitors** are in large part redundant with the information stored in classical debug records for the explanation of runtime structures to interactive debuggers, but they are far more direct and general: while their correlation with structures in the original source code may just as easily be obscured by optimisation processes as may that of conventional debug information, in the case of **Visitors** we can make a claim of completeness of representation that is not classically possible. Using the **Visitor** for the programme state we can completely decode all structures accessible to the computation, complete with whatever structural information has actually been preserved.

The most effective manner of translating information back and forth between internal and editable displays is through a process of lazy synchronisation employing back-to-back **Visitors**, one traversing the subject data structure and one a standard descriptor that in turn serves as input to the imaging logic.

In the presence of standard-format code, the possibility also exists of (fairly transparently[7]) executing unoptimised code in interpretive mode.

6.5 User Code

Although it has its share of technical difficulties, one important place to put the power of the **Visitor** is directly in the hands of the user. Many programmes exhibit identifiable abstract data types which can be viewed as (possibly among many other things) expressions subject to one or more interpretation disciplines. If we take a broad enough sense for "expression" (and within the present paradigm we are free to) this is true of any well-founded data structure subjected to structural induction.

For such applications we can generate **Visitors** as fairly normal user-level functions: in essence we supplement the more conventional automatically-provided eliminators or patterns for the decomposition of the user's types with a primitive global interpretation operator that analyses it into the components from which it was built. This has the immediate effect of extending the standard functional operators like map and reduce to all the "classical" data structures.

There are two difficulties with this application: the first is the restriction to *well founded* data, as already noted, which arises from the difficulty (at least in most languages) of establishing a clean computational vocabulary for computations involving fixed-points.

Second, it is not clear what the ideal source syntax for these **Visitors** would be. The most obvious solutions are a single fixed function that decomposes an object according to the lexical form of the type declaration (whatever was compound is compound, whatever was referred to by a single name is considered atomic)—an appropriate notation might be to apply the substitute operators directly to the datum to be eliminated—but this has the drawback of being somewhat arbitrary and

[7]The tricky part being the changeover between the two modes of execution, something easily accomplished with a rollback debugger but difficult without.

inflexible. Alternatively, an explicit *inductive case statement* of some kind where the programmer has control over the effective atomicity of substructures in the form of the labeling of the branches might be provided; but such a construction would appear to have excessive syntactic weight for the simple task it performs.

Notwithstanding these minor drawbacks, the automatic provision of **Visitors** for types as they are declared would be a powerful technique and provide strong support for a type-based functional programming style capturing much of the flexibility of the object oriented paradigm and the transparent efficiency of C, in a package with tight control over both scope and semantic coherence.

6.6 Conclusion

In this chapter we have shown that **Visitors** provide a general-purpose software facility potentially important in three distinct classes of application. First, they form the provide a unified and efficient backbone for a number of important facilities below the normal level of attention of the programmer, including several varieties of both transparent interprocess communication and storage virtualisation. Second, they provide a level of infrastructure for common interface tasks at both textual and binary levels, and with various degrees of interactivity, providing little new functionality (aside, perhaps, from cleaner polymorphism in a strongly typed environment) but a measurable reduction in programming tedium. Finally, at the user level, they provide a straightforward facility for the efficient *global* manipulation of arbitrary data, while still enforcing a reasonable level of type opacity.

In all of these applications, **Visitors** serve to break down the divide between primitive and defined objects, furthering the goals of linguistic orthogonality, modularity and opacity. In providing a new, higher-level approach to some recurring problems it reduces source code size, encourages code sharing and should ultimately increase programmer productivity. Finally, their use exposes the inductive structure of certain computations directly to the compiler, with potential benefits to optimisation.

In the next chapter we investigate garbage collection, the lowest-level of all out applications, in detail.

Chapter 7

Garbage Collection in Detail[1]

In this chapter we treat in some detail of the main application to which **Visitors** have in practise been put. This application is that of garbage collection, and in order that the issues be clear, we start by discussing the problem of garbage collection in general.

One of the main objectives of programming language development is to reduce the amount of overhead imposed on the programmer by issues that are not directly relevant to the solution of the problem at hand. Not only does this reduce the amount of code that must ultimately be written (though there is by now a large body of accumulated evidence that this alone will repay a heavy investment in both language design and implementation), but it serves to reduce the tendency for the underlying algorithms to be obscured by the ropes and nails of the programmer's craft. "In the large," furthermore, it is essential to limit the amount of "convention"—programme-external cultural information—that must be understood in order to engage supposedly opaque interfaces appropriately. After notational issues have been addressed, one of the most significant contributions that a language can make to this cause is the automatic handling of storage management, since complex issues of ownership otherwise arise wherever a structure must be passed between different modules.

Much of the literature of garbage collection is based either on the severely im-

[1]A fairly detailed design for a garbage collector supporting multiple languages and employing an early version of **Visitors** (therein "**markers**") can be found in [Spa86].

poverished unitypic or "untyped" systems[2] in the tradition of such languages as LISP and Smalltalk-80, or, recently, on the pessimistic assumptions imposed by the use of the lax type systems found in C and its descendants or (typical implementations of) Pascal. The former approach can be seen as buying easy garbage collection at the expense of running the user's code in a partially interpreted manner (since the type information, at least, is decoded by looking up enumerated tags in tables, and is, except where special local efforts have been by the compiler, basically ineliminable), while the latter necessarily compromises the effectiveness of the garbage collector (garbage collection becomes imprecise) and places a burden on the programmer to follow restrictive programming guidelines with great care.

Although, technically speaking, the mechanics of the approach in this thesis could be applied *mutatis mutandis* to an imprecise garbage collector in order to improve its precision, this would require either a heavy investment in static analysis or a very involved and expensive runtime (in this case, garbage-collection-time) mechanism. The situation is thoroughly comparable with the parsing of simple context-free grammars: the "bottom-up" approaches (to which such information-poor techniques are—like string parsers that must cope with fragmented input—condemned) are either chartish or require extensive LR pre-analysis; while LL grammars translate neatly into executable code. Our mechanism is intended primarily for application in the context of a strongly, statically typed language, wherein data structure layout is unambiguously determined (and only in this context is it likely to be useful for applications beyond garbage collection, which continue to require precise analysis).

In the context of strong (and thus precise) typing, our technique provides something of the best of both worlds: data are untagged and directly accessed, potentially without overhead relative to naïve representations;[3] yet garbage collection can be car-

[2]See section 4.3 for an explanation of the inverted commas.

[3]Whether it is possible to deliver on this promise will depend on the precise details of the garbage collection algorithm selected—and whether the type system permits objects of great complexity.

ried out as efficiently (sometimes slightly more so, sometimes slightly less, depending on the exact details of memory and CPU systems) as with unmodified classical algorithms. As we saw in the previous chapter, furthermore, the investment in garbage collector support here contemplated has further payoffs in other applications relying on global properties of data structures. Relative to tagged or reference-counted systems, on the other hand, we free the compiler from the burden of manipulating data with constrained representations that fragment its knowledge of its task and limit its options at the tactical level.

7.1 The Garbage Collection Problem

The approach taken to storage reclamation by a garbage collection system[4] is that while data structures are created according to the text of the programme (modulo the results of any static optimisations that the compiler might perform), their deällocation is the responsibility of a separate storage recycling module, the garbage collector. The garbage collector has very limited interaction with the client programme; typically it imposes a requirement on the client to obey exceptionally precisely defined linkage and register allocation conventions (to some extent we are able to relax these restrictions, providing greater latitude for optimisation, as shall be seen below), but only enters the control flow graph at points where storage allocation takes place, to handle the (unusual) case where an allocation cannot be satisfied directly out of already available memory.[5] Thus, while obviously the decision to compute a value should be considered

[4]In contrast to reference counting systems, where storage recycling and client code are mingled.

[5]In the best optimised implementations (see, for instance, [AL91]) it may not make an explicit appearance in the control flow graph even at this point, being handled as a recoverable exception, much after the style of stack extension in conventional virtual memory systems. This does, of course, impose restrictions on the acceptable use of registers at storage allocation points comparable to those in effect at callpoints in traditional garbage collectors; but these same restrictions would be in force across any explicit call to a garbage collector, and the net performance gain from making the code (in the usual case) branch-free is substantial. Typical branch penalties are of several base instruction times (and many machines additionally predict forward branches not-taken—which cannot be corrected through the obvious code-reördering without making loops branch forward and thus, typically, mispredict). Particularly in systems with high allocation bandwidth, and on platforms like the Mach operating system kernel [ABB+86] where user page fault handlers are straightforwardly

entirely the concern of the programmer(!), the allocation of storage to hold it is made an implicit part of its construction, and its return to the global pool is removed from the programmer's purview.[6] The worst that can be said of this approach is probably that it introduces an artificial symmetry between heap-allocated and register- (or stack-) allocated values.

In particular, unlike reference counting techniques, in a garbage-collected system the compiler does not classically emit any code structures to support storage reclamation: this is removed from the domain of the client programme altogether. We modify this paradigm slightly in that we generate no *synchronous* code structures to support reclamation, though code is generated to which the storage reclamation subsystem (the garbage collector, *i.e.*) and other facilities may link. This code is derived, furthermore, not from the "executable" portion of the source code of the programme, but from type declarations themselves.

Once a shortage of available memory has been detected (or garbage collection has otherwise been triggered—preëmptively while waiting for input, for instance, or while the stack is at low-tide [Wil88]), the garbage collector proceeds to determine which storage (or which objects) are actually possibly in use, judging by an inductive traversal of the entire state of the computation, based in the actual machine context as saved at the moment of invocation. This process relies on detailed information about the contents of every field of every data structure in the computation, including register frames, stack frames and, potentially, the object code itself. It is, of course, the efficient representation—through **Visitors**—of this information that is the primary topic of this thesis.

implementable, some variant of this technique should now be considered the baseline.

[6]For reasons of tuning it may be desirable to provide the programmer with methods of allocating and, particularly, freeing storage more explicitly; the programmer may know, for instance, that some value can never outlive some computation, a fact whose determination by static programme analysis may be difficult or impossible. This issue is touched upon in [Spa86], though current developments in static programme analysis are reducing the practical importance of this concern—or rather are making it increasingly likely that the *compiler* can do a good job of inserting such hints.

While it is not true that everything reachable through an inductive traversal of the computational state forms part of the future of the computation (consider, as a trivial example, the fate of the last two arguments to an *if*-expression: at least one of them is dead to the computational result, though, like Schrödinger's cat, it may not know it yet[7]), it *is* the case that, in a strongly typed language, everything *not* reachable through such a traversal is *not* part of the future—since there is no way for storage to be drawn into the computation except through allocation, itself under the management of the storage recycler and synchronised with garbage collection.[8]

The conceptual utility of garbage collection, therefore, relies fundamental on the same manner of insight as the conceptual adequacy of strong typing: the latter is justified by the observation that storage uninterpretable by the source programme is uninteresting to any (usual) goals of the programmer; the latter by that storage inaccessible to the object programme is uninteresting to the computational goals of the programme.

Recent research has demonstrated that it is in fact possible to provide for some kind of garbage collection facility even in systems that (like C++) are only weakly typed. These schemes operate by making the worst-case assumption that everything which, judging from its location, alignment and bit pattern, might possibly be a pointer, is a pointer in fact, and recursively tracking down the (putative) objects thus (putatively) referenced.

This class of scheme involves quite high overhead (because of the necessity of deciding whether something might plausibly be a pointer and the need to maintain all auxiliary data structures in external tables because of the uncertainty over whether objects reachable from a maybe-pointer are as they are believed to be). It suffers from the further disadvantages of being imprecise (in particular, aside from, say, integral values that are quite coïncidentally pointer-like in form, they will treat as live

[7]The cat itself is not an observer since it is either dead or imaginary, often both.
[8]Which is, incidentally, why imprecise garbage collection (see below) works at all.

objects transitively referenced from dead fields in inactive cases of variant structures, potentially a serious problem for naïvely-written linked-structure mutation code), failing to resolve the loss of memory due to fragmentation (insofar as objects whose types cannot be determined reliably cannot be moved safely—for only real pointers—and not pointeroids—should be fixed up when their apparent referent moves) and somewhat unsafe (since a weakly typed language actually permits the determined programmer to "disguise" pointers in such a way as to make them opaque to the garbage collector—as when the application performs pointer arithmetic or makes use of "spare" bits in pointers to hold administrative data[9]). Since its main purpose in being is to support weakly typed languages[10] which we hold to be undesirable for independent reasons of software engineering, it is not clear that it provides a good approach to the problem—though it assuredly constitutes a valuable contribution to the literature.

We should also mention at this point the profound sensitivity that garbage collected systems exhibit to the presence (and precise details) of virtual memory, since a number of our remarks will be justified with reference to these interactions. On the one hand, we find that garbage collected systems (particularly ones that employ tagged object representations, since they tend to encourage the proliferation of small objects related by pointers) tend to possess very poor spatio-temporal locality properties, both locally (related objects get allocated to remote pages[11]) and globally (for garbage collection itself is a global-scale periodic event, more or less the worst case for virtual memory systems). On the other, the presence of virtual memory makes the process memory space *elastic:* using excess memory tends to degrade performance asymptotically rather than terminating the process abruptly. Advantage can be taken

[9]A despicable practise to which the author hirself has stooped on occasion.

[10]And, in particular, it seems, C++, the mutant descendent of a mongrel language whose only positive attributes are a kind of pugnacious hybrid vigour and a long (if not unbroken) tradition of compilers so bad that they inspire hackerly programmers with the warm comfy feeling that they can trust the code generator never to surprise them. ☺

[11]Or cache lines, or whatever: this applies across the entire storage hierarchy.

of this elasticity to tune data structures whose sizes can be predicted statistically on the basis of recent behaviour, but not known with certainty. For example, in an algorithm that uses a stack with a reasonable expected size but a worst case proportional to the size of the entire heap (we shall see such an algorithm later) the heap size on each garbage collection cycle (and thus, indirectly, the garbage collection frequency) might be chosen so that it and the *expected* stack can fit in physical memory simultaneously. In the event that a larger stack is required, the next garbage collection will be *slow* but will not fail (under reasonable assumptions about the availability of backing store, at least).[12]

7.2 Two Basic Approaches to Garbage Collection

The two basic approaches to garbage collection which dominate the literature are *stop-and-copy* and *mark-and-sweep*. The more sophisticated techniques—the parallel methods, imprecise garbage collection, the various scavenging collectors, and so forth—are generally seen as modifications of one or the other of these two fundamental types, either improving its performance or weakening its preconditions through the exploitation of knowledge about which are the usual cases encountered in the process. There is at least one other technique of great importance: that of *static* analysis of data lifetime, whereby it is determined at compile time how long a datum may persist; this is of course at the heart of any competent compilation strategy, but (as it is essentially a method for the *avoidance* of garbage collection) is outside the scope of this thesis, except in its indirect implications for the handling of compiled code.

The Visitors herein described can be applied directly to the support of either

[12]Actually, given the reference patterns associated with stacks (reference frequency decreases sharply with distance from the top of the stack), algorithms are relatively insensitive to the amount of physical memory available to hold it, on the assumption of an informed memory management system. But the existence of uniformly accessible memory beyond that which is optimally available is crucial. We shall later consider other, more specialised data structures whose probable size is much smaller than their worst-case size, and which *do* require random accessibility.

of these primary techniques (though as we shall see, there are reasons to expect greater payoff in the case of systems that can be expected to select a mark-and-sweep approach).

7.2.1 Stop-and-copy reclamation

Probably the simplest of the garbage collection strategies is the basic copying collector. In essence, this technique operates by discarding the *entire* heap, and salvaging the active term (that is, those structures that are accessible from the present state of the computation) into a new heap, one that was presumably cleared on the previous cycle.[13] The entire garbage collection task is thus reduced to a single operation: that of making a faithful copy of the current term (or, in more traditional terms, of all objects that are accessible from any part of the present state of the programme). The complications are (1) that it is necessary to identify all the "roots" of the current term—all the places that the programme state makes reference to objects in the heap; (2) that the heap is in general composed of a fairly complex and heterogeneous collection of objects; and (3) that these objects are potentially connected in an arbitrary graph, somewhat complicating traversal.

Of these three difficulties, the first must be addressed head on, one way or the other: either by restricting the implementation of the language in such a way that the identification of marking roots is straightforward (in the most extreme case perhaps restricting all references into the heap to a single stack, itself possibly a heap object—this is the approach we adopted in the modular graph reduction interpreter Jeter); or, at the other extreme, by ensuring that there is sufficient ancillary information available to determine what parts of the normal machine state in fact represent such references at the moment of garbage collection (see, for instance, section 5.4).

The third question, that of correctly manipulating a potentially cyclic structure,

[13]In practise even fairly naïve implementations employ *three* areas: one to hold the surviving term in odd cycles, one to hold it in even cycles, and a single, larger, area, in which new allocation is performed when the current hemispace is full.

is easily addressed for a stop-and-copy collector, since the very nature of the copying operation is to change the address of object. Provided some method exists to record the new address of an object that has already moved (and it is possible in some circumstances that the space vacated by the moved object will be appropriate to this purpose), the very fact that an object now resides in the new area will suffice to indicate that it has already been processed, handling both forms of multiple parentage—cycles and "DAGginess"—at once.

The middle of our three complications is, of course, the main topic of this thesis, and is addressed in the current context throughout the following sections of this chapter.

Aside from being very simple, the stop-and-copy approach has the advantage that it is potentially very time-efficient, in that it need only ever examine memory that is still in use at the time of garbage collection, and that only once; it has no cost component dependent on the amount of space that is reclaimed (making the term "storage recycler" more appropriate than "garbage collector," a term suggestive of a procedure with the opposite characteristics). This is particularly attractive in the presence of large amounts of cheap virtual memory, since it now makes sense to employ a very large heap of which only active parts will generally remain resident at the top of the memory hierarchy. Techniques are known to reduce the input-output bandwidth and external storage cost associated with a paged heap, at least in those operating systems which provide for user paging processes [Wil91], further increasing the attractiveness of the method. There is still, of course, a concern that if the lower parts of the storage hierarchy are too slow, the repaging of the inactive parts of the present term will dominate the cost of garbage collection: for the time between garbage collections will necessarily exceed the basic time constant of the pager (by the hypothesis of large virtual memory). This can be addressed either through the application of generational techniques or, parallelly, by adjusting the estimated cost of outpaging old survivor (pages) appropriately in the pager's victim selection code, on the generational assumption that new *pages* die young.[14]

7.2.2 Mark-and-sweep reclamation

In contrast to the copying method, a mark-and-sweep garbage collector operates with the heap *in place,* rather than by moving the current term between disjoint hemispaces. This leaves the door open for an approach in which active objects never move, an important factor if relocation is for some reason prohibited, as in imprecise garbage collection (where pointer fields requiring update cannot be located accurately) or when physical pointers to heap objects have been exported from the garbage collector's domain. Considerations of fragmentation control, on the other hand (particularly significant in virtual memory systems with heavily loaded physical store), exploitation of generationalism, and reduction of the complexity (and concomitant increase in the performance) of allocation code can all argue for a compacting approach in which retained objects are moved into contiguity.[15]

In either the compacting or in-place variants of the technique, the first step in garbage collection is to perform a complete traversal of accessible storage, making note (in some fashion) of its extent. If compaction is not desired, the next step is to gather the remaining—unmarked—space together into a **freelist** for subsequent reällocation; after which any auxiliary data structures in the garbage collector are reset to their initial values and normal processing of the client programme is resumed.

In the compacting case, the marking phase is followed, separately or together,

[14]The crucial advantage of a dead *page* is that its residency is immaterial: it can be deallocated from the pagemap without examination (being replaced by a demand-created page to keep the address-space in use) and thus need *never* cause a slow page fault.

[15]At least if objects are not of uniform size. Even then, uniform object size need not be *universal:* implementations in which objects of different size and/or type are segregated into distinct subheaps (**zones**) can be quite effective (particularly on segmented-memory architectures). With care, in many type systems amenable to a simple tagged object layout, such an approach can exhibit the high storage utilisation and facility of array representation of untagged implementations, without losing the benefits of tagging. Effectively, tag information is moved up into the segment (or zone-determining) portion of the pointer.

by the relocation of objects to their new, contiguous, positions and the updating of what pointers they contain to reference these new locations.[16] Once again, client computation resumes after reinitialisation of any auxiliary structures.

The mark-and-sweep method thus involves at least one pass over the retained storage (two or three in the compacting case), *and* a full scan over recycled storage (or some summary image thereof) in order to distinguish it from that which is retained.[17] The technique may be justified, however, on other grounds: in particular, in not requiring hemispaces it has lower address space requirements, and it appears to have other advantages in handling more complex or highly constrained objects (though new technology may yet shift this balance).

7.2.3 The common ground

A number of fine-grain operations are manifestly common to both these approaches, mark-and-sweep and stop-and-copy. First, there is the inductive determination of which objects are accessible to the current state of the computation, whose basis is the location of the marking roots—those places where the heap values are mentioned in the implementation machine model—and whose induction step is the traversal of heap objects to determine their bounds and locate any transitive heap references they may contain.[18]

Secondly, each of these basic strategies (though not the compactionless variant of mark-and-sweep) relies on the ability to move arbitrary objects from old addresses to new.

[16]Obviously we are glossing over the details of how the new pointer targets are determined.

[17]The use of a hierarchic structure for occupancy marking could presumably reduce this to more like $O(n \log m)$, where n is the size of the *retained* term and m, the overall size of the heap, a much more satisfactory bound when the heap is large—at least if storage is elastic.

[18]Actually, it should be noted that it is potentially necessary to traverse arbitrary amounts of storage that are not themselves in the heap and susceptible to conventional recycling. Consider two cases: that of a generational garbage collector, in which portions of the heap may be tenured, but from which pointers to new objects must still be examined; and that of, say, a system file descriptor table, where although the table and its entries may themselves be (for technical reasons) static, heapened buffers may be associated with those that are presently open (as is the usual practise in implementing C).

Finally (and again excepting compactionless mark-and-sweep) they both need to update objects so that any pointers they contain continue to reference the original targets, despite their possibly having moved within the address space of the process.

Each of these common elements has a range of possible implementations; but **Visitors** provide a straightforward, unified and efficient mechanism for handling all three. As we saw in chapter 6, furthermore, they capture similar commonality in a number of other domains where similar needs arise.

7.3 Garbage Collection and Type Tagging

Perhaps because of the great success of LISP, or perhaps because of questions of comparative complexity, garbage collection is normally associated with a *tagged* heap: one in which all the objects share a uniform structure (they have, that is, only a single type at the implementation level), a discriminated union in which the discriminant field, the tag, is at the (fixed) least offset. Although this is far from the only context in which garbage collection is feasible, it will be instructive to spend a few paragraphs in consideration of the techniques appropriate to it.

7.3.1 Traversal

The single most striking implication of object tagging is that objects are discrete, uniquely typed, and identifiable even when removed from their contexts (if, at least, it is known where they start): a (machine) pointer identifies a distinct object.[19] Among other things, this permits traversal of the heap to be performed in a *stackless* manner: a linear scan of a compact heap of tagged objects can reliably locate all pointers; thus, for instance, with tagged objects, a stop-and-copy recycler needs no auxiliary storage beyond its *to*-space.

[19]While in the absence of tagging it will be observed that a structure and its first field typically share the same address, being distinguished only by their type. Note that the extra clarity is being paid for in space: each object *must* be allocated storage of its own (though it may be possible to recover some of this cost in a virtual memory system through the use of a compressed backing store).

Similarly, where (as in the relocating mark-and-sweep algorithm) an update pass must be performed to "fix up" pointers to their referents' new addresses, a linear scan of the heap will suffice.[20]

Even in the event that such a linear traversal is not performed, an inductive visit of every object is greatly facilitated by the presence of a type tag: for the induction code can be a single recursive function that dispatches on the value of the type tag. It need contain no logic dependent on the particular programme whose state is being manipulated. This is bought, though, at the cost of essentially interpretive execution: each type tag can be seen as an opcode of a virtual machine specialised for data structure traversal. As we shall see shortly, the normal methods for interpreter acceleration—ultimately resulting in the compilation of types and the elision of the tags—are available.

(It may be possible to reap these benefits of object tagging even in basically untagged systems. If objects are gathered together into breadth-first pages during their first garbage collection, as suggested in [Wil91], it may be possible (under certain assumptions of object boundary identifiability) to tag the *pages* on which objects reside with the types—in our case the **Visitors** and their argument packets[21]—of their root terms (from which the types of the subterms follow inductively). Then all objects that have survived at least one garbage collection will acquire tags—and at a storage cost that can be amortised, at least for large objects, which untagged systems encourage, over an entire page.)

7.3.2 Forwarding

Another stop-and-copy subtask that is greatly simplified in the presence of tagged data is the representation of *forwarding* information, the record of where an object

[20]There is an amusing alternate reconstruction of the stop-and-copy algorithm in which *fixup* drives the process, with the movement of objects to *to*-space being a side-effect of the determination of their new addresses.

[21]In order to ensure progress we can use pointers in the rare case when these are large.

is located after it has been moved to the *to*-space. The idea is to reserve a type tag for the "type" Forwarded, whose representation has only one field other than the tag, and whose interpretation is that the object that was being sought is actually at the address in that field (and no longer the address being examined). The use of this mechanism *does* require that every object have at least one full-width field in addition to its type tag (there are types, such as "atoms" and various distinguished constants which might otherwise coherently need no fields at all),[22] but it does *not* involve additional runtime overhead[23] since these pseudo-objects exist only during the process of garbage collection: by the time the client application is resumed, all pointers have been fixed up and the forwarding "objects"—being, as they are, at the addresses of "dead" values—are no longer present in the graph.

This will not work in the mark-and-sweep case, of course, since in this scheme space is reused immediately, without regard to whether its previous occupant survived recycling. Thus auxiliary data structures must exist to hold this information, whether or not tags are present.[24]

(Once again it should be noted that all hope of superimposing forwarding information on untagged structures is not lost. Since (1) the fact that an object has been forwarded can be recorded in only one bit, and (2) where objects overlap they must be moved as a single unit and so can share a forwarding record, the tag need for these purposes be a single bit that is shared between all objects at a given address. If the underlying hardware provides "exception bits" on values, or if the data structure rep-

[22]There are strong arguments from scoping (as well as from the conceptual semantics of the atom as *atomic*) that even atoms with "properties" should have those properties stored in tables according to the *property* rather than being bound directly into the data structure of the atom. There is an attractive implementation of shallow-bound values under this arrangement and the building of coresident compilers under any binding discipline should be greatly simplified.

[23]Except in the case of a parallel garbage collector.

[24]As described in [Spa86], it *is* possible to store much—but not all—of this information in space originally occupied by objects that have been discarded, at the cost of performing fixup *before* relocation, a slightly more complex undertaking. The crucial observation is that objects (at least those that do not change in length during garbage collection!) *either* move to their new addresses together, as a continuous block with a single fixup value, *or* are separated by at least one cell of free storage, which can be used to hold fixup information.

resentation mechanism is sufficiently sophisticated, this might prove to be a practical approach.)

7.3.3 Handling cycles

As has already been noted, the stop-and-copy approach to storage recycling does not suffer from difficulties with cyclic data structures that are any different from those caused by partially ordered structure sharing: once an object has been moved its new address is used, and no attempt is made to move it a second time. In the mark-and-sweep arrangement, however, destination computation, motion and marking take place at different times, and so some interlock mechanism must be provided to permit the marking phase to terminate even on cyclic structures. When data are tagged, this is conveniently accomplished by stealing one of the bits from the tag to indicate whether the object has yet been reached. When traversal encounters an object whose bit is set, it can ignore it: it has either been marked already (in the case of structure sharing) or is in the process of being marked (for a cycle).

There is a particular subtlety here that is worthy of note: the correctness of this method of storing state in the object tag relies on the tag being associated with a unique object: with its address encoding a [type, instance] pair. If the tag (as with the reduced, single-bit tags we suggested at the end of the previous section) is associated only with the *address*, traversal could be incorrectly halted on encountering a large structure whose first *field* had already been marked through an independent reference of its own.

7.3.4 The tag

Let us turn our attention to the representation of the tag itself.

Perhaps the most obvious choice (at least for languages with only a handful of types, as is common where the tagged representation is chosen) and the most compact, is to use a small enumerated representation to encode the type of an object. Functions, like those in the garbage collector, needing to interpret the type field[25] will be coded internally with a case-statement dispatching on the value of that tag. Typically, the tag need be only one byte long.[26]

Considering, however, the penalty that many processor implementations place on the manipulation of unaligned data, the fact that such a tag requires 8 bits only and not 32 may be immaterial. One approach to "using up" the remaining bits has been to collapse array descriptors into their headers, putting some sort of repetition count in there. Another possibility is to allow the *header* to contain a forwarding pointer, using pointer values that do not point into the heap to represent types. But given that, as we have observed, the code *using* the tag field is likely to be in the form of a case-statement, which in turn—at least if, as in the case of the garbage collector, it is not sparse—is probably translated into a jump-table; and given, finally, that the number of places where a dispatch across all types is likely to be performed is small in comparison to the total size of the programme, it seems sensible to use an indirectly-threaded implementation in which the tag is itself a pointer to a structure containing entries for both administrative data (such as the printable name of the type, perhaps) and the branch targets for the various pieces of type-dispatching code—with each such application granted its own offset in the type descriptor structure.[27] This indirectly-threaded system is precisely the arrangement that the author's lazy functional database engine kernel, Jeter, employs for the tagged objects that comprise its combinator graph.

Three further refinements of this scheme suggest themselves. One is that, now the type has become a structure reached through a pointer, it might be made into

[25]Intuitively this would include *only* the garbage collector and type-checking mechanisms, but recall that in languages like LISP one can *enquire* as to the "type" of an object, and this notion of "type"—the top-level discriminant of a grand union—is efficiently represented by the same tag we are discussing here.

[26]Providing for the encoding of 127 types, since in the course of the preceding few sections we have "stolen" one bit plus one code from among the possibilities, marking and forwarding to effect.

[27]The astute reader will observe that this is precisely the structure that implementations of languages like Smalltalk-80 need to virtualise.

a heap object in its own right (a convenient method, among other things, of implementing type-introduction mechanisms). A second is that if one of the "aspects" of the type is more frequently used than the others (a **Visitor** is one likely candidate, while in graph-reduction systems the interpreter for a representation object is a good choice), we can convert from indirect to *direct* threading by moving the code for this most favoured routine up into the type descriptor itself. The remaining fields of the descriptor can be stored preceding the entry point (if the processor architecture supports offset branches well); they can be moved to negative offsets within the structure (unless type objects are kept in the heap and advantage is being taken of linear-scannability of objects); or, in desperation, they might even be located through an associative table elsewhere, if the performance gain justifies. Finally, in the most extreme extension of this development, we might move (a copy of) the most favoured interpretation routine *into* the object itself, producing a *subroutine*-threaded structure (a long-favourite technique of the author; and see [KL89]).[28]

These techniques for associating additional administrative information with efficiently represented code will be of further use to us in another context, below.

7.3.5 Immediate data

One optimisation that is quite common in implementations of LISP (and which is supported directly in the hardware on LISP Machines and the SPARC architecture) is to represent the members of certain data domains *immediately* within the word that would otherwise be a pointer into the heap. Typically this is done by the

[28]Of course, this is only advantageous when the most frequently encountered nodes have *very* simple semantic routines associated with them—routines of one or two instructions, at most; for longer sequences must be translated as calls or jumps to centralised routines to hold space consumption down, and double-jumping is likely to be inefficient. In fact, we should note two further drawbacks of the scheme: first, as noted in [KL89] (though in light of the performance benefits of deep pipelining, branch target buffers [MH86] and a separate code cache we would not be as ready as the authors to criticise hardware architects for this difficulty), many modern architectures do not take kindly to self-modifying code (and placing instructions into mobile and mutable data cells is self-modification with a vengeance from a hardware perspective); second, at least in those machines which lack such refinements as pipelining, indirect branching—at least through a *register*—may be faster than long direct branching, by simple virtue of denser instruction format.

expedient of reserving a few bits in the word to discriminate; since for efficiency (or even architectural) reasons many implementations will align all heap objects on boundaries whose pointers are divisible by four,[29] this may well not interfere with the address range at all. There may even be hardware support for the detection of misaligned data access, removing the need for an explicit validity check before dereference.

Another form of immediate data (that may not, however, be so straightforwardly recognisable) results not from such bit-stealing, but from code-stealing. Implementations may reserve certain pointer *values* to special implementation-defined meanings: sometimes (as with the null pointer) they may not in fact be pointers at all; in other cases such pointers may refer otherwise normally to objects that are not in the heap, for instance.

Such phenomena do not conceptually require any special handling; but they do introduce a second level of discrimination that must be performed as a garbage collector interprets objects: the type of a value must be checked both before and (when appropriate) after the word that names it is dereferenced.

7.4 Garbage Collection in the Untagged Heap

We turn our attention now to the issues of garbage collection in the context of a heap that is *un*tagged, but still strongly typed. Here we find that although the type of an object is still uniquely determinable, it is no longer a local property of the object's representation; information accumulated along the access path to the object may also come to bear. Nor, in the absence of tags, do objects have any unique existence: it

[29]There is, of course, nothing remotely "magic" about this number: it is simply a convention of contemporary hardware architecture that machines are grudgingly byte addressable, use 32-bit pointers, and honestly prefer their data aligned on 32-bit boundaries. This is justified by the "fact" that characters are 8 bits wide, though given that there are (by count of the Unicode consortium) something between 2^{15} and 2^{16} characters (not, of course, counting differences of language or font, for example) presently in use world-wide, this strikes the author as a somewhat peculiar piece of logic.

cannot be decided (other than by global analysis) whether the two fields of a complex rational are parts of a greater whole, for those fields may be accessible both through the complex number containing them and individually, by separate paths.

In the most extreme case, a system supporting subtyping or multiple inheritance might theoretically exhibit the behaviour that the same *copy* of the same pointer must at different times (when viewed, that is, through different access paths) be interpreted as having different types and referring to different—even disjoint[30]—objects. In fact, this case is pathological for us (as will be seen in section 7.4.2 below), but can always be avoided if the language requires that pointers and mutables have unambiguous type.[31]

7.4.1 Traversal

In the untagged case, traversal of the contents of the heap is really no more complicated than in the tagged case; in fact it will be seen that our eventual solution is somewhat simpler (if subtler). It is true, however, that the lack of tagging makes the problem more tightly constrained.

As we have just observed, in the absence of tags, the type of an object in the heap becomes a property only known by reaching it through a complete access path (starting, as it happens, from any object whose type is not dependent, though we shall not be using this fact). The implication is that while an hierarchic approach to term traversal is completely feasible, it is not in general possible to use a linear scan to perform breadth-first processing, as we could using tags.[32]

The actual mechanics of traversal are not significantly different, however: starting from a root object (whose type is known), we examine each of its fields in turn; and for those that are pointers, we extract their types from the description of the

[30]If the representation mechanism is smart enough to derive them, there are common cases in which the best representation of some multiply-heritable type has a "hole" at offset zero.

[31]The author finds the concept of side-effects occurring at multiple overloads mind-norking, and sincerely hopes that sie never has cause to implement a language with this feature.

[32]Though see the parenthesised passage at the end of section 7.3.1.

present object, and recur. Once more we note that the most obvious representation for an object type in this context is an element of an enumerated type (possibly taken in conjunction with a tuple of parameters)—in fact, the type $\mathcal{T}$ of type names, from chapter 3. Each time we examine an object we pass the enumeration value corresponding to its type, and this can be used to recognise special cases (of which pointers are an example) and to index into a table of representations for the various types in the language, encoded in some type representation language having a few control operators to handle variant records and so forth and using $\mathcal{T}$ to refer to the types of constituents. The traversal process is identical to the tagged case except that pointers are always paired with their type tags where their *referents* were before.[33]

7.4.2 Forwarding, sharing and cycles

The representation of forwarding information that seemed appropriate for tagged objects fails abysmally in their absence: we can still use the space previously occupied by an object[34] (or whatever other arrangement we might have made for the mark-and-sweep strategy) to hold its new address, but we cannot modify the tag of an object to indicate that it has been moved, if tag it hasn't got. If there is no constant part of a structure that we can use to record that it has moved, and there probably is not, we must resort to an external table having (naïvely, at least) a bit for each cell of storage in the heap.

When it comes to the handling of shared and cyclic structures we have an even greater problem. As has already been noted, types are not associated uniquely with pointer values. This means that the fact that an object's *address* has been visited

[33]It is amusing to note that the scheme of *zoned* storage as mentioned in footnote 15 of section 7.2.2, encoding type in segment values, is precisely intermediate between these extremes.

[34]It might be thought that even this poses problems in the case of types possessed of a hole at offset zero: writing a forwarding pointer into this field (even after moving the object "at" this address) is catastrophic if subsequent processing reveals another interpretation for the address, under which that cell is occupied. In fact, however, since structural overlap is significant, destination addresses cannot be assigned until *all* retained storage has been located, for systems that permit overlapping objects—by which point there is no difficulty, so long as every independently accessible cell of a moved object is subjected to the forwarding mechanism.

does not mean that the object *itself* has been visited, and so associating a *processed* bit with each target heap cell will not suffice. Fortunately it *is* adequate (in the absence of overloaded pointers) to associate a *traversed* bit with each *pointer*. In the event that there are multiple references to the same object (a frequent event, but one that is statistically dominated in practise by the singly-referenced case) this will result in additional work being performed; for the target object will be examined once per pointer and not once for all. The solution is correct, however, for the pointers *within* a structure will not be traversed if *they* are marked as already processed.[35]

7.4.3 Representation

The issue of finding a good representation for the type information is here rather different from when we had to tag each object. In particular, we are no longer paying a per-object space cost no matter how we decide[36] and so need not be especially concerned with density.

In the preceding we suggested that the basic approach to encoding the structural information associated with a type was through a data structure stored in an entry in a table, that structure to describe the layout of the objects of the type in some object-representation description language. The astute reader will already have divined that our preferred representation description language is in fact machine code: the entries in this table[37] are in fact the (cc npiled) **Visitors** for the various types, and the table

[35]Particular care must be taken in the presence of overlapping objects to ensure that no commitment is made to process a marked pointer and that the pointer is marked as soon as a commitment is made.

[36]Hopefully. In fact, during a worst-case recursive traversal of the heap, where all objects are linked linearly through fields other than the last (so that they are not susceptible to the tail-call optimisation), it is possible for (representations of) type descriptors of *all* heap objects to be on the stack simultaneously, a (large) $O(n)$ expenditure. An algorithm (related to me by H. J. Boom, but whose origin we have been unable to discover) is known which consumes space of $O(\sqrt{n})$, but its application in the present context is tricky, and we shall not present it here (the interested reader is referred to [Spa86] where this algorithm is given and much of the mechanics of its interaction with **Visitors** is developed). Clearly it is desirable to choose an initial stack size more on the order of $O(\log n)$ frames; and this is one of those cases where in a virtual memory system we may choose to rely on storage elasticity to preserve correctness under an over-aggressive allocation heuristic.

[37]Though for debugging purposes it may be shadowed by a second table containing the abstract syntax—or even the source form—of the original declaration; in fact the same shadowing can be

is itself an object of the mysterious controller type $\mathcal{M}$ from chapter 3.

In and of itself, this will not suffice: for calling the **Visitor** of an object will result in the object being traversed, but no action being taken.[38] Each generic operation is therefore given a private copy of the table with the entries for those types that it manipulates or interprets directly modified to its ends. The new entries must respect the type signature of a **Visitor**, but can perform arbitrary computation: in the support of garbage collection we would expect $\mathcal{M}$-objects computing, perhaps, the size of an object; the marking of the memory extent which it covers (which may not be the same thing); its recursive copy to a new location (including or excluding objects reached through pointers, depending on the style of the garbage collector); the update of its pointer fields, and so on.

On the subject of space consumption, let us note that these type description tables (which are potentially rather large[39]) can be expected not to proliferate as rapidly as might be initially assumed. On the one hand, it should be observed that they can be treated as values and allocated and discarded as needed (if that should prove useful). On the other, experience has shown that the application of a few optimisations (as described in section 5.6.5) can both shrink the physical representations of the tables and collapse them together; actual garbage collectors implemented with this technique have shared a *single* such table in memory between *all* the internal functions of the garbage collector.[40]

expected to occur for *any* runtime structure that is a little opaque to dynamic examination—though see section 6.4 for a discussion of the application of **Visitors** themselves to this task.

[38]Other than its size being computed in an arbitrarily—possibly infinitely—inefficient manner, at any rate.

[39]Though it should be remembered that they contain entries for type *constructors* and not, strictly speaking, for individual types.

[40]There is, as always, a tradeoff here: the resulting optimised copy of the table has too much garbage-collection-specific information folded into it to be itself useful for many other tasks; the particular specialisation was chosen because, considering garbage collection in isolation, it provides excellent performance in both time and space.

7.5 Garbage Collection with Visitors

As we have just seen, something very much like the **Visitor** drops quite naturally out of the representational optimisation of a tagless garbage collector. Approaching the matter from the other direction, we now examine garbage collection as an application of visitation.

Not all of the garbage collection task is directly involved with the traversal of data, but in each method most of the work lies in locating and updating objects and the pointers within them. The interface between a garbage collector and the visitation system thus resides in the client routines that implement a few crucial operations. The simplest of these[41] is the determination of the *size* of an arbitrary object. Here we assume (for the sake of simplicity) that objects start at offset zero and contain no "holes," and that pointer arithmetic operates in the units of Size[42].

```
Size :< size @ (TYPE T, %T t) >:
  𝓜 s : baseMode
  ; 𝓟 :< s[pointer] @ (𝓟 p0, 𝒜[pointer] a, 𝓜 m) >:  p0 + pointerSize
  ; s[constructor[T]][t, arguments[T], s] - t
;
```

The operation of this function is straightforward: a visitation mode s is constructed, specialised to size determination. This new mode differs from the system-provided baseMode only in the treatment of (subobjects represented through) pointers, since in this case only the size of the pointer itself is to be included. Thus (s ⌜𝓟⌝) is updated to step silently over its argument in the manner of the visitors of primitive binary types. Visitation is then initiated fort (if T is a pointer type, *i.e.* t is a pointer to a pointer, then the entry point s (constructor[T]) will be the revised

[41]Though in a thoroughly optimised garbage collector we can generally expect *not* to see it as an independent routine.

[42]This diverges, for instance, from the C convention that pointer arithmetic is scaled by the size of the referent (practical because such operations are restricted to types of manifest size).

pointer visitor s $\ulcorner\mathcal{P}\urcorner$ itself; otherwise it will be, by construction of baseMode, exactly $\beta_{\text{constructor}[T]}$), and the total linear distance traversed, the difference between t and the pointer returned from the visitation, is the actual size of t's referent.

The storage-related garbage collection routines that do involve pointer traversal are not significantly more complex. Consider, for instance, the operation of marking all accessible storage in a mark/sweep collector, given functions mark[start, size] (returning start + size) and isMarked[pointer]:[43]

```
P :< traverseMarking @ (TYPE T, %T t) >:
  M s : baseMode
  ; P :< s[binary] @ (P p0, A[binary] a, M m) >:  mark[p0, a]
  ; P :< s[pointer] @ (P p0, A[pointer] a, M m) >:
      if    isMarked[p0]
      then  p0 + pointerSize
      else  P p : mark[p0, pointerSize]; traverseMarking[a, p0], p
      end
  ; s[constructor[T]][t, arguments[T], s]
;
```

Structurally identical code is used to implement pointer update (in the event that the garbage collection technique relocates objects within the heap) and to provide the copy operation of the stop-and-copy method (though as already noted this last may involve additional restrictions on the storage formats that the implementation can employ).

The code we have just presented is designed for garbage collection in the *un*typed heap, but as have shown, tagged data in fact a subset of of tagless data, corresponding to types of the form (TYPE T, T t) (possibly under some encoding of the type representation). Thus the code fragments above will apply directly to tagged data

[43]This function is coded in a side-effectual style; technically this is unnecessary, since the marking table could easily be reified and passed around the function nest; but here as elsewhere in this thesis we omit the additional uninterpreted polymorphic semantic argument to Visitors that would transmit this information. The present formulation is, in any event, probably more true to a practical implementation.

as well—in practise the (single) **Visitor** for tagged data performs dispatch to the sub**Visitors** for the various data fields, and in a matter of a couple of machine instructions. This dispatch operation is the same as would be performed by the garbage collection routines themselves in a more naïvely factored programme.

7.6 Conclusion

In this chapter, we have examined the application of **Visitors** to the problems of garbage collection. We have seen that those parts of the standard task that scan, analyse or update data in the heap can be reduced to triviality (specifically, to on the order of half a dozen lines of executable code each, with no type-specific code whatsoever provided by the programmer). There is no cause for surprise in this fact, since we have shown in the course of our development that **Visitors** themselves can be derived by transformation of the data structures natural to the task, abstracting the description of data, its processing and its physical format in memory away from each other.

Although this development recapitulates the history of the notion in our work, and it is thus with storage recycling that we have the most experience, it must be emphasised that the abstraction of the garbage collection application out of the **Visitor** proper is complete: it is in no way specialised to this particular task, and a development similar to the above could have been provided for any of a number of other tasks involving the traversal and manipulation of arbitrary data, as in fact we saw in chapter 6.

In the next chapter we describe the empirical experience on which this, and all our work on **Visitors**, is founded.

Chapter 8

Empirical Implementations

Although the full development of **Visitors** as described in this thesis awaits their incorporation into a .ative-code compiler developed from scratch with their application in mind, they have, in one form or another, seen implementation in a number of different systems. Taken together these pilot implementations validate the large part of the present scheme empirically, the main exception being the mechanisms relating to the processing of active code in unconstrained stack frames, as described in section 5.4. On this latter point, however, see [Gol91] and [App89], which effectively describe the application of **Visitors** as specialised to garbage collection, developing strategies identical in essence to those of [Spa86].

8.1 Lithp

The basic ideas behind the **Visitor**—that garbage collection support code should be derived automatically from data structure descriptions, that enough information was in principle present in object code and stack frames to enable a garbage collector to do its job without seriously constraining code generation, and that such routines, once generated, could in fact be turned to a number of other tasks (binary transput being a particular concern)—arose during the development of Lithp [BS85], H. J. Boom's Scheme-like LISP research system.

At the time the author joined the undertaking, Boom's code already featured a garbage collector in which—in the interests of modularity, rather than out of any clearly defined theoretical motivation—almost all the knowledge of data structure layout had been factored out of the garbage collector proper into two sets of traversal routines, one to locate pointers within an object and one to traverse all the fields by underlying (host architecture) type.

In the quest for better performance, the author replaced the original garbage collector with one having the following characteristics:–

- a tagged object representation (unremarkably, in view of the LISP client system's semantics);
- a compacting mark/sweep strategy;
- support for a number of data structures requiring special attention during garbage collection: file handles that must be closed on deallocation, hash tables keyed on address (with, unlike COMMON LISP hash tables, inverse retention dependency for the keys: see section 4.6.4), and objects that change size (as a dynamic tuning measure) during garbage collection;[1]
- careful consideration of the effect of memory reference patterns on virtual memory performance (with attention paid to the distinction between read and write access); and
- exploitation of generationalism (though this was not a generational collector *per se*) through a scheme of shallow and deep garbage collections, and using the fact that in compacting collectors memory address encodes object age monotonically.

[1]The interesting implication here is that during "compaction", not all objects move *down* in memory; in fact there is a possibility of futile garbage collection in which the size of the current term actually *grows*. This was handled in the manner that has now become standard, through the use of a succession of panic levels—since object growth is purely heuristic, it is possible to squeeze them through the application of "external pressure."

While working on the new garbage collector, we experimented with the automatic generation of lisp object formats, their constructors, eliminators and garbage collection routines, from field lists supplied in macro calls. Eventually the implementation limits of the VAX Macro-11 assembler forced us to abandon this approach (spurred by the success of the bootstrap compiler written entirely in Stage 2 we were, after all, asking rather a lot of its integral macro processor) and although we retained the **Visitor**-like software architecture we returned to hand-coded **markers**.

The practical precursor of the generally applicable **Visitor**—as opposed to the garbage-collection-specialised **marker**—arose when considerations of resource availability persuaded us to introduce into the runtime system a facility for storing compressed heap images. Rather than writing new code to perform the traversal of the objects to be saved and restored, we subverted (as we then saw it) the **markers**, recoding them where necessary so that they could perform both functions smoothly.[2]

The limitations of this early work were, then, that:-

- with the arguable exception of the stack, which was traversed with a similar mechanism even though it was not itself located within the heap, and although the heap traversal code did not depend on it (behaving instead as though every object were an independently licensed variant record), the objects processed were in fact tagged;

- automatic generation of **markers** from static data structure descriptions was eventually abandoned for technical reasons;

- the work of traversal was spread between several sets of routines, rather than being strongly identified with a single executable image of each type[3];

[2]The required changes were of the form of (a) increasing the precision with which field types were identified, particularly with respect to relocatability and a couple of factors relating specifically to compression; and (b) changing the order in which information was processed so the fact that during input objects must be manipulated that are only partially initialised did not pose a problem.

[3]Though note that even now we advocate the use of optimisations which *derive* such specialised structures from more general **Visitors**.

- though the various traversal routines were coded in such a way as to preserve some machine registers transparently and to use others for intercommunication in a more-or-less consistent fashion, there was no clear understanding of this technique as an optimisation over closures and the compilation of general recursive nests; and

- as a consequence of the taggèdness of the representation (which as we have seen restricts objects to single interpretations), all type analysis encoded by the **markers** went directly from the source "type" to the implementation structure; there were no intermediate (opaque) type levels, and no opportunity for them.

Its successes can be enumerated as follows:–

- it *worked,* supporting Lithp's own compiler, Fox,[4] and that of the total correctness language, Brouwer [Dev84], as well as various experiments in data structures and functional programming over a period of several years.[5]

- the performance of the code structured by these techniques was demonstrated to be more than adequate;[6]

- the technique of re-use of functionally (if not, at that time, closure-) parameterised general interpreters to perform tasks (in fact, unforeseen tasks) other than garbage collection was demonstrated most satisfactorily;

- it provided valuable experience in the design of language support facilities, and in particular in the techniques of *very low-level* functional programming.

[4]Fox—Boom's work, and which the author maintained and extended for some years—was itself an interesting beast, being a full-scale tree-grammar-based native code compiler written entirely (well, except for some minor instrumentation code) in the pure functional style.

[5]All of these applications were also direct early influences on the shape of present work.

[6]Overall, the new garbage collector was about an order of magnitude faster (by the wall clock) than the more naïve code it replaced. This performance gain must largely be credited to increased attention paid to the performance characteristics of virtual memory operating at the edge of its capacity and the implementation of a number of optimisations inspired by early generational systems [Ung84].

A clear *indirect* benefit of the **marker** was that the extreme modularity it permitted in the representation of data objects made it possible to expend relatively more effort on the macroscopic behaviour of the garbage collector. Similar benefits can be expected to derive from the automatically generated **Visitors** of the present model, since they permit much greater decoupling of the code generation and data structure selection mechanisms of the compiler, on the one hand, from the runtime system on the other.

8.2 Zarquon[7]

Zarquon, undertaken by the author as a term project in [[19??-??]], was conceived as a *fully generic* editor for tree structured data: while editing operations were to be presented to the user as manipulating a uniform display image, the abstract parse trees backing the image were in fact created on demand (on a cache basis) from the objects being edited, and unparsed back into them as edits were performed. The parsing and unparsing operations were delegated to **Visitors** of a form quite similar to the present design (though operating over a rather different type system, one revolving around types that are meaningful to the *user interface* rather than the host processor, and potentially performing nontrivial transduction). In this manner, it was hoped to build a tool that could edit objects on provision of a loadable driver module containing the requisite visitors and transput code (and possibly, though not normally, special display and interaction logic), without prior arrangement with the editor.

Zarquon (a project of frankly excessive scope) was never completed; in particular, the problems, both practical and theoretical, of generating and managing a suitable "generic editing" display were never solved.[8] Nonetheless, we succeeded in

[7]Etymological note: the Great Prophet Zarquon is, of course, notable for (among other things) cutting his Second Coming rather fine and arriving during Max Quordlepleen's magnificent emceeal of the End of the Universe, at the restaurant of the same name [Ada85].

[8]And it was learned that C is not the systems programming panacea that some would claim.

demonstrating that **Visitors** could fairly comfortably be extended to the domain of editing (with its irregular, local updates of objects), and would operate smoothly with *untagged* data not designed for amenability to this (or any other editing) technique.

Once again, let us summarise the positive and negative aspects of this project. On the down side:–

- Zarquon was never completed; the system as a whole never ran; and
- no attempt was ever made to generate Zarquon's **Visitor**-analogues (which were written in C) automatically: since the objects Zarquon manipulated were of external origin, writing **Visitors** by hand was the main step in adapting the programme to a new type (the idea being that this was at least as easy as and far more flexible than the alternatives).

While on the upside:–

- the project demonstrated the feasibility of this technique for the manipulation of *untagged* data, something that had not, to this point, been done in practise;
- the objects successfully manipulated included alien data structures (actually C structures used by the AmigaDOS operating system) which were *not* designed with amenability to the technique in mind;[9]
- it provided us with a feeling for what could—and what could not—reasonably be accomplished with this method in a high-level language;[10] and
- the project saw the (successful, as far as this aspect went) application of **Visitors** to a nontrivial external task, and one that involves more than structure-preserving global operations analogous to garbage collection (as the previous

[9]Though surely they did not exercise the full possibilities of the C language—which could have taken them into the domain of the untyped.

[10]In particular, it was at this stage in the development of the **Visitor** that we discovered that **Visitors** are not even typable in ANSI C!

non-garbage-collection application, that of moving a term to and from an external file, was analogous).

On a relative scale of abandoned projects, then, Zarquon was a qualified success: although its primary objective remained unsatisfied, it validated the technology presented in this thesis in a broader domain and provided a great deal of beneficial practical experience, in particular providing insight into the places where system level support—such as automatically derived **Visitors**—would be most useful.

8.3 Jeter[11]

The Center for Information and Language Studies at the University of Chicago is presently developing a large-scale structurally-oriented textual database system with a *normal-order* pure functional substrate. This system is designed for use by the academic public; present plans call for it to be shipped with a CD-ROM (being developed with the ARTFL project) containing the *Trésor de la Langue Française* database (to 1939) in early 1992.[12] As far as we have been able to determine, this is the first "industrial strength" application of normal order reduction, a technology which has hitherto been confined to the laboratory.

We dubbed the runtime system of this functional database server **Jeter**; it is in essence a graph reduction interpreter kernel, written in ANSI C, and designed to be efficient, portable and highly extensible; in particular we hope that in the future it will play host to a number of different compilation strategies, employing different schemes of combinators and/or supercombinators [Dil88, AJ89a]. It has, in fact, shown clear signs of satisfying all three of these objectives: it is many times faster than the Unix TIGRE interpreter (on which we did our early prototyping), and in some

[11]Jeter is named in part after the writer K. W. Jeter who, it must be said, out-Dicks Philip K. Dick; and in part because there was some initial concern that its performance characteristics might be such that we would have to throw it away in a hurry—a worry which, happily, has subsequently proved unfounded.

[12]See [BDM+86, ZDW90, DZW90, ZWS92]).

common applications—with naïvely coded queries and without extensive tuning—is significantly faster than the hand-crafted C query processing routines it replaces; during its development it was moved backwards and forwards between Unix SPARCs and IBM PCs without glitches[13]; and at one point a new illative datatype was installed in the interpreter by two people new to the system in the space of about two hours. All Jeter heap objects are described by **Visitors**; the only other description an object type has in the system is implicit in the code that implements its semantic primitives.

Most objects in the Jeter heap are tagged (since the graph reduction paradigm requires that subgraphs at least be marked as evaluated or unevaluated, there is no loss in this; in fact, it has been turned to advantage, and almost all code in Jeter is dispatched automatically on type by the interpreter). The exceptions (though the exact list changes frequently) are mostly system structures such as evaluation contexts, spine stack segments and so forth; but may shortly include various kinds of transput and string buffer. Stack segments in particular exercise the generality of **Visitors**, since although it is extremely desirable that storage not be retained on the basis of inactive stack slots, performance considerations led us both to store stack pointers remotely from the stack segments to which they relate, and to introduce some hysteresis in storage allotment between adjacent segments, yielding a quantity of state that must be propagated along the stack. It is, of course, unremarkable that this data structure exists and is garbage collectible; it is more noteworthy that it required no code and no support to operate beyond that which was already part of the system.

Jeter currently has two distinct methods of storage reclamation: it uses a one-bit reference counting scheme with counts in the pointers[14,15] [WF77]; and since this

[13]Aside from questions of Unix buffer flushing policy: under Unix, unbuffer output is prohibitively inefficient, while buffered output is jerky and makes demonstrations of lazy evaluation appear somewhat unconvincing!

[14]The fact that in order to do this we need to subvert a pointer bit—something *not* sanctioned by the ANSI C standard!—is our one major point of non-compliance and non-portability.

[15]The code that implements the one-bit reference counting scheme is intricately interwoven in the

(although it reclaims well in excess of 95% of storage allocated and will run for quite a long time in a multi-megabyte heap) is even more approximate than full reference counting schemes, it is backed by a conventional stop-and-copy collector. Both of these reclamation schemes employ **Visitors** for object traversal (though the copying collector currently has a breadth-first traversal strategy taking advantage of tags—and their ability to identify objects "out of context"—where they exist: objects that are known to be tagged can be visited with the "tagged object" **Visitor** even during a linear scan of the (new) heap).

During early development, both the output system and the interpreter itself (the two, in fact, being more or less indistinguishable) made extensive use of the **Visitors**.

In sum, the shortcomings of Jeter as an instance of **Visitor** technology are as follows:-

- since Jeter is at present entirely coded in (almost entirely portable) C, it has not been possible to implement techniques for manipulating either live code or the system stack directly;

- more or less at the requirement of the language implementation technique itself[16] all user-visible data are tagged in the heap, and there is at present no extensibility to the type system at the implementation level—the types and their **Visitors** are linked in; and

- once again we have not had the opportunity to generate **Visitors** automatically (in this case because of the inflexibility of the C macro preprocessor).

However,

present interpreter (though it is certainly possible to write less storage-efficient code that largely ignores its presence, since reference counting is only approximate anyway), in an effort to maximise the use that is got from that one bit per object. In practise this is turning out to be *the* major source of implementation and maintenance effort, and is a strong argument for either the timely automation of code generation or the abandonment of reference counting.

[16]Though there is hope for extensive computation with unboxed data in local eager mode [PJL91], and we have ourselves considered placing unboxed data into the stack for locally strict computations.

- we have been able to demonstrate the smooth coëxistence of tagged and tagless objects in the heap, mediated by **Visitors**;

- **Visitors** have again proven their applicability to many tasks that were not specifiable in any detail before the fact, a boon to both prototyping and final implementation;

- the same code supports not only all phases of storage reclamation by a single technique, but in fact supports two quite different reclamation methods on the same objects, in the same heap: both reference counting and stop-and-copy; and

- Jeter is well on its way to being what can only be considered a production system, as it is presently managing hundreds of megabytes of data and will shortly be in the hands of dozens of users of widely varying technical competence.

The next logical step in the evolution of Jeter—and the next step in the deployment of **Visitors**—will in all probability be the implementation of a true compiler, making, among other things, nontrivial decisions about the representations of data.

Chapter 9

Conclusion

Visitors are executable images of data types. Because of the simplicity and uniformity of types in conventional languages, even a simple compiler can generate high-quality code for them; complications arise at a reasonable rate as compiler sophistication and semantic demands increase. The methodology of **Visitor** implementation and use is, as a whole, principled, flexible, and efficient.

Since **Visitors** ultimately derive from the source programme's type structure it is imperative that the language they support be strongly typed; this is *trivially* true of the "untyped" LISP-like languages (for which **Visitors** provide mainly a convenient rapid-prototyping tool for various system internals), but is an important global property in which "typed" languages in general may fail; in particular, languages such as C and Pascal (as normally construed) *lack* this property and cannot be provided with (secure and automatically constructed) **Visitors**.

Visitation was originally invented as a tool for increasing the modularity of a garbage collector. In this application it can also vastly extend a garbage collector's flexibility, operating (if necessary) in a tagless environment without exhibiting "interpretive" behaviour.

Whether as an independent justification or an added benefit, the *same* automatically generated code can be used in support of other language-, operating-system- and user- level tasks that involve global operations over data structures under variable notions of type opacity and atomicity. Contrariwise, since **Visitors** are explicitly representable at the intermediate code level (if not normally in the source) there is scope for a good compiler to specialise them to a particular task without the need of special facilities.

If applied aggressively in the context of an operating system, **Visitors** have the potential to make type information both more present and more immediately useful to the *whole* system, making truly transparent remote procedure call and tolerably efficient inter-architectural virtually shared memory a practical and attainable goal.

9.1 Further Work

The single most important piece of work that remains to be undertaken is the construction of an untagged, garbage collected programming language implementation that relies on automatically generated **Visitors** for its semantic coherence and that makes them visible to the user as a general-purpose programming tool. As a subsidiary goal this involves the devising of a good source-level interface for **Visitors** and visitation modes.

Additional respects in which visitation deserves further exploration include:–

- Implementation and measurement of the performance impact of visitation as an approach to marshaling and unmarshaling in operating systems.
- Better integration of visitation with modern garbage collection technology as developed for tagged heaps.
- Application of **Visitors** to languages like SELF, where all type information is dynamically introduced by an aggressive compiler rather than being present, even implicitly, in the source form (something we do not consider to be a good engineering tactic but may be useful for prototyping and certainly poses a unique technical challenge).

List of Formalisms

Programming Languages

Ada [dod83] U.S. Department of Defense programming language. Notable for having everything but a clear semantics.

Alfonso Normal-order pure functional textual database retrieval language under development (by the author, among others) at the University of Chicago's Center for Information and Language Studies. Based loosely on Alonzo [*q.v.*], and implemented with **Visitors**.

Algol 68 [vWMP+76] Seminal strongly typed imperative functional language with (unsurprisingly) "Algol" syntax. Algol 68 is garbage-collected and has a rich array semantics including reference-yielding array slices.

Alonzo [Ram89] Normal-order pure functional language with Scheme-like syntax.

APL [Ive87] Interpreted array manipulation language renowned for its bizarre character set (but see J). Unitypic type system, with good implementations using significantly more representations than are visible to the user.

ASM-86 Intel's original assembly language for the ever-popular iAPX 86/88 microprocessors. Provides nontrivial data structuring facilities.

Awk [AKW88] Unix text file scanning and manipulation language.

BASIC ("Beginner's All-purpose Symbolic Instruction Code") Family of once-popular, typically interpreted FØRTRANesque programming languages of varying abilities.

Bliss [WRH71] DEC systems programming language.

Brouwer [Dev84] Total-correctness language developed by Dr. H. J. Boom at Concordia and based on the incredibly rich type logic ITT [*q.v.*].

C [KR78, HS87, ansi89] Systems programming language of the Unix operating system, now widely used elsewhere. Remarkably insecure; only recent versions have provided paramter type checking.

C++ [Str86] Half-hearted extension of C [*q.v.*] aimed at getting on the object-oriented bandwagon.

COMMON LISP [Ste90] Grand Unified LISP [*q.v.*] dialect, now with Object Orientation!

Concurrent Euclid [Hol83] Systems programming language with strict segregation of "unsafe" features, explicit parallelism and *monitors* for synchronisation.

FORTH [forth83] An explicitly stack based language, implemented with an exposed threaded interpreter. Highly insecure.

FØRTRAN [ibm] A line-oriented language specialised for floating-point arithmetic, now trying to evolve into something like Ada.

Haskell [HPW91] "A Non-strict, Purely Functional Language," intended to serve as a common platform for research in the field.

ICON [GG83] An interpreted language with streamed expressions and pattern matching operators, evolved from SNOBOL [*q.v.*].

INTERCAL [WL73] A language specifically designed for the purpose of being *completely different* from every other language. Not entirely successful in this goal.

J [Ive90] A modern dialect of APL [*q.v.*] using the ASCII character set.

Lazy ML [AJ89b] A normal-order dialect of ML [*q.v.*], remarkable for its efficient implementation.

LISP [McC60] A programming language granting explicit programmatic access to the abstract syntax level, and originally based on the λ-calculus. All early implementations seem to have messed up the variable-scoping rules, and (perhaps as a consequence) an imperative programming style has become the norm, making the language by now remarkable only for its illegibility and (with certain notable exceptions) inefficiency. Of particular interest for its very small kernel of basic data representations, corresponding roughly to BNF in form.

Lithp [BS85] A Scheme-like LISP dialect [*qq.v.*] developed by Dr. H. J. Boom at Concordia University. The original testbed for visitation.

Mary2 [pen83] A systems programming language notable for its programmable surface syntax and link-time code generation.

Miranda [Tur85] A functional programming language.

ML [MTH90] Another functional programming language, originally conceived as a metalanguage for proof construction work, but ultimately more popular in more general contexts.

Modula-2 [Wir83] Systems programming language with modules and monitors.

Pascal [JW74] A teaching language addressing the didactic climate of the early '70s.

Perl [Wal] Perl ("Practical Extraction and Report Language") is a general-purpose Unix text-processing language providing interpretive access to most Unix system calls.

Prolog [CM87] Mistakenly believed by many to be a programming language, actually a primitive knowledge representation syntax with a remarkably inefficient operational semantics glued on at an angle. Spectacularly popular. Prolog's code format is very simple, and comparable to BNF; see the comment under LISP.

Russell [DD85] A programming language with first-class types.

Scheme [RC86] A very much cleaned up dialect of LISP [*q.v.*] with lexical scoping and first-class continuations.

SELF [HCCU90] A *classless* object-oriented language in which *all* operations are dispatched along a fully dynamic multiple-inheritance prototype chain, which is also the sole name resolution mechanism. Possesses an implementation of profoundly amazing efficiency, all things considered.

SETL [SDDS86] A language specialised for the manipulation of finite sets, once notable f r providing explicit facilities for the description of data *representations*.

SETL2 [Sny90] An updated SETL, adding modules and first-class functions, but deleting the data-representation sublanguage.

Smalltalk-80 [GR83] A class-based object-oriented language. Despite lacking for many years even tolerably efficient implementations on non-specialised hardware, the root of the current object-oriented programming fad. A wonderful language for its day.

SNOBOL [GPP71] A *very* line-oriented imperative string-manipulation language.

Stage 2 [Wai70] A macro processor much abused in the bootstrapping of Lithp.

Turbo Pascal Commerical Pascal dialect [*q.v.*] with extensive systems programming and string manipulation extensions.

VAX Macro-11 Assembler for DEC's VAX architecture.

yacc [Joh79] "Yet Another Compiler Compiler," actually a crippled LR parser generator. The precise mode of crippledom is that although semantic actions can be associated with productions, they are pieces of C code that are executed at reduce-time, communicating with each other by indexing into the stack in a most unsanitary manner. Very famous; to be avoided.

Other

Autolexical Syntax [Sad91] A theoretical linguistic superstructure developing the idea of multiple, parallel, independent linguistic modules.

GPSG [GKPS85] (Generalised Phrase Structure Grammar) Lexicalist syntactic theory factoring phrase structure rules into ID/LP (Immediate Dominance/Linear Precedence) format and making heavy use of unification.

ITT [ML84] (Intuitionistic Type Theory) A formal type theory employing the notion that a type can be identified with a proposition true if the type is inhabited. This duality is sometimes cumbersome (since every relevant property of an object has to be formally encoded in its surface form), but as a type system it is spectacularly rich.

λ-calculus [Bar84] Mathematical object providing the foundations for the functional model of computation. Based on the notion that $f : x \mapsto y$ can be interpreted as declaring f to be identical with a *function value* $x \mapsto y$ (classically written $\lambda x.y$).

LFG (Lexical-Functional Grammar) [Bre82] A unification and attribute-grammar based lexicalist linguistic theory.

NuPRL [CAB+86] (New (or "Nearly Ultimate"?) Proof Refinement Language) A theorem-proving system using ITT [*q.v.*] as its logic.

Bibliography

[ABB+86] M. J. Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for Unix development. In *USENIX Conference Proceedings*, Atlanta, Georgia, July 1986.

[acm90] ACM SIGPLAN. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990. Published as SIGPLAN Notices 25(6), June 1990.

[Ada79] Douglas Adams. *The Hitch-Hiker's Guide to the Galaxy.* Pan Books, 1979.

[Ada85] Douglas Adams. *The Hitch-Hiker's Guide to the Galaxy: The Original Radio Scripts.* Pan Books, 1985.

[AJ89a] Lennart Augustsson and Thomas Johnsson. The Chalmers Lazy-ML compiler. *The Computer Journal*, 32(2):127–141, 1989.

[AJ89b] Lennart Augustsson and Thomas Johnsson. Lazy ML user's manual. Technical report, Chalmers University, June 1989. †Lazy ML.

[AKW88] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language.* Addison-Wesley, 1988. †Awk.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programmes. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, California, April 1991. Published as Computer Architecture News 19(2); Operating Systems Review 25(special issue); and SIGPLAN Notices 26(4), April 1991.

[ansi89] American National Standards Institute. *American National Standard for Information Systems—Programming Language C*, 1989. †ANSI C [ANSI X3.159-1989].

[App89] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus: its Syntax and Semantics.* North-Holland, 1984.

[BDG+88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. COMMON LISP Object System specification. X3J11 document 88-002R. Published as SIGPLAN Notices 23(special issue), September 1988.

[BDM+86] Abraham Bookstein, Scott Deerwester, Robert Morrissey, Keith Waclena, and Donald Ziff. A system for integrated bibliographic and full-text retrieval in a distributed computing environment. In *Computers and the Humanities: Today's Research, Tomorrow's Teaching*, pages 285–291. University of Toronto, March 1986.

[Bre82] Joan Bresnan, editor. *The Mental Representation of Grammatical Relations.* MIT Press, 1982. †LFG (Lexical-Functional Grammar).

[BS85] H. J. Boom and Stephen P Spackman. Brief notes on Lithp. Technical Report PLSG-3, Concordia University Department of Computer Science Progamming Languages Study Group, 1985. †Lithp.

[BW88] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, September 1988.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. Y. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall, 1986. †NuPRL.

[Car] Luca Cardelli. A polymorphic λ-calculus with type:type. Technical Report 10, Digital Systems Research Center.

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, third edition, 1987.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *OOPSLA '91 Conference Proceedings*, pages 1–15, Phoenix, Arizona, October 1991. ACM SIGPLAN. Published as SIGPLAN Notices 15(11), November 1991.

[DD85] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[Dev84] Michel Patrick Devine. A program verifier for total correctness based on intuitionistic type theory. Master's thesis, Concordia University Department of Computer Science, 1984. †Brouwer.

[Dil88] Antoni Diller. *Compiling Functional Languages.* John Wiley & Sons, 1988. Actually about implementation techniques for normal-order languages, a nice introduction, survey and handy reference.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on the Principles of Programming Languages*, pages 207–212, 1982.

[dod83] United States Department of Defense. *Reference manual for the Ada programming language*, 1983. [ANSI/MIL-STD-1815A-1983] †Ada.

[DP80] D. R. Ditzel and D. A. Patterson. Retrospective on high-level language computer architectures. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 97–104. IEEE, 1980.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 System. In *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, 1984.

[DZW90] Scott C. Deerwester, Donald A. Ziff, and Keith Waclena. An architecture for full text retrieval systems. In *DEXA 90: Database and Expert Systems Applications*, pages 22–29, September 1990.

[forth83] Forth Standards Team. *Forth-83 Standard.* Mountain View Press, 1983.

[GG83] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language.* Prentice-Hall, 1983. †ICON.

[GKPS85] Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. *Generalised Phrase Structure Grammar.* Harvard University Press, 1985. †GPSG.

[Gol90] John A. Goldsmith. *Autosegmental and Metrical Phonology.* Basil Blackwell, 1990.

[Gol91] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 165–176, Toronto, Ontario, June 1991. ACM SIGPLAN. Published as SIGPLAN Notices 26(6), June 1991.

[GPP71] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language.* Prentice-Hall, second edition, 1971.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983. †Smalltalk-80.

[Gro85] Peter Grogono. *A Typed, Applicative Programming Language.* PhD thesis, Concordia University Department of Computer Science, February 1985.

[Gro86] Peter Grogono. Bias user manual. Technical Report PLSG-10, Concordia University Department of Computer Science Progamming Languages Study Group, October 1986. †Bias.

[HCCU90] Urs Hölzle, Bay-Wei Chang, Craig Chambers, and David Ungar. *The SELF Manual*, July 1990. †SELF.

[Hol83] R. C. Holt. *Concurrent Euclid, the Unix* System, and Tunis.* Addison-Wesley, 1983. †Concurrent Euclid.

[HPW91] Report on the programming language Haskell, version 1.1, August 1991. Available by anonymous ftp from nebula.systemsz.cs.yale.edu [128.36.13.1]:/pub/haskell-report, though there is a README file there that implies that people who do this are liable to prosecution. †Haskell.

[HS87] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual.* Prentice-Hall, second edition, 1987. A far more useful reference for pre-ANSI C than Kernighan and Ritchie [KR78] in that it attempts to describe something of the range of extant C implementations.

[ibm] IBM. *IBM System/360 FORTRAN IV Language.*

[iso87] ISO. *Information Processing—Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*, 1987. ISO International Standard 8824 †ASN.1.

[Ive87] Kenneth E. Iverson. A dictionary of APL. *APL Quote Quad*, 18(1):5–40, September 1987.

[Ive90] Kenneth E. Iverson. The ISI dictionary of J. Technical report, ISI, 1990. †J.

[Jac91] Steve Jackson, editor. *GURPS Basic Set.* Steve Jackson Games, third edition, 1991.

[Joh79] S. C. Johnson. Yacc: Yet another compiler-compiler. In *Unix Programmer's Manual.* Bell Laboratories, 7th edition, January 1979. †yacc.

[JW74] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report.* Springer-Verlag, second edition, 1974. †Pascal.

[Kan87] Gerry Kane. *MIPS R2000 Architecture Manual.* Prentice-Hall, 1987.

[KL89] Philip J. Koopman, Jr. and Peter Lee. A fresh look at combinator graph reduction (or, having a TIGRE by the tail). *SIGPLAN Notices*, 24(7):110–119, July 1989.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978. †pre-standardisation C.

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a dsitributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LB90] Vernon A. Lee, Jr. and Hans-J. Boehm. Optimizing programs over the constructive reals. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* [acm90], pages 102–111. Published as SIGPLAN Notices 25(6), June 1990.

[Li88] K. Li. Ivy: a shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Computing (Vol. II)*, pages 94–101, St. Charles, Illinois, August 1988.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[MH86] Scott McFarling and John Hennessy. Reducing the cost of branches. In *The 13th Annual international Symposium on Computer Architecture*, pages 396–403, Tokyo, Japan, June 1986. Published as Computer Architecture News 14(2), June 1986.

[Mil26] A. A. Milne. *Winnie-the-Pooh.* E. P. Dutton & Co., Inc., 1926.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[ML84] Per Martin-Löf. Intuitionistic type theory: Notes by Giovanni Sombin of a series of lectures given in Padua, 1984.

[Moo85] David Moon. Architecture of the Symbolics 3600. In *The 12th Annual international Symposium on Computer Architecture*, pages 76–83, Boston, Massachusets, June 1985. Published as Computer Architecture News 13(3), June 1985.

[MTH90] Robin Milner, Mads Toffe, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990. †ML.

[niso91] National Information Standards Organization, New Brunswick, New Jersey. *Electronic Manuscript Preparation and Markup*, 1991. [ANSI/NISO z39.59-1988] †SGML.

[pen83] Penobscot Research Center, Reach Road, Deer Isle, Maine. *Mary2 Language Reference Manual*, version G, December 1983. †Mary2.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* [acm90], pages 16–27. Published as SIGPLAN Notices 25(6), June 1990.

[PJL91] Simon L Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture (5th ACM Conference)*, number 523 in Lecture Notes in Computer Science, pages 636–666, Cambridge, Massachusetts, August 1991. Springer-Verlag.

[PW90] William Pugh and Grant Weddell. Two-directional record layout for multiple inheritance. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* [acm90], pages 85–91. Published as SIGPLAN Notices 25(6), June 1990.

[Ram89] John D. Ramsdell. The Alonzo functional programming language. *SIGPLAN Notices*, 24(9):47–56, September 1989.

[RC86] Jonathan Rees and William Clinger. Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[Reb91] Samuel A. Rebelsky. An introduction to *tours*, a protocol for demand-driven communication of terms. Technical Report CS91-27, University of Chicago Department of Computer Science, 1991.

[Sad91] Jerrold M. Sadock. *Autolexical Syntax.* University of Chicago Press, 1991. †Autolexical theory.

[SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, 1986. †SETL.

[Sny90] W. Kirk Snyder. The SETL2 programming language. Technical Report 490, Courant Institute of Mathematical Sciences, New York University, September 1990. †setl2.

[Spa86] Stephen P Spackman. Garkbit. Technical Report PLSG-7(?), Concordia University Department of Computer Science Progamming Languages Study Group, 1986.

[Ste90] Guy L. Steele Jr. COMMON LISP*: The Language.* Digital Press, second edition, 1990. The second edition incorporates X3J13 revisions, including the Common Lisp Object System [BDG+88]. †COMMON LISP.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986. †C++.

[sun87] Sun Microsystems, Inc., Mountain View, California. *The SPARC Architecture Manual*, version 7, 1987.

[Tom87] Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1–2):31–46, January–June 1987.

[Tom91] Masaru Tomita, editor. *Generalised LR Parsing.* Kluwer Academic, 1991.

[Tur85] D. A. Turner. *Miranda: A Non-strict Functional Language with Polymorphic Types*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[TvRvS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Amoeba system. *Communications of the ACM*, 33(12):46–63, December 1990.

[Ung84] David Ungar. Generation scavenging: a non-disruptive high-performance storage reclamation algorithm. *SIGPLAN Notices*, 19(5):157–167, May 1984.

[vWMP+76] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. *Revised Report on the Algorithmic Language Algol 68*. Springer-Verlag, 1976. †Algol 68.

[Wai70] W. M. Waite. The mobile programming system: Stage2. *Communications of the ACM*, 13(7):415–421, July 1970.

[Wal] Larry Wall. Perl. Unix manual page.

[Wen90] Alan L. Wendt. Fast code generation using automatically-generated decision trees. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* [acm90], pages 9–15. Published as SIGPLAN Notices 25(6), June 1990.

[WF77] David S. Wise and Daniel P. Friedman. The one-bit reference count. *BIT*, 17:351–359, 1977.

[Wil88] Paul R. Wilson. Opportunistic garbage collection. *SIGPLAN Notices*, 23(12):98–102, December 1988.

[Wil91] Paul R. Wilson. Some issues and strategies in heap management and memory heirarchies. *SIGPLAN Notices*, 26(3):45–52, March 1991.

[Wir83] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1983. †Modula-2.

[WL73] Donald R. Woods and James M. Lyon. The INTERCAL programming language reference manual, 1973. †INTERCAL.

[WRH71] W. A. Wulf, D. B. Russel, and A. N. Habermann. Bliss: A language for systems programming. *Communications of the ACM*, 14(12):780–790, December 1971.

[ZDW90] Donald A. Ziff, Scott C. Deerwester, and Keith Waclena. Using a functional language for textual information retrieval: Design and implementation. Technical Report 90-1, University of Chicago Center for Information and Language Studies, February 1990.

[ZWS92] Donald A. Ziff, Keith Waclena, and Stephen P Spackman. Using a lazy functional language for textual information retrieval. Technical Report 92-02, University of Chicago Center for Information and Languages Studies, 1992.

The mark † indicates a canonical reference for a language, theory or formal system.