A Lisp through the Looking Glass

submitted by J. C. G. Sturdy
for the degree of Ph.D.
of the University of Bath
1991

COPYRIGHT

'Attention is drawn to the fact that the copyright of this thesis rests with its author. This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognize that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.'

'This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.'

Summary

This thesis presents a new architecture for programming language interpreters, in which interpreters are not only first-class values, but are also arranged in a tower of meta-circular interpretation which is accessible reflectively—so that a program may modify elements of the meta-circular tower under which it runs, and thus cause changes in the manner of its own interpretation.

To facilitate such modification, we develop a representation for interpreters that splits each interpreter into a language (a collection of independently implemented constructs) and an evaluator (connecting the constructs together).

To implement such a mutable infinite meta-circular interpreter, we need another interpreter outside the tower, the meta-evaluator. We present this, along with a systematic way of linking it to the meta-circular tower. We show that a further form of meta-circularity may be introduced by bringing the meta-evaluator into the reflectively accessible part of the system; and that this may be repeated without limit, using the same techniques.

These techniques for meta-interpretation are then shown to be similar to the "language and evaluator" model for interpretation, and a concise version of the system is presented that uses common code for many of these functions.

Provision for reflective and mixed language facilities pervade the infrastructure of the system. We show that despite all this power, it is possible to implement such a system efficiently—well within an order of magnitude of the performance of a single-language non-reflective system—and we show how a wide range of languages may be implemented on this infrastructure, which also allows transparent mixed-language programming.

Acknowledgements

Acknowledgements are due to many:

- *God* for making the universe so very reflective
- *Julian Padget* for supervising and encouraging me
- *James Davenport and John ffitch* for assorted ideas and advice
- *Phil Yelland* for discussions of the more mathematical parts of reflection
- *Jim des Rivieres*, *Olivier Danvy* and *Karoline Malmkjær* for comments
- *My family and friends*, particularly Nicky Sanders, for encouragement
- *Harlequin Ltd* for providing facilities while I was working part-time on this thesis, and for writing the Lisp system I used
- *My colleagues* at Harlequin, especially Mark Tillotson, Colin Meldrum, and Pekka Pirinen, for assorted discussions on Lisp-related topics and advice on getting the fastest code from the compiler
- *The friars* for putting up with me at their guest-house, and particularly Brother Edmund for discussions on the metaphysics of reflection
- *Other research students* whose work overlapped with mine in time, particularly John, Russell, Val, Phil, Mary, Andrew, Daniel, and Isobel
- And, of course, *SERC* for funding the project—ever present, but usually only visible as a series of forms shadowing what was actually going on.

Contents

Chapter 1

Survey

1.1 The background to this thesis

An active area of research in Computer Science concerns the manipulation of programs by other programs, and investigates the use of automatic logical reasoning systems to reason about programs. It has come to the attention of many researchers that, if it is useful for a program to manipulate, and reason about, other programs, it may be even more useful for it to be able to manipulate and reason about itself—that is, to *reflect* on itself.

Reflection differs from one program working on another program in one important respect: changes made by the program to itself must affect the behaviour of the program, and, correspondingly, the behaviour of the program must affect what the program sees of itself. Thus, there is a *causal link* between the program as agent and the program as subject. A overview of the implications of this, with a survey of some seminal work in the area, is given in [**?**].

To be able to make sense of a running program as a subject, we must take it in the context of

- the language in which it is written
- the active evaluating agent that interprets it as a program in that language

Therefore, in this thesis, we include the language and the evaluating agent as part of the subject program. Once we make the language part of the subject, we are handling languages as data values, and so can also handle systems that allow each program to be written in a mixture of languages.

By including the evaluator (or interpreter) explicitly in our model of computation, we add another layer, or level, to the interpretation; as we look further into the model, we find that the evaluator itself must be interpreted likewise, and so on, and so we add an infinite number of levels of interpretation [**?**]. It has already been shown [**?**] that it is possible to implement an interpretive system for a single language using this infinite number of levels of interpretation, with little less efficiency than a normal interpreter. In this thesis, we investigate the

possibility of doing this for a system with explicit and variable languages; and at the same time, in investigating further the nature of interpretation itself, we change the infinite tower of levels of interpretation to an infinite tower of infinite towers ... of infinite towers of levels of interpretation.

Though a tower of interpretation is slightly more complicated to construct than a plain interpreter, it makes it simpler to modify the interpreter later, for example to add new language features. When used as a construction kit for building mutually compatible interpreters in a mixed-language environment, the slight extra complexity is adequately compensated for by the ease of adding features and even whole languages within the framework.

Just as the tower model simplifies the construction of individual components based on it, the meta-tower simplifies the construction of the tower in a very similar way, and, since the relationship of successive meta-towers is the same as the between a tower and its meta-tower, it can also be said to simplify its own construction, through the development of a regular model for towers and meta-towers.

The implementation of a system using this model turns out to be concise and elegant, and when refined as far as possible appears to be rather simpler than the ad-hoc systems of which it is a replacement and superset.

When we include the evaluator as part of the subject of the program, we give the program access to the thing which defines the *meaning* of the program–and through this, the program can manipulate its own meaning or interpretation.

Since this involves operations on the interpreter of the program, the meaning of the interpreter must be defined in some way. The interpreter is a program (hardware interpreters–computers–are outside the scope of this thesis) and so its meaning is determined by the interpreter of that program. Thus, we see an infinite tower of interpreters defining the meaning of a program. For a program to inspect and manipulate its interpreter meaningfully, it must also have access to the whole tower of levels of interpretation (which, for practical purposes, come into existence only as needed–a lazily generated tower).

1.2 Reflection into context

Reflection is not simply a programming tool, but a general model for describing a wide range of things that perform processes in some intelligent manner. While this thesis is confined almost entirely to Computer Science, it is often helpful to see the field in the contexts of psychology and philosophy, harnessing the anthropomorphising tendencies of the human mind to make useful analogies regarding self and meaning.

Since the search for the definition of the meaning of an action (process,program, utterance, trace) leads toward our projection of meaning onto a free formula—something which does not have meaning until taken in a particular context—some philosophical digression from the mathematical logic is used in places to give a meaning to the mathematics uttered.

1.3 Related areas of programming language research

In constructing a system for handling programs and their environments, we must draw from several areas of Computer Science, as well as adding new material concerning the causal link between action and behaviour. While using techniques from these areas, we also contribute techniques that they can use.

Grammars

A program must be written according to the *grammar* of the language in which it is written. To manipulate programs, we must have formal grammars of the languages in which they are written.

This area has been researched extensively, and the results of this research are in common use, not only for constructing programming language systems, but also in many other areas of information processing, as all computer-manipulable information must be in a form describable by a grammar.

In this work, the grammars we use (both formally and informally) are those describing programming languages—particularly procedural programming languages. To bring a language into the system developed here, a grammar for it must be provided, describing the lexical and syntactic aspects of the language, to enable some program, called a *parser* or *lexical analyzer*, (with which we are not concerned here) to convert (parse) the textual form of each program into a *parse tree* or *abstract syntax tree*. This conversion may be a bare transliteration of the program text, or it may include annotation—an *augmented syntax tree*—for example, global and local variable references may be distinguished explicitly in the parse tree. Some languages, notably non-procedural, non-functional ones, may require considerable processing at this stage; for example, prolog programs could be *procedurized* here.

Many tools for generating parsers are available. The best known are perhaps those provided on the Unix system: Lex (a lexical analyzer generator) and YACC (a syntactic parser generator, also sometimes used to generate the main framework of a compiler). These work across the boundary between textual data processing and symbolic data processing, and both are mixtures of an existing, general-purpose programming language, *C*, with specialized languages. Lex and Yacc (or similar tools) may be used to complement the work behind this thesis; and the work presented here helps, in turn, to formalize the language mixtures in which Lex and YACC grammar descriptions are written.

This thesis looks very little at the lexical and grammatical levels of language processing—it is treated as being largely a separate task. However, a full mixed-language system would have to be able to accept grammatical descriptions as well as the semantic descriptions that the present system uses.

Interpretation

Reflective program execution is both a form of interpretation, and a tool for studying interpretation. Some techniques from writing conventional interpreters apply here; more significantly, reflective programming is a new technique for constructing interpreters, and may be used as such either by itself, or in conjunction with other techniques. It also provides a framework in which to reason about interpreters.

The reflective model of interpretation has several advantages over ad-hoc interpreter architectures. Not only does it make it possible for an ordinary program to extend the interpreter, but it also makes the design and implementation of interpreters more systematic–perhaps making it less of an art and more of an exact science.

With a well-defined model of interpretation, constructing and debugging interpreters becomes easier, and it is possible to write tools to assist in writing and testing them.

Partial evaluation

Partial evaluation is an interpretation technique in which a program is run as far as possible on the data available to it at the time, reducing it to a *residual program* which accepts some or all of the remaining data to procede to another residual program or to the result. Whereas reflective interpretation tends to add new levels of interpretation to the execution of a program, partial evaluation can remove levels of interpretation by combining them with adjacent levels, which is done by evaluating the interpreter with the program that it interprets as input. Thus, partial evaluation and reflection may have complementary effects on the interpretation of a program. This effect is described and explored in [**?**], which explains how partial evaluation and reflection may be used together, giving the advantages of reflection, while collapsing the extra layers that it can introduce.

Compilation

Compilation, like partial evaluation, reduces a program, and so in some sense opposes reflection. It is sometimes possible to compile a reflective program into a non-reflective one, thus allowing systems developed under reflective interpretation to become more efficient compiled programs.

Mixing

Mixing [**?**] is a form of partial evaluation that allows for self-reference by the evaluator, and, through this, allows the evaluation to specialize interpreters to run particular programs. This is a form of compilation, as it reduces an interpreter I in a language L_i and a program P in a language L_p to a combined program IP, with the same meaning as P, but in the language L_i. It can also evaluate itself, M (the Mix program) with respect to any interpreter I_a in

language L_a, leaving a residual program MI that accepts programs P_b in any language L_b and produces combined programs IP_a—this residual program is a compiler. Mixing the mixer with itself produces a residual program that accepts interpreters I_a to produce residual programs MI_a—this is a compiler-compiler. Although this self-reference is not causal, there are some strong similarities between mixing and reflection, as both relate program and interpreter [**?**].

Abstraction

Handling data obtained from the state of a running program requires a suitable abstraction for program interpretation; such abstractions are discussed in [**?**] and [**?**]. Such an abstraction also provides a tool for further abstraction in the field, as it describes causal self-reference abstractly.

The abstraction must describe not only programs in their static form, but also the active process of running them. As it turns out, a suitable abstraction for this is reflective interpretation itself—the very feature that required the abstraction in the first place. Such cycles of reference as this are natural in this work, as the systems described and implemented include descriptions and implementations of the abstractions used in their descriptions and implementations. This goes against the traditional form of semantic definition, of which the main aim is to describe the system in terms of things outside the system itself. Here, we prefer to include a description of the system within the system, and describe how something outside the system may project a meaning onto the system, and thereby may give it an external definition. We then show how a system may be constructed that includes within itself a description of how an outside system may project a meaning onto the system.

1.4 Why do we want reflection?

Reflection is useful both as a tool for abstract description and manipulation of programs and other active agents, and as a practical programming tool. Active agents here can mean anything that has defined behaviour and processes information, such as a computer, a program, an organization, or an animal. Reflection describes a causal link between abstract information handling and concrete behaviour. In mentalistic terms [**?**], it is the part of introspection that links the self as the subject of reasoning to the self as reasoner. This works in both directions: it links observation of the self to reasoning, and reasoning to action on the self—which is part of learning. A fully reflective agent can reflect upon reflection, and can thus can observe observation, reason about reasoning, and learn about learning.

Reflection is interesting and general

The idea of reflection is useful for describing either reflective or non-reflective interpretation of programs. It can describe programs, languages and processing

agents. It describes the meaning of any of these, and, as described in this thesis and other work [**?**] [**?**], relates the *extensional* meaning of something—that is, its description, or what the thing means—to the *intensional* meaning, that is, how it works, or its implementation. It does this by putting the intension into a frame of reference which binds it to the world outside the description, that is, the extension.

In mentalistic terms [**?**] we manipulate our surroundings (as we perceive them) in terms of abstract concepts, a form of manipulation that is widely accepted as being deeply connected with our linguistic faculties. For example, hammering a nail involves the concept of momentum; even if someone does not have a spoken name for it, they must have this specific (and hence nameable) concept in order to understand why hammering is more effective then steady pressure applied with the hammer. Earlier work on procedural reflection allowed programs to modify their language environments but did not have a specific model for languages as data values. Here, we make it possible to reflect on languages more formally and abstractly than before.

Describing and implementing languages

Reflection, in the form used here, links a program with its interpreter in such a way that the program can modify the interpreter [**?**]. By adding new features to the language, then removing the features of the original language, a program can construct an implementation of another language, and continue to run in that language instead of the original one.

This is a novel approach to language definition, and offers interesting prospects for programming in general. Not least among the possibilities is that of tailoring a application-oriented programming language for a particular program, as part of the application program and in terms of the application-oriented language rather than as part of a separate language processing program written in a language less suited to the application domain.

This form of language implementation also makes language development and debugging more like other application development. Debugging facilities provided within the language system can be used on the language system, and new debugging facilities (for example, printing messages at points in the code) for working on the language can be inserted just as they can be into an application.

Program debugging

Reflection provides some useful facilities for debugging programs, such as access to the program's state as data—this data can then be passed to an inspection tools, as is done in SmallTalk-80 [**?**]. Such data can then be modified, and installed back in the program to continue running it. The routines to display, manipulate and re-install the reified data may be written as part of the application program (and thus in the language of the application, not of the interpreter), and so the debugging facilities can be presented in terms of the application domain.

New possibilities

Reflection opens some new possibilities; one example that has been investigated (but not found to have any important practical use) is *migration* [**?**]: for a program to transmit its state and data to another host computer, and continue running on its new host. It has been demonstrated that such a facility can be made machine-independent, and so can be used to make programs that install themselves intelligently on new machines.

Another possible application of reflection is explanation in expert systems, where a program can go through the code and state of its own reasoning to explain why it reached a particular conclusion. An example of a system that does this is SHRDLU [**?**], a system for manipulating solid block in response to natural-language commands. SHRDLU is capable of answering questions about why it had to make particular parts of a sequence of movements (for example, moving a block to be able to pick up one from beneath it).

Since reification takes a snapshot of a running program (or returns a handle into the running copy of the program), and reflection resumes the execution of the snapshot, it is possible to take checkpoints, either in case of the program or computer crashing later, or for holding versions of a program in various useful states: for example, a PostScript program could be frozen at the end of each page (thus collecting all the global state changes accumulated by running all the preceding pages) to make a collection of programs ready to run any specific page without working through all the preceding ones.

1.5 Approaches to reflection

There are two approaches to reflective interpretation:

- *flat reflection*, in which the program is given access just to itself and, perhaps, to its own processing agent. This is described in [**?**].
- *tower reflection*, in which the program also has access to its processing agent's processing agent, and so on, infinitely. This is described in [**?**]; [**?**] presents a summary of this, which is more accessible.

Of these two approaches, tower reflection is a better study of the general process of interpretation, as it requires a general definition of the link between a running program and its interpreter. It is also the more powerful, as it allows successive transformations on the interpretation process to be composed (for example, lazy evaluation and program tracing).

1.6 Why mixed languages?

Writing programs in a mixture of languages has been possible for some time now, and there have been many formal descriptions of individual languages. However, little work has been done to describe mixed-language programming,

except in some specific cases such as the Poplog system [**?**], which combines Pop-11, Lisp and Prolog. Mixed language programming has shown its usefulness in several ways; for example, in creating new languages which draw on features of existing language (*make* using *sh* on Unix is a common example); and in accessing obscure features (such as `asm` statements in C; FORTRAN-based arithmetic libraries called from a variety of languages). Some problems have solutions in which different areas of the problems are best tackled by different languages, for example the `pic | refer | eqn | tbl | troff` combination of text processors on the Unix system, where each of the components is good at just its own proper task, but either weak or completely incapable in the other areas (for example, making a table with `eqn` or a mathematical display with `tbl` or `refer` is difficult and generally inappropriate).

1.7 Summary of survey

The mechanisms of program interpretation may be analyzed into several areas, some of which are currently active research areas. These include program transformation, partial evaluation, reflection, and mixed-language programming. This thesis concentrates on the latter two.

Reflection—the causal link between actions of a program and its state, text, behaviour and environment—combines ideas from interpretation theory, logic, mathematical philosophy, linguistics, compilation, abstraction and other fields of Computer Science. It also contributes new techniques which may be used in these, and other, fields.

Mixed-language working is already common practice, but has not been formalized. Its existing use argues strongly for its usefulness, and the limitations on its present use, and its current haphazardness argue for further development of the ideas underlying it.

In common between these fields is the systematic definition of programming language interpreters, and the idea of provision of meaning for a value in terms of the context surrounding it (including its interpreter).

> "I should see the garden far better," said Alice to herself, "if I could get to the top of that hill: and here's a path that leads straight to it—at least, no, it doesn't do *that*—" (after going a few yards along the path, and turning several sharp corners), "but I suppose it will at last."
>
> *Alice, Through the Looking Glass, Chapter 2*

Chapter 2

Introduction

The design of programming systems has traditionally been divided into two parts: the *program* being executed, and the *evaluator* executing it. The techniques for designing and implementing each have been kept largely separate, although they have much in common, particularly when the evaluator is itself a program (an *interpreter*). This work aims to bring them together, allowing the techniques of both to be used in each, by adding some special facilities to an interpreter. My thesis is that these facilities may be added, and this without unreasonable overhead in program execution time. Since this field is connected largely with program and language *development* work, this means that the overhead must be reasonable in the context of a large-scale prototyping system, a factor of ten being chosen as a reasonable limit. A program or language developed on such a system can then be delivered on one in which these facilities are either attenuated or removed altogether, thus bringing the speed of execution up to that of a comparable conventional system. In researching this thesis, I developed several versions of a language implementation framework *Platypus*, which have tended to support this thesis.

2.1 Languages, interpreters and programs

In the terms used here, a *programming language* (or just *language*) is a collection of *operators*, with definitions of what each operator means, and rules relating the different operators. An operator, in this sense, is the part of an expression or statement that determines what the rest of the statement means—that is, how the expression is to be interpreted. It is a *token* or *symbol* value, which is distinct from other such symbols, but, in itself, carries no further meaning. To have meaning, an operator symbol must be taken in the context of its language.

A *program* is a specification of some actions or functions, written in a programming language. Each function has a *name* and a *body* or *expression*. The language defines the meaning of the program by defining the meaning of each expression of the program and the way the parts are connected.

An *interpreter* uses the meaning inside the program to cause actions which are outside the program but inside the interpreter. The interpreter then causes action in the computer's peripherals to produce an effect that has meaning to an observer outside the computer. Thus, the meaning of the program is extracted from its representation in the program, and hence mapped into a different representation in a different domain.

An interpreter for a language puts these actions to work, using the rules to find its way from one operator to another. It is an active definition of the meaning of each operator and of the structure of the language.

Without a language, the program is without meaning. The program by itself is just an abstract formula in a *free algebra*,which does not express anything. To have meaning, it must be understood in the context of a particular language. To clarify this, consider this sentence. Without English, it would make no sense.

Taken with a language, the program can be manipulated between equivalent forms, but expresses nothing. Such manipulations are similar to reading a sentence in a natural language while writing out definitions of each word found from a dictionary and using structure descriptions in a grammar book; this performance does not need any interpretation of the sentence in any language. The program by itself is purely a data value of a type specified by the language.

But a language also can be regarded as a value in a free algebra, which can be manipulated, and applied to programs (or have programs applied to it). Continuing the analogy with natural languages, it is like a a combined dictionary and grammar book: it can be used to transform sentences, without understanding the language.

Neither the program nor the language have any further meaning unless handled by an active processing agent that makes it do something in the *structural field* [**?**] of that agent, that is, the world of things that the agent knows about and manipulates. We call such an agent an *evaluator*. The evaluator might be the circuitry and microcode of a computer, or perhaps it might be another interpreter. The evaluator, which is a concrete representation of an abstract language, gives meaning to the program, and in turn the program gives meaning to its input data, that is, makes results from them. The language is given its meaning by the evaluator, which is part of the interpreter—an interpreter is a combination of language and evaluator. This gives us

where the evaluator provides the meaning, which, in the traditional understanding of computation, it conjures up from nowhere in particular. In this thesis, we examine the way in which each part of a computation involving evaluator, language and program gives meaning to the other parts, and the way in which a meaning can be given to the outermost evaluator so that the evaluator can then give meaning to the rest.

To clarify this, consider this sentence. With no-one to read it, it means nothing to anyone.

Later in this thesis we refer frequently to *languages*, *interpreters* and *evaluators*, and it is important to distinguish between them. As we use the terms

here, a *language* is analogous to a combination of grammar book and dictionary, as described above. An *evaluator* is an active agent, capable of doing something with a language and a program. An *interpreter* is an evaluator equipped with a language, which is then capable of doing things with expressions in that language.

An algebra of interpretation

In order to compute about computation, we must be able to represent information about computation, in a form which can be manipulated by a program. One such form, that has been used extensively in research on computation, is the *partial program*, which is evaluated through *partial evaluation* or *reduction*. Partial evaluation of a program proceeds as far as possible with the parameters available, leaving a partial program which will accept the other parameters giving another partial program. The object that remains once all parameters have been accepted and substituted is the result of the evaluation.

For example, the partial program

$$\lambda ab.(a + b) \tag{2.1}$$

when partially evaluated with just one argument, produces a partial program which adds that first value to a value that it accepts as its only argument:

$$(\lambda ab.(a + b))2 \rightarrow \lambda b.(2 + b) \tag{2.2}$$

which when applied to a second argument produces the result

$$(\lambda b.(2 + b))3 \rightarrow 2 + 3 \rightarrow 5 \tag{2.3}$$

Application of interpreters to programs, and in turn of computers to interpreters, is described in an abstract form in the Futamura projections [**?**] and implemented in the *Mix* system [**?**].These projections use partial programs throughout, and describe how an interpreter may be mixed with a partial program to specialize the program further, thereby compiling it. Partial evaluation for Scheme is described in [**?**].

In Mix, the partial program representation is based on Lisp and on the lambda calculus. An interpreter is simply a function, as is the program, and the two are brought together by an external mechanism, the partial evaluator (or Mixer). In this thesis, the structure of a program, and that of the interpreter, are again similar, but we add a mechanism which links each program to its interpreter, and defines the relationship between them. This is a formal description of the connection between a program and its interpreter, and enables us to reason about the abstraction of interpretation and to compute with its representation, which is not possible in non-reflective systems.

Including the computer in the software

In this thesis, we examine ways of including the evaluator (which could be a computer, or an program running on a computer) within our system for de-

scribing the meaning of things. This enriches our ability for giving meaning to computational processes and values, and allows us to manipulate those meanings both formally, for reasoning about them, and computationally, for handling their representations as program-as-data.

Let us now look at some properties of the components of a system that computes. In listing these properties, we will call the specification of what something does its *algorithm*. In functional programming, the same specification would be called a *function definition*. We take *algorithm* to be the more general term, and to embrace functional definitions. Proponents of algorithmic languages do not consider a language not to be algorithmic if it allows function definitions; whereas proponents of functional languages deem a language not to be truly functional if it allows algorithmic definitions. It is in deference to this that I have treated algorithm as a more general term than function—the two are both fully general.

- An evaluator is an *active* entity—that is, it performs some action (a process) upon something (a subject) in a structural field
- An evaluator is also an *algorithmic* entity—the action that it performs is prescribed by an algorithm

The above two items are the full defining properties of an evaluator. We can also see the following properties of programs:

- A program is an algorithmic entity.
- A program being evaluated is an active entity, by virtue of the action of the evaluator that evaluates it.
- A program being evaluated acts on its subject.
- A program being evaluated is thus also an evaluator, for its subject.

and from these properties, we can extend the formula on page 18 into a more regular structure, in that neither is the program any longer a singular piece at one end of the formula, nor is the evaluator a singularity at the other end:

In this formula, the evaluator in one system is equivalent to the program of a second system. The evaluator of the second system is in turn a program running in another system, and so *ad infinitum*. Each of these systems is known as a *level* of *meta-circular interpretation*, and the chain of levels is called a *tower*. This thesis is concerned both with the building of such levels, and with the ability to pass information between adjacent levels.

While much has been written about the theory and practice of program design and about the design[1] and implementation of languages, little has been

[1] Actually, not so much about the language design as the collections of features included in the language.

done to link theory and practice with a theoretically sound and soundly practical bridge. The ideas of *structured* and *functional* programming reach out across this gap, but cannot address the link between programming *in* a language to the programming *of* the language—that is, its implementation. They can only address issues concerning one realm of programming—either the program in a given language, or the program in some language, that incidentally implements a language (either the same one or another one). The latter is the programming *of* a language.

To address this gap a way is needed to apply developments in the technology of software implementation to language implementation. Such a way may be found through defining language interpretation as a programmed process, with a definite model for how an interpretation is computed. With this new model, language implementation is now described as a specific form of programming, with a formal connection between the problem domain and the program domain.

Some possible reduction in execution speed, mentioned at the start of this chapter, might come from making interpreters conform to this model rather than taking an arbitrary form. One of the aims of this thesis is to find how efficient interpreters in this model may be, compared with traditional interpreters.

2.2 Terminology: Reification and Reflection

A pair of techniques called *reification* and *reflection*, not found in most computing systems, begin to build a bridge between languages and the programs which are written in them. They build this bridge by describing the domains of the elements and rules from which the language is built, in terms of the domains of values which can be handled by the program, and allowing the program access to itself and to its interpreter in these terms. Use of such access for inspection is called *reification*, and for modification is called *reflection*. For example, a language which allows a program to store a stack frame for use as part of the target of an interprocedural jump (such as `longjump` in BCPL and C) provides a *reifier* for stack frames. The *reflector* for stack frames is part of the actual jumping operation.

Terminology

Reification and *reflection* are complementary actions. In some of the existing literature, the term *reflection* is used for them both. Likewise, a system architecture providing them may be known either as a *reified* architecture or as a *reflective* architecture. In this thesis, we use the two terms mainly in their proper rôles. When a general term for both together is needed, we use *reflection*.

An operator that returns part or all of the state and code of the system is called a *reifier*, and one that sets it from its argument is called a *reflector*. Reifiers and reflectors that operate on the whole state we call *grand* reifiers and reflectors. Those that operator by evaluating a piece of code in a new context, in which reification or reflection has been done which leaving the original context

unchanged, are called *pushy* reifiers and reflectors, as they push a new context to reify or to reflect (like a procedure call pushing a return address) whereas those that return or set things in the current state are called *jumpy* (like a `GOTO` jumping without saving any return address). (The terms jumpy and pushy come from [**?**].)

We will use the term *program text* or just *text*, or *code*, for the representation of a program. The actual representation of the program is immaterial. The text is certainly not necessarily stored as actual text. In practice, we store it as a parse tree. This way, all considerations of the concrete syntax of a language are removed from the discussion (see section **??**), and we can concentrate on the more abstract aspects of the language.

The state of execution of a computation—the combined contents of all its variables, along with the current point of execution and the stack of saved execution points at procedure calls—we refer to as the *state* of the program.

An *extensional* definition of an operator is given in terms of absolute meaning—what the operator is meant to do—whereas an *intensional* definition is in terms of how it works. In tower reflection, this distinction coincides closely with the distinction between operators used to implement a tower from outside, and operators inside that tower.

The *closure* of a procedure is a value that represents the text of that procedure along with any data that it requires, that is, with the bindings of any variables that it accesses but does not define. We say that the text and variables are *closed over* by the closure.

For use in mixed-language interpretation, we extend the closure to contain also the language and a means for applying the language to the rest of the closure. This we call an *interpretive closure*, but we will often simply refer to it as a *closure*. For consistency, we close the rest of the state into it, too.

Referential transparency is the property of a language by which an expression may be made into a separate (named) function, and then a reference to that function (by name) used transparently anywhere where the original expression was used. It is important for abstraction.

Procedures here may return results. Like Scheme [**?**], we call functions (in the computational sense) *procedures*, keeping *functions* for mathematical functions. Here the extensional equivalent of any given procedure is a function. Procedures are, implicitly, intensional.

Notation in formulae

We use a notation similar to that used for reduction in [**?**, Chapter 3]. We write "x interprets y" as

$$x \to y \tag{2.4}$$

and "i_m interprets a_m in n steps of computation" as

$$i_m \xrightarrow{n} a_m \tag{2.5}$$

We write "y is accessible (visible, reifiable) to x" or "x looks at y" (these are equivalent) as

$$x \blacktriangleleft y \tag{2.6}$$

which reads in words as "x looks at y".

Where we use two distinct symbols for equality in describing values, we use '$a \equiv b$' to mean 'a and b are (structurally) equivalent values' (`equal` in Lisp terminology), and '$a = b$' to mean 'a and b are the same value' (`eq` in Lisp).

A reflective system, although it may appear to make the evaluator on which it runs into a manipulable data value, must eventually be executed by an evaluator running on a conventional language system. We call this the *substrate evaluator*, and the language in which it is implemented, the *substrate language*.

2.3 Forms of reflection

A program is a value in the domain handled by the interpreter. When a program is run by an interpreter that provides reflection, the program has access to its own text and state, because reflection can provide access to all the information that the interpreter handles.

In the following diagrams, the boxes represent levels of towers, and the arrows represent the flow of information between tower levels.

Flat reflection

The simplest form of reflection is portrayed like this:

The interpreter has access to the information that represents the application (lower arrow) and the application has access to the information representing the interpreter (upper arrow). The application can get information about itself via the pair of arrows (drawn with a u-turn just inside the interpreter). Therefore, $\texttt{program}_0 \blacktriangleleft \texttt{interpreter}_0$, as well as $\texttt{interpreter}_0 \blacktriangleleft \texttt{program}_0$.

Tower reflection

In this simple form of reflection, this access goes no further (see section **??**). In the form of reflection investigated in this thesis, called *tower reflection*, the program is also given access to its interpreter's text and state, as well as the program's own text and state. This is in the same terms as access to the program itself, since the interpreter is also a program:

Here, $\texttt{prog}_0 \blacktriangleleft \texttt{int}_0$, $\texttt{int}_0 \blacktriangleleft \texttt{prog}_0$, $\texttt{prog}_1 = \texttt{int}_0$, and $\texttt{prog}_1 \blacktriangleleft \texttt{int}_1$, and $\texttt{int}_1 \blacktriangleleft \texttt{prog}_1$.

This access is modelled by including the program's interpreter in the representation of the program, like this (we are drawing these diagrams with just single arrows from now onwards, for clearer overall diagrams):

Since the interpreter is a program in the same form as the interpreted program, and the interpreter of any program may be found from the value representing the program, an infinite *tower* of interpreters appears, with information being passed up and down the tower in stages:

Here, for example, the program (prog0) can get information about its interpreter (int0) by asking it to ask *its* interpreter (int1); int1 sees int0 in the context of being prog1, so the operations required are just the same as in the flat reflection described above.

This thesis explores tower reflection, in which programming languages and the components of their implementation are first-class values that programs can inspect, manipulate and construct.

2.4 Program interpretation

In a conventional interpretive computing system, two programs are running at once: the *application*, which is commonly referred to as the *program*, and often thought of as the only program active in the system, and the *interpreter*, which may be a program or the circuitry and microcode of a computer.

The sense in which both programs are running at once is quite different from concurrency. Each step taken by the computer advances the state of *both* programs, by the same action, in different ways. One action of the computer has different meanings at the two different levels. Also, a single step at the application level is the same piece of work as many steps at the level interpreting it. A familiar analogy to this is found in the action of a digital clock. At 8 seconds past midnight, one tick will change the seconds digit, but nothing else. The next tick will change that digit again, and also change the tens of seconds digit. Many ticks later, a single tick will advance both of those, from 5 and 9 to 0 and 0. It will also have a third effect, of advancing the minutes by 1. Many more ticks later, the tens of minutes will change... and after still more identical ticks, one more tick— itself just the same as all the others—will have at least seven distinct meanings, including "a new second has begun" and "a new day has begun". However, the days are *composed of* the seconds (and so decomposable back into seconds), rather than running *parallel to* the seconds.

Another very relevant point in this analogy is that there are causal relationships between the adjacent digit devices of the clock. "Tens of seconds" changes only when told to by "Seconds". In turn it tells "Minutes" to advance at the proper time.

The metaphysics

The metaphysical aspect, regarded by many authors, including myself, as necessary in work on reflection, in the clock analogy is that the progression of the

digits of the clock does not cause time to advance. Time (an abstract thing) advances independently of the clock (a concrete thing). The activity of the clock merely models the progression of time. It does this because *we* choose to see the clock as representing the time. The same basic mechanism, with little modification, could be used to model or represent many other monotonous processes, such as the progress of a vehicle along the road. Likewise, an interpreter is only interpreting something, and an evaluator evaluating something, because we see it that way. Although this may seem like a frivolous aside, it is an important part of the background to reflective computation, particularly to the underlying type theory, which, in describing how a is represented by b also implies—through the action of an understanding agent—that b means a. By *understanding agent*, we mean something that takes in meaning (or projects meaning onto something), such as a person watching a screen. No meaning can be given to any of the system if no outside agent projects a meaning onto it in the terms of its own understanding. This is part of the *symbol grounding problem*—the ultimate need for a frame of reference that we can take as absolute[2].

This need for an external agent to project meaning onto something which can then be seen as an agent (but only in the light of that projected meaning) is examined in more depth and more breadth in section **??**. There, and further on in the thesis, it is shown that the process of projecting meaning onto a process must itself have meaning projected onto it; and also that several apparently different forms of the process of projecting meaning onto processes are shown to be equivalent in both their intensional and extensional expressions. These parts of the study combine to reflect on the meaning of meaning in the context of a system that contains causal self-reference, and place that context inside the context of a system which is observed from the outside. The projection of meanings within the system, in turn, may be taken to reflect on the inclusion of that external observer as part of the system. What matters is neither the number of levels nor the dimensional complexity of the relationships between them, but the difference between looking inward into the system and outward from within the system. Looking inward, we can construct a system and move our attention into it, but looking outward cannot move attention to anything beyond the boundary of the system.

Program as data

The interpreter handles the *text* and *state* of the application as data values. The text is some representation of the program code, and the state is the variables, point of execution, and the stack. From the interpreter writer's point of view, an application program is a collection of data objects—for example, the text of the program might be stored as a parse tree, the variables in a hash table and the stack as a vector, each in a variable of the interpreter—or possibly all in one variable of a suitable structure (record) type.

[2]Ceci n'est pas une thèse—c'est seulement une impression d'une thèse. [**?**]

Language as data

From the application writer's viewpoint a conventional language implementation is largely a black box providing little scope for adjustment apart from the extension techniques already mentioned. Although the application program may occasionally handle values passed to it by the interpreter *de profundis* (such as error codes) these values have little significance and is certainly not a formal model of what the interpreter really is.

For computing about language interpretation, we need to make languages available in the form of data, as well as presenting programs and their state as data. Central to this thesis is the development of a suitable representation for programs in languages, the interpretive closure. In the interpretive closure, the semantics of the language are defined in two parts: the *evaluator*, which drives the interpretation, and the *language*, which defines all the operators used by the language. Reflective operators are used in both parts of the language definition.

Evaluator as data

The evaluator of the system is a program, and we represent it in the same way as any other program, that is, as an interpretive closure. The closure representing the evaluator, in turn, contains a field for its own evaluator.

2.5 Some related concerns not addressed by this thesis

This thesis is concerned with manipulating programs dynamically in the form in which they run. Transformation from input text to this form is considered static, and is not addressed here. We are not concerned here with the textual representation of the program or its data—we assume it has already been transformed from its external representation into a form in which it can be handled readily by an interpreter. Techniques for this are already well-established: for example, YACC [**?**] and Lex [**?**] have been in regular use for a considerable time now, and much experience of programming them has accumulated. These and other such tools are regularly used to construct language systems for practical use, rather than on an experimental scale. Some languages allow new syntax to be established dynamically, for example by the read table and reader macros in many dialects of Lisp [**?**, section 22.1.5]. This is a form of reflection (a Lisp program can modify the behaviour of the Lisp reader) but it is not reflective interpretation. It may be taken further by allowing a language to extend its syntax in other ways [**?**] [**?**].

Another form of static manipulation of programs is *macro expansion* which is a transformation on program source [**?**]. Like syntax description, it is an important part of programming language technology, and can make use of reflection, but is outside the scope of this thesis. It can be used together with the

developments described here to provide complete language implementations on a regular framework.

2.6 Formal and informal reflection

Before reflection was developed as a technique in its own right, it appeared in small ways in many languages, either to provide specific features or to make the language more flexible. The languages that happen to provide reflection informally tend to be straightforward and of uncluttered design, with a clear model for program interpretation, a good example being *FORTH* [**?**], a simple stack-based language that allows the introduction of new operators and new syntax.

Informal reflection

In some languages there is no clear distinction between the functions that are part of the interpreter and those that are part of the application. Examples of these open languages, sometimes described as *ball of mud languages*, include *FORTH*, *POP*, and many dialects of *Lisp* [**?**]. They are noted for the ease with which other languages can be implemented using them as a base.

There are two ways in which such languages can do reflection:

- parts of the interpreter may be called directly from application programs, so that they can reflect by carrying the data of the user program into a running portion of the interpreter;
- the interpreter may call application-supplied routines in the interpreter's context through variables often known as *hooks*, accessible both to the interpreter and to the interpreted program, thus also reflecting user program data into the running of parts of the interpreter.

Both of these techniques allow application programs access to the internals of the interpreter, and so perform reflection by accessing both variables of the interpreter and variables of the application program.

These languages provide an informal form of reflection and reification through making their interpreter architectures open enough to move things in and out of them. Although typically they provide little systematic support for handling programs as data, they are flexible, and provide ready interchange between the parts of the system that are manipulated and those that manipulate other parts. These language facilities have helped to show the usefulness of reflective language features.

Formal reflection

In an explicitly reflective system, the relationship between application and interpreter is clearly defined and symmetrical. The application program can inspect

and modify the internals of the system just as the interpreter can. This access is provided by the language implementation, which includes in the language the necessary facilities:

- *reifiers*, which are functions returning, as results, values from within the interpreter
- *reflectors*, which install data values, given as arguments, into the interpreter.

This allows an application program to add features to the language or to alter the way programs are executed. A consequence is that an application program can build a new language interpreter on a plain language base, installing each new feature in turn by reflecting it into the interpreter. Used in conjunction with syntax descriptions for parser generation, this provides a way to add a language or language features to a system, as an ordinary program rather than as a special modification to the system. Thus language and interpreter design and implementation slough off their black box status and take on the mantle of conventional programming techniques.

Through these facilities, programs have access to the *computational model* of their language implementation. To make proper use of this, the language should also have operations on the data types used by the model to represent interpretation, such as the insertion of new features into an interpreter, and the composition of two interpreters to provide the special facilities of both—such as tracing from one and lazy evaluation from another.

2.7 Reflection and reification in normal program execution

Reflective actions are also useful in the ordinary functioning of programming languages. Many languages have operations which implicitly perform a limited form of reification or reflection. For example, there may be an operation for remembering a particular point in the program as it is executed, and reverting to it dynamically from anywhere within the dynamic extent of that point in the program (that is, until the procedure containing that point has been returned from). In C [**?**, section 4.6] the reifier for this is

```
place: jmpbuf = setjmp ();
```

which stores that point of execution in the variable `jmpbuf`. The point is stored as a structure containing the address in the program of `place` and the end address of the stack at that time. This reified value can then be passed around by the application, and eventually reflected back into the execution state, causing execution to continue at `place` again, by

```
longjmp(jmpbuf);
```

C provides simple handling of functions as values, by allowing pointers to functions to be stored in variables. This provides a static form of access to the program. Another reifier found in C is the `stdarg` system (or `vararg` in some versions) [**?**, section 4.8], in which successive arguments to the current procedure call, which are components of the execution state, are extracted for use in application expressions. The program can start reading its arguments using `va_start(pvar);` and then read successive arguments using `f = va_arg(pvar, type);` and finish reading them by `va_end(pvar);`. In `vararg`, the C program is using a library to handle parts of the C system, just as it might use libraries to handle, for example, graphical output.

Another example of a way in which an application program could use reflection is the dynamic installation of tracing facilities in the interpreter to show the user what the program is doing. This is done without modifying the parts of the program under inspection. Likewise, an application can change its own evaluation from applicative order to normal order without modifying the program. The system described here is capable of considerably more radical reflective operations, including doing either of the above with modifying neither the program nor the language.

All features of a programming language interpreter may be programmed reflectively. In a reflective system, the code which implements any language feature used by a program may as well be in that program as in the language implementation. Taking this a step further, we can arrange for a program to implement a language. Initially, the program must run in another language, but as it installs parts of the new language, it can begin to use those parts. (Requirements for the minimal initial language are explored, as the "Tiling Game", in [**?**].) This is a straightforward and natural way to define a language.

2.8 Kinds of reflection

As mentioned in section **??**, There are two main forms of reflection: *flat reflection*, in which the application has access to its own code and state; and *tower reflection*, in which it also has access to its interpreter's code and state, and its interpreter's interpreter's code and state, and so on. Flat reflection is sufficient for such operations as the non-local jump described in section **??**, but it does not provide the facility to inspect and modify language features.

It is possible to provide a program with some access to its own interpreter without using a tower [**?**]—that is, to just one interpreter, which runs conventionally—but this is not a regular structure and does little to model how an interpreter works. The weakness in this results from the lack of a consistent model for interpretation, that is, from its interpreter being an ad-hoc program rather than having the regular structure that is imposed by a tower of levels. In terms of our earlier analogy of the digits of a digital clock (in section **??**), it explains the hours in terms of the minutes alone, but does not make the conceptual jump of generalization necessary to explain that hours are related to minutes as minutes are to seconds, and so cannot capture the principle behind

such a clock.

Thus, flat reflective systems are suitable for implementing reflection, but not for explaining it in all its splendour.

Flat reflection—program and state as data

Reifiers and reflectors for the application program to access its own text and state are simple to write into an interpreter. They access the interpreter's own variables. For example, the current execution point (such as the program counter, in a machine-code system) of the application will typically be kept in a variable of the interpreter. The interpreter can provide a language operation, available to application programs, to place this value in the result of the operation that it interprets, and another operation to set it from a value given as an argument to the operation.

Because the action of the interpreter *causes* action in the application program, when the application changes part of the state of the interpreter, it causes a change in its own state or behaviour. Through this causality, these reflective operations provide, in this example, a simple *label* and *jump* facility. This is not unusual: what is unusual is its provision through an abstract model of how a program is interpreted. Through the reifier and reflector functions the application program can add a *jump* facility *to the language itself* that was *not* initially present in the language. This cannot be done with non-reflective languages.

2.9 Introducing the tower

The reifiers and reflectors of a simple reflective system provide means for a program to access its own text and state. This is not sufficient for work on a reflective tower. To access anything beyond the program itself, the program must also have access to its interpreter's text and state, which can be done as already shown in the diagrams of section **??**.

As with flat reflection, the data must be handled *by* the interpreter *on behalf* of the application—but how is the interpreter to handle this data? This data is part of the interpreter, not part of the program it interprets, and so the interpreter cannot provide it in the same way that it provides data on the application that it is evaluating. The explanation of how this facet of the tower works depends on the continuing repetitive nature of the tower itself, and the ability to pass data from level to level to level, as detailed below.

Tower reflection—evaluator and evaluation as data

A solution to this is to model the interpreter as the application run by another interpreter, using reification and reflection provided by the second interpreter to access the text and state of the first interpreter on its behalf. (This is shown in the diagrams in section **??**) In turn, the second interpreter needs a third interpreter to obtain access to itself, and so forth. Hence, an infinite tower of

levels of interpretation appears, in which access to each interpreter is provided by the interpreter that implements it—that is to say, the one above it.[3]

There are infinitely many *levels* in the tower, each of which runs an evaluator. This series is infinite because each evaluator must be run by another evaluator. Using this model, we conceal any mechanism outside the tower (such as a real computer), and so can use the same model to explain the whole system. We call this mechanism the *meta-evaluator* or, for reasons explained in section **??**, the *umbrella*, of the tower. [**?**] calls this the *ultimate machine*, a term which we do not use here because in our system, there is no *ultimate* machine—it is possible to go behind the meta-evaluator, as explained in section **??**, which also explains the extension of these concealment techniques to handle the necessary processing outside the tower that makes the tower work.

Continuing our use of locative terms (such as *above* in the tower), the geometry of this concealment is that such a mechanism (the real computer) is alongside rather than above the evaluator that is above the application.

However, it must also be above the evaluator, to be in a position to interpret it—where Smith describes it as the 'ultimate machine':

We resolve this by placing it both beside the tower, because tower levels only look up and down, not to the side, *and* infinitely far up the tower by inserting infinitely many levels between this real evaluator and the application.

A result of this is that however far along the tower the application looks through reification, it can never see this infinitely distant non-reifiable evaluator. Neither can it see it beside any level (because there is no means for looking in that direction). Thus, it is equally invisible in both directions, and so from the application's viewpoint these positions are equivalent. On the other hand, the evaluator beside the tower can see beside (or below–see in the diagram that they are perpendicular) itself, to the tower, and so can make the tower work.

Meta-circular evaluators - towers without stairways

Before the reflective tower was invented (or discovered), the concept of the *meta-circular evaluator* was used already to explain languages. A meta-circular evaluator is one written in the language that it interprets, and therefore capable of interpreting a copy of itself. As this was devised with no formal concept of reflection, it requires extensional operators (see section **??**) for its implementation. Reflective operators can achieve the same ends as extensional operators but in the context of a consistent framework. For example, the implementation of a conditional statement *by* the first evaluator (let us call it e_1) requires a conditional statement *in* e_1—which requires the next evaluator e_2 to implement a conditional statement, which means that e_2 requires a conditional statement

[3] By convention [**?**] we say a level is interpreted by the level *above* it.

in its text, which must be implemented in e_3 (the next evaluator) and so on. Alternatively, an extensional conditional, grounded outside this evaluator chain, may be used.

Of course, in a real implementation there must be an evaluator that provides these extensional operators. However, the real evaluator can be hidden away as it replaces *exactly* an infinitely recursive definition of the intensional one:

Here, a definition in the meta-evaluator is just one procedure, whereas the definition of the same procedure within the tower is an infinite series of procedures. *The key to reflective interpretation is the link between the infinite series within the tower and the finite definition in the meta-evaluator.*

It is possible for the finite real evaluator to replace exactly an infinite tower of evaluators because an application which terminates (that is, runs in a finite time) can only reify a finite amount of the infinite tower. Thus, the finite implementation of the tower can always instantiate enough of the infinite abstract tower to satisfy the reifying demands of any finite application. Any application which examined or processed an infinite amount of tower could not terminate anyway.

One of the keys to understanding the reflective tower model is to be found in understanding this implication of the finiteness of applications. Consideration of the abstract types involved is also very helpful in gaining this understanding. The types are described in chapter **??**, but before then we describe the overall system of which the types are part.

2.10 Central points of this thesis

This thesis introduces several new ideas, and develops further some existing ones, which are also central to this work. The main points are introduced briefly below:

- *reified evaluators*, as introduced by [**?**], are developed from an ad-hoc procedure to one parameterized by a *language* parameter, which provides the means for:
  - *mixed-language program execution*, in which procedures in different languages may call each other as though all in one language; and for
  - *language-as-data*—a step beyond Lisp's program-as-data—in which we also use...
  - *reflective tower evaluation*—also introduced by [**?**]—which is an extension of meta-circular evaluation, and allows fundamental control structures, such as `if`, to be defined by procedures at the user program level; an extension to this is...
  - *meta-tower evaluation*, which brings the tower implementation—previously backstage—inside the tower, and does for tower evaluation

what tower evaluation does for evaluation, allowing the tower implementation to be defined at the user program level;

- re-use of the idea of meta-tower evaluation shows that it does the same for meta-tower evaluation as it does for tower evaluation and for evaluation, and so may be seen as a fixed-point from which to construct any of these forms of evaluator.

- *shadowing*, a technique necessary in all reflective interpretation systems, by which a procedure outside the tower stands in for an infinite series of procedures within the tower, is treated more systematically than before, and it is made reifiable, and
  - the similarity between (reified) shadowing and (reified) languages is made apparent, leading to the development of...
  - a pair of very simple procedures, for use as the basis of an evaluator and of a meta-tower evaluator respectively.

Using these ideas, an implementation of a mixed-language meta-tower, called Platypus, is constructed, and shows itself to be passably efficient compared with conventional evaluator systems.

2.11 Summary of introduction

Reflective techniques are based on two facilities: *reification*, by which a program may examine its own code, state and interpreter; and *reflection*, by which it may modify any of these. The connection that reflection makes between a programs actions and its behaviour is causal in nature: modifications that a program makes to its interpreter may cause changes in the way that the program is interpreted.

Some programming languages, not normally regarded as reflective, provide a limited range of reflective operations, such as access to the parameter list of a procedure call. However, in a fully reflective program interpretation system, all the features of any programming language can be implemented through reflective programming in the program, thus removing the distinguished status from the interpreter of a language, and making it equivalent to any other program in the system.

There are two kinds of reflection: *simple reflection* and *tower reflection*. Simple reflection provides a program with access to its own code and state and interpreter. Tower reflection also provides it with access to its means of interpretation—that is, the mechanism by which a program is related to its interpreter, and thence to its interpreter's interpreter, and so forth. Tower reflection is more general, and, not having an arbitrary stop after the first level, is a more regular conceptual structure. Thus it is a more powerful tool for reasoning about intensional reflection and about interpretation.

Although tower reflection deals with infinite structures, it is possible to implement it with finite constructions. This thesis explores the infinite towers and

their finite implementations, and investigates whether an interpretive programming system built this way can be made reasonably efficient, compared with conventional, non-reflective interpreters.

In this thesis, we develop a reflective tower implementation, called Platypus, and use it to demonstrate many of the points discussed.

"Oh, what fun it'll be, when they see me through the glass in here, and ca'n't get at me."

Alice, Looking Glass, Chapter 1

Chapter 3

The Closure—a building block for computation

Adding reflection and reification to a meta-circular evaluator can make it a more useful tool (or model) for describing the language it implements, but does not add to our ability to handle programming languages in general. We need to make the language an explicit parameter of each level of evaluation, and provide each level with access to its language. Consequently:

- successive levels of the tower can be implemented in different languages, and each tower level must store which language it is in, *and*
- there must be types and operations concerning all the parts of a language implementation, and that these must be the same for the components of all languages, so that we have a single way of describing any level of the tower.

In making the language an explicit part of each level, and thus in letting it vary from level to level, we must ensure that there are compatible data representations throughout the system, so that values may be passed freely between languages and between levels. The requirement is not that all levels must have the same representation system, but that adjacent levels must be able to translate information from one representation to another so that they can transfer data between them. This is normally met by the reifiers and reflectors translating the data between the different representations, as discussed further in section **??**.

The availability of data produced by reifying levels, and the consequent possibility of levels passing levels as data to each other, requires that the representation of levels also be compatible at all levels.

3.1 Representing computation as data

In order to handle parts of a computing system as data, we need a data type for representing parts of a computation. The computation is something active, and its activities may be prescribed procedurally. However, such procedural descriptions are static, and do not represent the actual computation. Computation must be represented with a combination of its (procedural) prescription and a snapshot of its state at some particular time—which part of the procedural prescription it was executing, and on what data. It is important that the representation used:

- is at the right level of granularity, that is, represents the right amount of a computation in each unit
- holds its piece of computation in a form appropriate both for manipulation and for execution

Closures

Since the components of program texts are *procedures* (see the note on procedures and functions in section **??**), and the components of program states are the states of individual procedures being evaluated, we will use a type combining them, the *closure* of a procedure (see section **??**) as our basic type for describing tower objects.

A closure *closes over* the code and state of a procedure, making procedures into self-contained values that can be passed around complete with the state that they need. For convenience, we split the state into two parts, local and non-local, so the closures used here have three components representing the state of a procedure:

- the `expression` of the closure, which represents the `text` (or `body`) of a procedure
- the `value-list`, which contains the arguments when the procedure is called, private internal state while it is being evaluated, and the results as they are prepared to be returned to the caller;
- the `environment`, which holds all non-local variables accessible to the closure, bound in other closures (called *lexical* or *free* variables).

The expression of a closure we represent as a '*parse tree*' of the procedure text it represents. Each *node* of the tree has as its first sub-expression a node type (which we call its *operator*) and possibly some other *sub-expressions*, which are taken as arguments to the operator. For example, `a+b` has the operator `+` and the sub-expressions `a` and `b`. This representation is used for all programming languages in this system. Although the system we build does not make this form of expression a strict requirement, in practice we have found no need for other representations.

In fact there are two parts to the expression of a closure: the *procedure expression* and the *continuation expression*. The procedure expression is the whole body of the procedure, and the continuation expression is the sub-tree of the procedure that is currently being executed. (At any point, a frozen state of computation may be continued from its continuation.)

The reason for this is that we need to know at all times both the current point of computation, and the whole definition of the procedure. As the continuation expression moves inward down the expression tree, we need the procedure expression for reifiers to find what the original procedure was. If we were to have only one expression in each closure, it would not be possible to return to the evaluation of an outer expression on finishing evaluating an inner one, except by storing the information in the next level above in the tower—which, although convenient for interpretation (because it is part of the state of that interpreter), is inappropriate for reification, because that is not the level to which belongs. For example, when in evaluating `(lambda (a b c) (* (+ b c) a))` the evaluator is at `(+ b c)`, the `(* ?  a)` must be stored ready to be resumed when `(+ b c)` has been evaluated. It would also make it impossible to make procedure calls (or even expression evaluations) that do not start new instances of the interpreter's context.

This part of the model of execution is analogous to that of compiled code in many conventional systems. Each call level has a stack frame, in which are (or may be) stored pointers to the local variables used, the environment active at that point, a pointer to the start of the procedure, and a current execution pointer for the procedure (the return address for the unreturned call it is making).

The interpreter uses the operator of the continuation expression to choose what to do at each step of interpreting a closure. For example, in `a+b`, it might evaluate each of `a` and `b`, sum their results, and return that value to the surrounding expression. The operator is all the information the interpreter needs to decide how to select what to do at that node.

The classification of variables into two kinds formalizes the common practice in language implementation, while also providing purely positional parameter passing—that is, without needing common knowledge of variable or parameter names between caller and callee—which is necessary for referential transparency (see section **??**).

The value list corresponds to the stack frame or activation record of common practice, and shared between call levels within an evaluation level, and the environment is equivalent to the display of languages such as Pascal.

The value list as we use it takes the form of an *open stack*. Each procedure may add or remove things at the end of it, and access items by indexing into the stack from the end. Before calling another procedure, a procedure making a call pushes the call arguments onto the end of the value list. The callee will then be able to use them as its arguments, and also use the same locations and any that it may add beyond them for its own local state (workspace). Before

returning, it will pop its workspace and arguments, and push its results, if any, where the arguments and workspace were.

This form of stack is compatible between languages with an explicitly open stack, such as FORTH and PostScript, and those with closed stack frames, such as Lisp and Algol. With such a stack, cross-calling between these kinds of languages is automatically available.

As well as being used as a Lisp-style environment or a Pascal-style display, the environment may be used to hold unification variables for languages such as Prolog. This allows variable bindings, made by other languages, to appear as instantiations (unification bindings, or substitutions), and allows unification bindings to appear as ordinary variables when a procedure in a unification language calls one written in a non-unification language.

Here is a Lisp definition of a closure, which will be used in example code later:

```
(defstruct closure
   ;; a function to interpret the closure:
   (interpreter (type closure))
   ;; a Lisp s-expression:
   procedure-expression
   continuation-expression
   ;; arguments, workspace, result:
   (values (type vector))
   ;; an alist or hash-table: , perhaps?
  (environment (type lookup)))
```

The nature of the *interpreter* field of the closure is described in section **??**. Here, we just show it as a closure, but in fact it contains several other components.

3.2 Including the language in the closure

The introduction of mixed languages to a reflective architecture, and with it the idea of programming languages as values, adds a new requirement to the closure operation. As well as the procedure texts and states, closures must close over the language in which that text is to be interpreted. Commonly, a program text is written on the assumption that it is to be interpreted as a program in just one specific language, and so it is generally reasonable to say that a program is in (or for) a particular language. However, the language is just part of the context—it is the semantic, or linguistic, context—within which the program is be evaluated, and so could vary, just as the value context provided by the lexical and dynamic environments may vary. For most purposes, though, it is appropriate to close the text and the language together, so that wherever the text is seen, the associated language is made available with it. Therefore, we may say “the language in which an expression is written”.

There are several places at which the language could be specified in representing programming systems. Attaching a language to each procedure (as part of the closure of that procedure) allows procedures at the same level of interpretation to be written in different languages, which is what is required for normal mixed-language programming as mentioned in section 1.6.

Two effects of closing the language into the procedure representation are that:

- closures are independent of external language information, and so are of the same form for all languages. As a consequence of this, we can make procedure calls between any two languages as easily as we can within one language.
- the closure indicates in which language its expression is written. This information is needed for interpreting programs in a mixture of languages.

Representing languages as data values

To evaluate a closure, we use the language interpreter which is a component of the closure. This interpreter is itself a procedure (stored as another closure), and takes the other parts of the interpreted closure as its arguments:

```
(defun eval-closure-0 (closure)
 (funcall                                ; to evaluate a closure,
  (closure-interpreter closure)          ; call its interpreter
  (closure-procedure-expression closure) ; with the expression,
  (closure-continuation-expression closure) ; values, and
  (closure-values closure)                  ; environment
  (closure-environment closure)))           ; as arguments
```

but, for orthogonality and for convenience in reification and reflection, we make it take one argument, namely the closure in which it occurs:

```
(defun eval-closure-1 (closure)
  (funcall                        ; to interpret a closure, call
    (closure-interpreter closure) ; its interpreter, with the
   closure))                      ; closure as argument
```

Since the interpreter of a closure is itself a procedure (represented as a closure) this evaluation goes infinitely high up the tower, and we see a tower of active closures forming, rather like a tower of meta-circular interpreters.

3.3 The relationships between levels

There is a simple set of rules describing how adjacent levels of a tower are connected, and how the rules for adjacent levels can be extended transitively to

longer strings of levels.

Unlike a meta-circular interpreter i_a that must always be interpreted by an interpreter i_b such that (using the notation introduced in section **??**)

$$(i_a \rightarrow i_b) \wedge (i_a \equiv i_b) \tag{3.1}$$

our neighbouring interpreters are allowed to be different:

$$i_a \neq i_b \tag{3.2}$$

while still requiring

$$i_a \rightarrow i_b \tag{3.3}$$

Each interpreter also provides means to pass information between its own closure and the one it interprets, ($i_a \blacktriangleleft i_b$ as well as $i_b \blacktriangleleft i_a$—the latter is always the case), so we have a reflective tower, where each closure is provided (by its interpreter) with access to itself:

$$\begin{aligned} &(i_a \blacktriangleleft i_b) \wedge \\ ((i_x \blacktriangleleft i_y) \wedge (i_y \blacktriangleleft i_z) &\Rightarrow (i_x \blacktriangleleft i_z)) \\ &\Rightarrow (i_b \blacktriangleleft i_a) \end{aligned} \tag{3.4}$$

and (*by* its interpreter's interpreter and *through* its own interpreter) to its interpreter's closure

$$\begin{aligned} &(i_b \blacktriangleleft i_a) \wedge \\ &(i_c \blacktriangleleft i_b) \wedge \\ ((i_x \blacktriangleleft i_y) \wedge (i_y \blacktriangleleft i_z) &\Rightarrow (i_x \blacktriangleleft i_z)) \wedge \\ &(i_b \blacktriangleleft i_c) \Rightarrow (i_a \blacktriangleleft i_b) \end{aligned} \tag{3.5}$$

Thus, the ability of a closure to receive access to itself depends on the closure's interpreter; and, in turn, that interpreter can only have access to *its*elf if *its* interpreter provides reification.

3.4 Procedure calling using closures

Before looking at how this tower is held together, we must understand how a procedure is called using these closures. There are two ways of doing this:

- the same instance of the interpreter can itself handle the interpretation of the called procedure, (a *non-reflective call*)
- or the interpreter can recurse to make and call another instance of the interpreter to interpret the called procedure (a *reflective call*).

In the first form of procedure call, to interpret a call of a procedure directly, the interpreter must do the following actions:

- Take a copy of the closure of the procedure
- extend its `values` with the arguments to the call.
- Save on the stack the continuation of the calling procedure—which was the currently active closure in its level
- Continue interpreting, with the copy of the called procedure as the current closure being run by the interpreter. This closure has had its value list filled in, thus supplying it with its arguments.

In the second form of procedure call, the interpreter may itself recurse to make the new stack frame, thereby making *its* interpreter hold the saved continuations, which it must do in the manner described above—or, in turn, by making the next interpreter hold the previous continuations. . .—and that may in turn make another interpreter hold the previous continuations, and so forth. The second method is particularly appropriate for interpreting a language where all calls are reflective, as the whole closure is always the object passed around.

However, this makes it impossible to make non-reflective calls, which is unfortunate, because reflection into a interpretation of a level no longer has the desired effect; instead, we would just reflect into the interpretation of a procedure, and the effects of the reflection would be lost when the procedure returns. So, we choose the first method for normal procedure calls. These two forms of procedure call are covered in more detail in section **??**.

Interpretation of a procedure works by recursive descent of the procedure's expression tree. At each node of the tree, the node's *operator* directs the interpreter how to handle that node.

For example, an operators for flow control, such as `if`, may call the interpreter explicitly to implement that flow control, using its own flow control to decide when to call the interpreter: so to interpret

```
if a then b else c
```

we can use

```
(defun if (closure)
   (let ((expr (closure-continuation-expression closure)))
     (if (interpret (first expr) closure)
        (interpret (second expr) closure)
       (interpret (third expr) closure))))
```

An operator which does no flow control, which we sometimes refer to as a Lisp-like operator, as it is similar to a Lisp procedure in the way that its argument sub-expressions may be evaluated, may evaluate its sub-expressions in an arbitrary order. An example is:

```
(+ a b c d ...)
```

The simple applicative order version shown here evaluates them from left to right, using the `reduce` procedure of Lisp [**?**, Section 14.2], which applies a given procedure taking two arguments to each element of a sequence and an accumulating result:

```
(defun add (closure)
   (reduce #'+
      (map 'vector
        #'(lambda (expr)
             (interpret expr closure))
          (tail (closure-continuation-expression closure)))))
```

The definition of each operator is a closure a procedure which, when applied to a closure, interprets it in the appropriate way. Operators are done this way to facilitate language definitions and mixed language working, as explained in section **??**.

Active and inactive closures

The call stack of each level is made of a sequence of closures each representing an *active instance* of a procedure. A program is stored as a collection of *inactive* closures waiting to be copied. When a procedure is called, its inactive closure is copied onto the call stack to make an active one, much as the calling mechanism of a conventional interpreter builds a stack frame for the procedure being called. Pointers are stored in the stack frame to point to the previous stack frame, to the procedure body (code), and to the static or constant data needed by the procedure; the variables provided by the procedure will also be linked into the environment.

Whenever a procedure is called non-reflectively, its closure is instantiated, the value list of the new copy filled in with the call arguments, and the closure placed on the stack of the bottom level of the tower, where it becomes the current closure. This creates a new activation record within the bottom level of the tower, but does not add a new level to the tower.

When a procedure is called reflectively, a copy of the interpreter of that procedure's closure is taken, with the closure as the interpreter's argument. The copy of the interpreter is installed just above the bottom of the tower, where it runs a new level of the tower.

Since in either form of call, the closure is placed in the tower, the tower always provides the context for evaluating the closure.

3.5 Summary of closures

At each level in our infinite tower, there are one or more procedures, of which at any one time one will be active, and some (possibly none) will be on a list of saved procedure evaluations to be returned to.

In common with many other language systems, we represent procedures by *closures*, each containing the code of the procedure and any context that must be carried with that code to interpret it.

To make explicit the interpreter of a procedure, we *close over* the interpreter when constructing the closure of the procedure, and thence the rest of the tower of which that evaluator is the lower end, thus making it contain the whole of the context in which the procedure is interpreted.

A closure constructed this way we call an *interpretive closure*, since it encloses all the information needed for interpreting a procedure.

We use interpretive closures as the building block for constructing reflective interpretive towers. Each level of a tower contains a closure (actually, a stack (or list) of closures). The interpreter of an interpretive closure is also an interpretive closure, as are the operator definitions. Closures are also used to represent procedures available for calling. When a procedure is called, its closure is instantiated by copying in onto the top of the stack, and filling in fields that come from other parts of the level and the tower (such as the dynamic environment within which it was called.)

> And the whole earth was of one language, and of one speech. And it came to pass, as they journeyed from the east, that they found a plain in the land of Shinar; and they dwelt there. And they said one to another, Go to, let us make brick, and burn them throughly. And they had brick for stone, and slime had they for mortar. And they said, Go to, let us build us a city and a tower, whose top may reach unto heaven; and let us make us a name, lest we be scattered abroad upon the face of the whole earth.
>
> And the Lord came down to see the city and the tower, which the children of men builded. And the Lord said, Behold, the people is one, and they have all one language; and this they begin to do: and now nothing will be restrained from them, which they have imagined to do. Go to, let us go down, and there confound their language, that they may not understand one another's speech. So the Lord scattered them abroad from thence upon the face of all the earth: and they left off to build the city. Therefore is the name of it called Babel; because the Lord did there confound the language of all the earth; and from thence did the Lord scatter them abroad upon the face of all the earth.
>
> *Genesis 11:1-9*

Chapter 4

How the tower levels are linked together.

The relationships between adjacent levels of a tower are as important as the contents of each level. Having examined the closures that are used to make up each tower level, we now look at how they fit together to make a tower, and how that infinite tower is linked to a meta-evaluator so that it may be interpreted in finite time.

4.1 Levels, strings of levels, and towers

A tower is made up of a sequence of levels, but not every sequence of levels is a tower. We will call a sequence of levels a *string* of levels, and here explain some of the properties of these strings.

A tower is a string of levels that has an application (bottom end) that is not an interpreter—that is, does not take another program as its subject and evaluate the result of that program. A grounded interpreter (one whose computation is known to take a finite number of steps, as explained on page **??**) at the top is not necessary for it to be a tower.

A string is at the bottom of the tower if its lower end interprets an application. Since this application may be another interpreter, the same string could be in a longer tower, not at the bottom of it. The lower limit of the tower is defined by what we consider to be an interpreter: at a more abstract level than computer science, we may in some cases consider the application program to be the interpreter of something in the world outside the computer; indeed, one possible view is that the observer of the system may be seen as a tower of meaning, in a morphic mirror image of the reflective tower. (This is one of the possibilities available through co-tower relations, as described in section **??**.)

A string interprets the closure applied to its lower end. In turn a string must be interpreted by a closure to which its upper end is applied. Thus, a string of interpreter levels is equivalent to an interpreter level: it interprets something, and is itself interpreted by something. The interpreters in the string are *functionally composed*, making what is intensionally a string of interpreters into what is extensionally a single interpreter.

We say that a string (or a level) is *grounded* if it interprets a terminating application applied to its lower end in finite time. In the formulae below, levels which are known to be grounded are written suffixed thus:

$$L_{Gi}$$

where G indicates that L is grounded. i indicates which level L is in its string, with level 0 being the application, and level numbering increasing toward the top of the string, such that L_{G1} is the interpreter of L_{G0} and L_{G2} is the interpreter of L_{G1}.

4.2 Links in each direction

Reification and reflection are complementary operations, reflection moving information up the tower from L_{G1} to L_{G2} into the level interpreting the level providing the data, and reification moving data down the tower, from L_{G1} to L_{G0}, from interpreter to interpreted. This implies that both interpreter and interpreted can access each other. Formulae (given in section **??**) explain the relationship between levels in terms of this knowledge of neighbouring levels. To use these paths to move up and down the tower, links must be present in both directions:

- *The link for reflection* The link through which reflection occurs is that from a level to its interpreter, that is, to the closure interpreting this level's closure. This link is part of the structure of each closure, whether or not it represents an interpreter. The link is established for each active instance of each function as the function's closure is copied onto the stack for execution.

- *The link for reification* The link through which reification occurs is established through the argument it is given when it is called. When the interpreter is called, its closure is instantiated on the tower, and the closure that it is to interpret is written into the first argument position of the interpreter's closure. Thus, this link is set up implicitly by the reflective calling mechanism. It is present only in closures that represent interpreters.

  This link is in a fixed place in any closure that is an interpreter: it is the first argument of the closure. (However, not every closure with a level as its first argument is an interpreter!)

These two links form a protocol that is observed throughout the tower. This is important not only dynamically, in the execution of a tower, but also statically or structurally, for inspecting and modifying the tower, allowing any level to perform the same operations on all the levels to which it has access.

Links always occurring together

Since both links are set up by parts of the calling mechanism, they always occur in pairs:

- every closure must have an interpreter to be interpretable;
- each interpreter in the tower must have something to interpret.

The application level is the only level that has no level below it for it to interpret; whilst the tower stretches up infinitely from the application level.

These links make a two-way chain throughout the tower, so any interpreter in the tower can access the application, although normally only the last interpreter should do this, and the application can access any interpreter in the tower, although it will usually only need to access the nearer ones, and (provided it terminates) will never access those infinitely far from it, as explained in section **??**.

Rolling up the infinite string for storage

Although the tower is infinite, it may be represented finitely, because its infinitely long upper initial string consists of identical successive levels (or identical strings of levels). In Smith's work, such levels are termed *boring* [**?**]. This repetitive string may be stored as a circular structure, in which those levels are not only equivalent but actually the very same level. A hidden mechanism, to be explained in section **??**, separates off copies of this level (or these levels) when they have to be changed, rather like printing off copies of a pattern from an inked roller:

As the levels produced this way are copies of the original, any of them can be changed without affecting either each other or the original.

4.3 The umbrella of the tower.

Since a tower is an infinite structure, any computation involving all levels of it cannot terminate. In particular, as each level is interpreted by the level above it, we can never find a level that can be executed non-reflectively. So, from a traditional computability viewpoint, a program represented as the base of an infinite tower could not be executed were it not for that we can rely on the application terminating. If it terminates, it can have reified only a finite amount

of the tower. If it does not terminate, we cannot tell whether it requires infinite amounts of reified tower, and so it only matters that we can realize (that is, convert into a form that really occupies storage, as described in section **??**) an arbitrary but finite number of levels.

A special interpreter can be constructed using this fact. It takes the place of an infinite amount of the tower (the boring part), and is always just above the highest level that has been touched by reification or reflection. Because it is always at the top of the tower, we call this interpreter an *Umbrella*. Umbrella interpreters are explained further in section **??**.

4.4 Tower computability; propagation; integrity

This section shows how a tower that is apparently infinite can be evaluated in a finite number of steps. We look at the complexity of each action at different levels that it involves, and then procede to remove factors from the complexity by introducing interpreters that absorb (or collapse) level shifts. Since adding a level of interpretation multiplies the complexity of an action, absorbing a level removes the multiplying factor that that level introduced.

From this, we see that the ability to absorb an infinite number of unwanted multiplying factors (by collapsing the levels at the top of the tower) allows us to prevent the infinite prolongation of calculation caused by tower interpretation.

Relative computability

We have shown that a terminating application in an infinite tower in one sense can be computed, because the infinite part of the computation must be collapsable into a finite computation for any terminating application, and in another sense cannot terminate, because the computation happens at an infinite number of levels. We wish to resolve this paradoxical situation, and so we develop a theory of computability for interpretive towers.

The conventional idea of computability, in the sense of what can be computed by a universal machine such as a Turing Machine, is not sufficient to describe this, as it works only in terms of absolute computability. We must use *relative computability* to describe what the lower end of a string of levels can interpret if its upper end is given an interpretation. In terms of Turing's work [**?**], this is called *oracular* computability, because something can be computed if the information on which that computation depends is given by some oracle. In a reflective system this information includes the computation process itself as a data value.

Using relative computability, we can describe what effects a string of levels may have on computability when it is inserted between an interpreter and a procedure that is interpreted.

Relative computability—quantitatively

What rules for computability hold in a infinite interpretive tower? Let us take an interpreter L_{Gx} which if used to interpret a terminating application a_x thus:

$$L_{Gx}(a_x(input_x)) \tag{4.1}$$

will calculate it in a finite number of steps of a grounded interpreter L_{Gx}. L_{Gx} is itself represented as a closure, and so tells us what interpreter to use to interpret it, which we will call$L_{G(x+1)}$. This lets us use it as the application of another interpreter in a meta-circular interpretation chain:

$$L_{G(x+1)}(L_{Gx}(a_x(input))) \tag{4.2}$$

where $L_{Gx} \equiv L_{G(x+1)}$ This is now also calculable in a finite number of steps, since, if, z terminates, $L_{Gx}(z)$ terminates.

But since $L_{G(x+1)}$ is a closure, it must be interpreted, by some grounded interpreter (either meta-circularly or by some other one):

$$L_{Gy}(L_{G(x+1)}(L_{Gx}(a_x(input)))) \tag{4.3}$$

where $L_{Gy} \stackrel{?}{=} L_{Gx}$ (where $a \stackrel{?}{=} b$ reads as "a may or may not equal b") and this continues infinitely, making our terminating program a_x terminate at each of a finite number of levels, but take infinitely many steps of an interpreter which is infinitely far up the tower.

Can we force this computation to become absolutely finite? Using the notation introduced in section **??**, we can calculate how many steps are involved in working a calculation through two levels of interpretation:

$$L_{Gy} \stackrel{a}{\rightarrow} L_{Gx} \stackrel{b}{\rightarrow} a_x \Rightarrow L_{Gy} \stackrel{ab}{\rightarrow} a_x \tag{4.4}$$

If we have an interpreter L_{Gz} capable of interpreting two tower levels at once, we can do

$$L_{Gz} \stackrel{e}{\rightarrow} a_y \tag{4.5}$$

and also

$$L_{Gz} \stackrel{f}{\rightarrow} a_z \tag{4.6}$$

Now let the application a_y in the first of the pair of formulae above be an interpreter L_{Gv} which interprets the second application a_z in g steps:

$$L_{Gv} \stackrel{g}{\rightarrow} a_z \tag{4.7}$$

Substituting a_y for L_{Gv}:

$$a_y \stackrel{g}{\rightarrow} a_z \tag{4.8}$$

and showing both stages of the calculation of the number of steps

$$L_{Gz} \stackrel{e}{\rightarrow} a_y \stackrel{g}{\rightarrow} a_z \tag{4.9}$$

shows us that L_{Gz} can interpret a_z in eg steps

$$L_{Gz} \xrightarrow{eg} a_z \tag{4.10}$$

However, from formula **??** above, we know that $L_{Gz} \xrightarrow{f} a_z$, and thence $f = eg$. Thus, through the use of an interpreter that can take on two levels of interpretation at once, we have removed the multiplication of the number of steps from the calculation relating steps at one level to steps at the level above it. L_{Gz} does this.

In L_{Gz} we have a kind of closure which, through running two levels at once, absorbs and generates level shifts within itself. Since it is the level shifting that prolongs a terminating computation at one level into a non-terminating computation at an infinite number of levels, we can use this interpreter that performs level shifts within itself in finite time to provide full grounding for an infinite tower.

This makes it possible to evaluate a tower that has at the top an interpreter that can handle level shifts within itself, while still providing the illusion of an infinite number of levels of interpretation, as described in section **??**.

We call this kind of interpreter an *umbrella* since it appears to occur at the top of the tower, and caps the tower.

Shadowing interpreters

We cannot take this solution as it stands because each level can have only one closure running in it at once—it contains a list of closures, as a call stack—and each closure can represent the state of just one level. (The way that several closures are running at once is at separate levels of interpretation, as explained in the clock analogy of section **??**) Because of this relationship between closures and the representation of tower levels, this shift-absorbing closure must, by the rules, be *outside* the tower, *shadowing* an infinite string of ordinary levels that are within the tower. This is as explained in section **??**:

A closure in the tower is *shadowed* by a closure outside the tower if the one outside the tower implements the one inside the tower and all those above it in the tower that take part in its interpretation. The closure outside the tower is called the *shadow* of the corresponding closure within the tower. As the meta-evaluator of a tower begins the evaluation of each closure, it first checks whether the closure is in a table of shadowed closures—the *shadow map*. If the closure does have a shadow listed in the shadow map, the meta-evaluator calls the shadow directly, that is, as a procedure within the meta-evaluator. If there is no shadow, the closure is interpreted stepwise by the meta-evaluator—that is, as a regular interpreter does.

The idea of shadowing is central to this thesis. In later chapters, we develop the details of a mechanism for shadowing the upper initial string of a tower,

which makes it possible to implement the interpretation of an infinite tower in finite time.

The *shadowing interpreter* of a tower, in performing some number of steps, makes each of the levels that it interprets—that is, those below the one that it shadows—take some smaller number of steps in each of their actions. In this way, it is like the infinite string of levels which it shadows, since their progression through their procedures would also cause the same effect on the lower parts of the tower.

Let us return to the digital clock analogy of section **??**, in which we examined how the minutes digits affect the hours digits, and so forth. Starting at the seconds digit, and setting off in the other direction, we can posit the existence of further digits, which are usually invisible: a deciseconds digit, a centiseconds digit, and so on. In a typical clock, some of these digits will exist, but there will not be infinitely many of them. While our model of how a clock works may continue to produce more and more digit devices, each counting faster than the last, in practice we eventually reach some device that ticks without the aid of a previous ticker—or rather, one in which the ticks are of a very different kind, such as the resonance of a quartz crystal. The shadowing interpreter of the tower takes the place of the crystal in the clock. Although there could be more counting devices, a non-counting ticking device (with the same interface at its slow end as a counting one) rounds off the chain of counters, and shadows an infinite number of even faster counters.

It is worth noting that the shadowing interpreter has not taken control away from the string of interpreters that it shadows. It exists beside that non-existent string, just as the quartz crystal in the clock exists beside the non-existent extra counters. It does not displace them! They are still there as a concept.

From the abstract, extensional point of view, the last counter sees something equivalent to a string of counters. From an implementational, intensional viewpoint, the crystal is there. They are views in two worlds of the one thing performing its function.

The extensional view is what you see if you look at the counter chain from any one of the counters. If you look at it from the quartz crystal, you are viewing it from the intensional viewpoint.

Since reflection is a means for abstraction, it is natural to view it reflectively, using its own abstraction. This always involves taking the extensional view of the system—looking upwards in a tower, or outwards in a meta-tower. This is not useful for creating and starting a (meta-)tower; the tower's creator must work from the outside in (like Smith's lightning bolt), as the tower cannot start itself from nothing. This necessity for an outsider follows the same logic as that of section **??**. From the outsider's point of view, the system within the (meta-)tower is flat; it is all processed at one level, as the application of the outsider; in the clock analogy above, the crystal puts its output into what may be seen as a single counting device; whether that counting device is infinite or finite is irrelevant.

Rules for groundedness

In the discussion above, we have assumed that we work with grounded towers only—as indeed we must for the practical evaluation of a tower. However, the same rules hold for non-grounded towers, which cannot be evaluated, but can still be manipulated in the same way. A non-grounded tower is ungrounded because it has no grounded interpreter anywhere in it. A non-grounded tower can be made into a grounded tower by connecting to it a grounded tower, which will interpret it in a finite number of steps. (The connection will be at the bottom of the boring part of the tower.) These grounded towers ground the towers to which they are appended by acting as oracles [**?**] for the interpretation of their highest levels.

Integrity of strings

If an interpreter level evaluates its application finitely for a finite application, then we say the interpreter has *integrity*. A finite string of adjacent levels also can have integrity (as a relative property; this is integrity with respect to an oracle), if its highest level of interpreter terminates evaluation for a terminating application run at the lowest level. These are both the same concept of integrity. As explained above, an interpreter which evaluates more than one level is an exact substitute for a string of interpreters (an infinite string, if the upper end is boring).

Since an infinite tower can be evaluated by an interpreter that absorbs level shifts, and is thus, effectively, an extensible finite tower, it may also have integrity, but only if its higher end is accessed via its umbrella interpreter. This is because it is the umbrella that provides the computability. If we look at the infinite upper string of interpreters that the umbrella stands in for, instead of looking at the umbrella, the tower then appears not to be grounded.

The integrity of a string of levels is broken by it having any level without integrity. A level is grounded[1] if it is linked with integrity to a level that is grounded, such as an umbrella, or a string that is known to be possible to link to an umbrella.

Unlike conventional semantic definition systems, reflective interpretation allows us to reason about non-computable programs, and constructs which cannot be computed, just as easily as computable ones. This may not seem very useful in its own right, except for pure theoretical linguistics, but since computability is undecidable for most programs (and so all programs must be treated as potentially non-computable) this is not necessarily a problem, and so this approach is very generally applicable.

The solution to the problem of grounding towers is mentioned at the start of this chapter: at some level on our way up the tower, we find a meta-circular

[1] N.B. the ground is at the top, and so can be used to grow computer science trees which have their roots at the top.

interpreter (let us say L_n) whose interpreter, L_{n+1}, is the same as the interpreter L_n itself, that is, $L_n \equiv L_{n+1}$. Since this level is a *meta-circular* interpreter, this relationship holds for the next level too, and for all subsequent pairs of levels L_m, L_{m+1}. Closing this transitively, we have

$$\forall_{p,\,q\,:\,p\geq n, q\geq n}.L_p \equiv L_q \tag{4.11}$$

that is, once we have reached the level L_n, all levels thereabove are the same.

Since we wish to use reflection in real computations, for practical purposes we can make the tower above that level into a circular structure, such that not only is

$$L_n \equiv L_{n+1}$$

but also

$$L_n = L_{n+1}$$

In terms of data structures, the interpreter field of that closure points to the closure itself. We call L_n the *standard evaluator*.

Shift-absorbing levels

The real interpreter outside the tower, called the *meta-evaluator*, mimics this closure L_n, and all the closures past it in the tower. This imposes a constraint on towers in the system: any which are to be evaluated must have L_n as their upper initial string, which, given that the meta-evaluator mimics them, implies that they are grounded. A tower which has a repeating chain of anything other than L_n as its initial string cannot be evaluated by the meta-evaluator, which keeps on realizing new levels until it gets to one running L_n.

The operators that we provide for reflection help to maintain this, in that by default all new closures are created with L_n as their interpreter. It is possible to enforce this rule absolutely, by insisting that any closure that becomes the interpreter of another closure must be grounded through the standard interpreter. This can always be determined, since the route up from a closure can be determined statically from its contents.

The standard evaluator and the meta-evaluator

Since the standard evaluator is an invariant part of the system, mimicked by code outside the tower, it must not be mutable by reflection operations. The meta-evaluator employs a technique for maintaining this integrity while still allowing reflection at any level of the tower, without requiring infinite storage or time. This technique is described in detail in section **??**.

If the reflection system tries to access or modify L_m (where L_n is the lowest level mimicked by the meta-evaluator, and $m > n$— that is, any level representing the standard interpreter), new levels, all running copies of the standard evaluator, are created in memory for levels $L_n \ldots L_m$. The 'evaluator' link of L_{n-1} points to the new L_n, and the 'evaluator' link of L_m points to the standard evaluator. Then L_m is modified as specified by the call to the reflection

operator, and the meta-evaluator continues execution as before, but now shadowing levels $L_{m+1} \ldots L_\infty$ instead of $L_{n+1} \ldots L_\infty$. Note that level L_n has not been changed by any of this activity.

This activity has made into real levels some levels that previously were not stored anywhere, although they were accessible for inspection. We say it has *realized* them.

This realization is part of the process of level-shifting, that is, the transfer of information or control between levels of the tower. It takes place only as needed, so not all level shifts will realize any levels; they may simply move between levels that have already been realized. The realization is performed by the meta-evaluator, which is explained in detail in section **??**.

4.5 Reifying the meta-evaluator

The meta-evaluator implements reflection through non-reflective means, and so can be an ordinary program, written in a language without explicit reflective features (such as C, or Common Lisp). But, since reflective interpreters provide a superset of the facilities of other program evaluators, the meta-evaluator could itself be the application run by another reflective tower. To extend the reflective facilities provided by a tower, we can provide a connection between a tower and the tower of its meta-evaluator.

In doing so, we make the whole system more regular, and we bring reflective techniques, and meta-circular interpretation—already accepted as a standard and powerful means of describing interpretation in the Lisp world—to bear on the problems of how the meta-evaluator is connected to the tower. We later find a similarity between this connection and the connection between operator names and operator definitions in a language.

The dimensions of the tower

When we provide a means of reaching this second tower (for reifying and reflecting it) from the first one, we have a new form of reflective system, which has a two-dimensional tower rather than the one-dimensional tower already described. The second dimension, like the first, may continue either for one level, as a non-towering second dimension:

However, since the point of a tower-reflective system is to make the interpretation mechanism visible and manipulable, such a meta-evaluator stands out as an oddity—although it is a running procedure, it is not decomposable in the same terms as are the other running procedures in the system.

A more consistent model for reifying interpreters demands that we include the meta-evaluator within the reifiable part of the system, *on the same basis*

as the other reifiable parts of the system. This can be done by providing a type `tower`, and including in each `level` a field pointing to the `tower` of which that level is part. (The definitions of the `level` and `tower` types are given in section **??**.)

It would be possible to make the meta-evaluator, which is attached to (or part of) this tower structure, be simply a procedure in the language of the substrate system. However, for consistency, to allow the same reasoning to be applied to the meta-evaluator as to the other interpreters, we model it as the application of a new tower—the meta-tower of the first one:

Note that this second tower is reached not from the end of the first one, since that has no end, but to the side from *any* closure on the first tower. In other words, the meta-evaluator is a property of the tower, rather than of a particular level.

The structure of this second tower is just like that of the first; each of its levels is represented by same kind of structure used to represent levels in the first tower. Thus, the same operations for manipulating reified data may also be used. The reifier and reflector operations themselves may also be the same as in a single-dimensional tower; it is simply that there is another field, the meta-evaluator (or meta-tower), in the data structure used to represent towers, and the content of that field is a tower level running an interpreter. The difference is not in the form of reification, but in the data that is provided by reification.

To the second tower, the entire first tower is a value within its problem domain. So, for example, while for the second tower the problem is a program to execute, for the first tower the problem might be a representation of some problem outside the world of computers, such as weather prediction. Or, of course, it could be another tower to interpret.

The dimensions of the dimensions of the tower

Continuing this idea, we can provide third and subsequent meta-towers, continuing in an infinite *meta-tower* (which we draw as a spiral, to fit it better onto the page; we draw the successive meta-levels not quite orthogonally to each other, so that they do not altogether hide those drawn behind them). Note that for each successive meta-level link, all the levels of one meta-level are accessible to the lowest level of the next meta-level.

These ideas may be continued to any number of towers of levels, as the structure provided through reification becomes more and more extensive. However, however many towers are provided, the relationship between adjacent towers remains the same.

Just as a tower is implemented by its meta-evaluator running in the tower's meta-tower, we can explain the spiral of towers as being evaluated by an evaluator external to the spiral. We draw this as a rod (as if seen in section) running

up the middle of the spiral. This interpreter is, of course, similar in form to any of the meta-evaluators within the spiral of towers. Just as a meta-evaluator of a tower shadows the boring part of the tower, the meta-evaluator of a spiral of towers shadows the boring part of the spiral.

In turn, this program may be the application of another tower... and a form of reflection could be provided that allows access to this tower. It is noteworthy that all these multi-dimensional forms of reflection differ from the first form we saw only in the structure provided by reification. The form of reflection and reification is still the same. The tower type has a field for the meta-evaluator, which is itself a level or tower, and also through the meta-evaluator's tower, a field for the next dimension of meta-towers. However, this is only one of the directions that may be pursued; to implement this direction of provision of meaning, there must be an orthogonal direction also; and another to implement that... and so we have an infinite list of meta-towers to describe each meta-tower. Having established this, we can see that, we now settle down to a fixed-point, in which the structure is the same: from each tower, we have access not only to the meta-tower that interprets it, but also to the meta-tower that provides the link between these two. This is similar to the first form of tower reflection that we have looked at, and obeys the rules explored in sections **??** and **??**. In the terms already presented, this can be represented by the following diagram:

These diagrams show how the same idea of the reflective tower may be pursued to any number of levels.

Pursuing this reflection to higher meta-levels leads to examination of the ideas of what gives an interpretation its interpretation, or *meaning*, or *grounding* (as in the symbol grounding problem). Starting at the highest level, and constructing lower levels, by implementing them, leads to an understanding of meta-towers as a real programming device. As mentioned in more abstract terms in section **??**, it is easier to think in terms of running an meta-evaluator on a system that happens to be in a tower, than to think of using one tower to implement another.

The idea of shadowing is equally applicable to any of these towers and meta-towers. The relationships between them are equivalent, and can be extended to any number of meta-levels—a meta-tower is a fractal structure, where the detail of each meta-level is similar in form to the detail of any other meta-level. Any dimensionality of meta-tower may be implemented, although the generalized meta-tower has its own dimensionality, which is fractal. The known practical advantages of doing this are swamped by the computational complexity of such a system, but the ideas have proved worth pondering to further understanding of simpler forms of reflection.

The important point here is not that any number of levels and meta-levels may be returned by the reifier, but that each new plane of meta-levels is introduced in the same way as all the preceding ones: the deeper information about a level is provided by the level implementing that level. If one level does not provide its subject level (interpretee) with some particular information about itself, there is no way in which the subject can derive that information. This is explained, in terms of a single tower, in section **??**.

It is natural to think of reflection from within; as information about the self is reified, it is brought from outer reaches of the meta-tower in the application. Likewise, in mental introspection, it is normal to think from objective-world views out to deeper abstractions about self. However, an infinite regression along an inner route outwards will never get outside self; whereas other can observe self from the outside, without any regression whatsoever.

Reflection involving such an outsider is an interesting extension to the idea of meta-towers. One possibility is to have two meta-towers, each examining (and possibly interpreting, with the help of a suitable outsider) the other, on a co-routine-like basis—let us call these *co-towers*. Further (in the sense of more encompassing) related forms of reflection involve reflecting on the observer of the co-towering relationship (which is what allows each of the co-towers to interpret parts of the other). However, one thing is common to all these forms of inspection: introspection at any level is only possible when a higher (outer/other) level provides it as a form of extraspection and makes it available to the introspecting level.

4.6 Multidimensional ideas in single-dimensional towers

The full meta-tower system described above may seem far more complicated than is necessary to describe interpretation. So what is its relevance here? It first comes into the system as a way of making the meta-evaluator accessible just as the rest of the system is; but having been introduced, it finds further use in explaining the nature of ordinary reflection, and for explaining the meaning of types in a tower of meta-circular reflection (as explained in chapter **??**).

The two-(or higher-)dimensional tower is not just an interesting complication of the tower, but is a useful concept for designing meta-evaluators for reflective towers of one or more dimensions. Higher-dimensional towers are hard to explain without first looking at the type theory behind reflective towers, and the type theory cannot be explained without first looking further at the towers. Because of this, the following text makes some forward references to chapter **??**, and might well be re-read after reading that chapter.

The benefits of treating even a flat meta-evaluator as though having a tower level itself include a similarity to the standard evaluator that it mimics, and a better match of the type systems, as the requirements for the evaluator and the meta-evaluator are very similar, and may even be identical (see section **??**).

The use of reflective techniques in this research helped to design a clean, concise standard interpreter, and the same conciseness has shown through in the meta-evaluator.

In particular, the tower's model of each level, and of the links between levels, has helped to define the meta-evaluator's view of each tower level and of its access points to the tower. This is a first shift of a new orthogonal level (see section **??**) and non-identity mappings of intensional values in and out of the first tower are useful here. These are kept to a minimum, as they involve computation instead of just copying, and so are slower than identity mappings. The mappings are really part of the meta-evaluator, and not visible to the procedures within the tower unless made available by the meta-evaluator. This strictly one-sided mapping ensures that while the tower can inspect *itself* absolutely completely, it has *no* access to the meta-evaluator, other than that which the meta-evaluator specifically provides. Thus, the appearance of the tower to itself as infinite and enclosed is utterly maintained. Here, use of the reflective model of level-shifting evaluation has been made explicitly, to prevent reification from occurring at a particular point.

This way of designing the meta-evaluator leads to the meta-evaluator being very similar to the normal evaluator, but extended to handle the level shifting. It is a significant, and pleasing, discovery that the difference between the standard evaluator and the meta-evaluator is remarkably similar to the standard evaluator itself (or rather, to the difference between the standard evaluator and the identity function)—there is a correspondence, perhaps even an isomorphism, between interpretation itself and level-shifting, as both implement the injection (or extraction) of meaning at one level by the level above.

Unidirectional visibility is provided automatically, as an evaluator always sees its structural field. The other side of intervisibility—the structural field seeing the evaluator (and therefore the evaluator being within the structural field of the evaluator's structural field)—is only possible if given by something that already has that evaluator within its structural field. This something may be the evaluator's evaluator, or the tower's meta-evaluator—or it could simply be any single operator at such a level.

This point is re-iterated several times in this thesis, at several levels of meaning; it is central to understanding the meaning of reflection, and also to the implementation of reflection. What matters is not the infinite regression, nor even the shadowing (see section **??**) but that reflection in level n can be provided only by level $n+1$ (for flat reflection) and tower reflection of level $n+1$ into level n can be provided only by level $n+2$.

4.7 What does this architecture allow us to do?

Using this model of interpretation, we can change the way that a program is interpreted, in accordance with a simple, consistent model—very distinct from the traditional type of self-modifying program. These changes may be composed systematically through the order in which the changed interpreters are installed

in the tower.

The reflective nature of the interpreter makes it possible to implement such language features as variable length argument lists through an orderly model of access to the state and code of the program and its interpreter.

It also provides, through giving procedures access to the program, a generalized form of Lisp's `fexpr` mechanism—procedures which do not evaluate their arguments automatically—and thus may be used to define language constructs ("special forms" in Lisp terminology). This makes possible the addition of language constructs by procedures at the same level of interpretation as the program in which they are to be used.

The `fexpr`-like facility is one way in which reflective interpretation can be used to define language features. Another facility is that for changing the underlying method of interpretation, independently of changing individual constructs; for example, tracing might be added, or the implementation of variable lookup changed. How this is separated from the definition of individual constructs is described in chapter **??**.

One way in which systems based on such reflective meta-towers differ from many languages and their implementations is that everything in the system is a first-class value—even such things as languages may be passed around and manipulated just as, for example, number may in many languages. This removes (or allows languages to allow programs to remove) many of the restrictions that many languages impose. In effect, it allows each language to be its own meta-language.

The multi-dimensional form of reflective interpretation allows the same things to be done to the tower implementation mechanism (meta-evaluator) that reflective interpretation allows for the interpreter implementation (evaluator). While perhaps not adding as much useful functionality as the first dimension, this does enable further control over the system; but possibly more significantly, it explains how reflection in the tower works by relating it to something more general and more powerful, and also and also clarifies that it is neither the number of levels nor the number of dimensions that matters, but whether a particular transfer of information or control is moving inwards or outwards in the towering system.

This gives us a new tool for looking at the symbol grounding problem in the context of procedural language interpretation. Accepting that no system can ground itself (an interpretation of Gödel's incompleteness theorem) it provides not only a model for moving towards and away from the ground of the system, but it also describes grounding of one tower in terms of another tower, which we treat as being grounded itself—an expression of Turing's "oracular computability" [**?**], in which, for each tower, the tower of its meta-evaluator is the oracle. With this description of groundedness as being only the concern of two adjacent level (or strings of levels, see diagram on page **??**), we can describe it in terms of the first meta-level that can describe the relationship between the two levels concerned.

4.8 Summary of links between levels

The tower of interpreters is made up from links between adjacent levels of interpretation. Reification and reflection are complementary operations, and they use complementary links between tower levels. Since both links are set up by parts of the calling mechanism, they always occur in pairs. These links make a bidirectional chain throughout the tower.

Since a tower is an infinite structure, any computation involving all levels of it cannot terminate. An application that uses reflection and does terminate must therefore make reference to only a finite part of the tower.

Therefore, the infinite tower may be represented finitely, and it is possible to reason about how many steps of interpretation are needed at one level to implement each step of interpretation at a lower level.

To make the finite representation of an infinite string of identical levels, we make the highest part of the tower into a circle, in which the same level occurs again and again. Whenever an interpreter tries to reify the level that makes up this circle, a hidden meta-evaluator makes a copy of the level in the circle, and passes that *copy* out as the reification of the level in the circle. The application can then modify the level it has been given, without upsetting the level that is still in the circle.

Thus, the infinite part of the tower is stored compactly as a circle, which is unrolled on demand to produce an infinite supply of identical levels. To the application, this is indistinguishable from there being a real infinite chain of levels at the top of the tower, instead of the circle and the unrolling mechanism.

The idea of the reflective tower and the means of reflection may also be applied to towers of towers—that is, meta-towers. Meta-towers might at first seem complicated to reason about, but an understanding of them brings a readier understanding of ordinary towers.

The design of the meta-evaluator, and particularly the mechanisms for detecting the need to unroll a new level from the circle and for unrolling the levels, are a major new development of this thesis.

Such knowledge is too wonderful for me; it is high, that I cannot attain unto it.

Psalm 139:6

Chapter 5

Mixing Languages

5.1 Introducing mixed languages

In this section, we look at the idea of having different languages at different levels of the tower—a possibility mentioned in earlier work on reflection [**?**], but not investigated further there, as, for simplicity in describing reflective interpretation, the languages were taken to be the same at all levels. However, in the Mix system [**?**] the languages are taken to vary between the different programs being mixed together.

The desirability of mixed languages

Mixed-language systems have been around for a while now, often for specific tasks. They are used when specific features of several languages are all combined. Typically, use of one language is nested within use of another. This may be done as a way of extending an existing language, or as a way of developing a new language without having to develop the parts of it that are already done adequately by an existing one. There are several such language combinations in common use on Unix systems, for example:

- *make/sh* in which *make* provides declarative control, and *sh* provides command parsing, environments, and non-declarative control.
- *lex/C* and *Yacc/C* in which C is used to perform low-level work under higher-level control from the parser generation language.

Other language systems which in effect combine languages are those in which different areas of one language are so dissimilar that they are handled separately, or are separated for convenience of implementation. Examples of this include:

- *Common Lisp / format strings* where the format strings are a specialized language embedded in the Lisp system

- *troff / eqn / tbl / pic / refer* which are a nest of languages which may be mixed in one program to produce a document, each of the languages being much simpler than a single language providing the same programming functionality itself.

Such arrangements, being well-established in practice, have shown themselves to be powerful and effective. They are, perhaps, examples of the best tools being selected for particular jobs, making elegant combined tools available. This may also take some other forms, such as merging pieces of assembly-language code in-line in FORTRAN or C (`asm` statements) or use of such a feature as the unix `system(2)` call, which allows a C (or other compiled language) program to interpret a line of shell, which might run any form of program—either a shell script or a program in any compiled (or interpreted) language—while also using the shell's own features, such as wildcard expansion.

When languages occur in groups or pairs like these, often one of them is being used as a higher-level language than the other—closer to its application domain. For example, in YACC, the YACC parts of the program are a higher-level description of what the program has to do, while the C is used for low-level actions. In many of these cases, the higher-level of the two is also more specific to its application field, and the lower is a relatively general language, for example *tbl* is specialized for typesetting tables, whereas *troff*, with which tbl works, is a general-purpose typesetting language.

It is interesting to note that Unix—a system designed, or allowed to evolve, more on practical considerations than for abstract theoretical elegance—includes, in its model of loading and executing programs, a device for allowing program files to be interpreted automatically by an interpreter specified in the file. When a file is about to be loaded for execution, its first few bytes are examined for being a particular magic number (representing the ASCII string `#!` or `#!  /`). If this is found (instead of the magic number value that indicates an executable machine-code file), the program in the file named in the following bytes of the original file is run instead, using the original command line arguments but with the original filename and optional flags (from the original file) prepended. This mechanism, originally provided to allow a choice of shell languages with automatic selection of the right one for each shell script, allows an interpreted program (in any language) to be run in place of a compiled one—with the one restriction that the interpreted language must allow `#` as a comment character! This is a tower of interpretation, although one which does not necessarily provide a program with means to access itself or its interpreter.

Another way in which mixed-language working is useful is in the provision of libraries of useful routines (for example, numerical algorithms); with cross-language calling being available, a library written in one language (chosen for its suitability for that use) may be used by programs written in any language, thus avoiding the need either for writing a version of the library for (and in) each language from which it is to be used, or for writing an interface library (typically written in assembler) to perform the adaptations from one language's calling protocol to another's. A practical example of this is that the NAG

FORTRAN library is now provided with a set of C header files, to enable it to be called directly from C programs.

With truly first-class mixed-language working, there need be no visible seams where routines written in different languages fit together.

Here, we attempt to formalize mixed-language working, and provide a general means for describing languages.

5.2 The requirements of mixed-language working

A system in which mixed-language working is available as a first-class feature has several requirements on its design not usually taken into consideration when designing an ordinary, single-language, program execution system.

A common framework for interpretation must be found into which a wide variety of languages may be fitted. At the same time, this framework must be suitable for reification and reflection—programs, program state, and languages must be in forms that can be manipulated readily: everything should be decomposable (or destructurable) to a suitably fine level—for example, it should be possible to extract the representation of a variable binding, a statement in a program, a term in an expression, a construct in a language—and all these representations should be the same in all languages.

There are two areas in this design task: a suitable representation for the static parts of the system—program and interpreter; and one for the dynamic parts—the program's state and the interpreter's state.

It turns out that the most flexible representation for the procedures of a program leads naturally to a representation for languages which has a suitable granularity for picking out individual language constructs and manipulating them. We weave this representation into the basic structure of closures (as described in chapter **??**), and in doing so find a solution to storing the dynamic part of the system: the application data and application state data, which are stored in the variable storage (value list and environment) of the program and its interpreter respectively. This form of data storage fits many languages' models of variable access very well.

A common program representation

Using a program representation based on parse trees and a state representation based on stacks or continuations, our mechanisms can support a wide range of languages. *Lisp* in some form (Scheme being perhaps the best example [**?**]) is perhaps the language that is closest to our model—also having the advantage of very little syntax. Scheme is the dialect most commonly used for reflection experiments so far, but the differences are not significant in the examples given. Being a simple representation of a parse tree, a Lisp expression is a convenient way of representing constructs parsed from other languages, and so we use it here as our main language for describing features relevant to languages in general.

Since the core of Lisp is a particularly simple language, we will also give special attention to how it maps onto our reflective system, taking it as the primary example language.

Also, Lisp is both a functional language and an algorithmic one, and thus its structure is able to receive conveniently mappings from either of these types of language. The form of Lisp that we use here includes state-dependent operators such as `while` loops, and assignment.

Much of the code given in examples in this thesis could be run on almost any Lisp system; the parts that are not in the languages provided within Platypus are actually Common Lisp [**?**].

In our representation, node (expression, or statement) in the parse tree is identified by its first element, a leaf (symbol or name, rather than a further branching sub-tree) which we call the operator of that node. This is always the case, no matter how the construct appears in the textual form of the language. For example, the expression written in many languages as `a + ( b / c )` is stored here in the form `(+ a (/ b c))`—its Lisp-like representation.

How various languages fit this model

Different languages map onto this common structure with varying degrees of ease: the model was designed for flexibility, but paradigms of computation vary widely, including pattern replacement languages such as *SNOBOL* and constraint languages, for example, along with the normal Turing Machines and algorithmic and functional languages.

Mapping *Lisp* onto our model of interpretation is trivial, since, as explained above, the model is so similar to Lisp.

The *block-structured algorithmic languages* also fit the model quite well, since their block structure is naturally easy to represent as a parse tree, and our model of interpretation has much in common with them. Those languages, such as *Pascal* and *Algol*, which are quite careful about type conversions and coercions fit in cleanly; *C*, with its freeer typing system, fits the control structure easily, but is not so comfortable on the tagged type system.

The *non-block-structured languages* such as *FORTRAN*, *COBOL* and *BASIC* can be fitted in, although slightly awkwardly. Their procedures must be written in terms of sequential execution operators which work their way through a series of statements (sub-trees) and which also provide facilities to switch (jump) to specified sub-expressions. (This switching may be done by a separate *GOTO* operator, or by the sequence operator directly.)

Stack languages such as *FORTH* and *PostScript* fit in as block-structured algorithmic languages; their stack-based data storage is not a significant difference as far as we are concerned, although these languages are not usually regarded as being in the *Algol* group.

Logic languages, such as Prolog [**?**], are one of the worst matches to our model; their definitions must be procedurized (as explained in section **??**) to produce something that we can store in terms of *or*s, *and*s and *binding*s. However, once procedurized, they fit our model of environments well, particularly in

how they can then share bindings/instantiations with non logic-based languages, as explained in section **??**.

One complication that occurs here is backtracking, which may be handled using continuation passing [**?**]. In the *committed-choice* variety of logic languages, this problem is obviated by not providing backtracking.

A further problem in interfacing logic languages to other languages (a general problem, not unique to this system) is that logic languages may return a choice of results, rather than a single result. This is part of quite a general difference between families and languages.

Object-oriented languages, which may be regarded as a variety of algorithmic language, can fit this model comfortably, using a closure to represent each object. The `language` and `type-evaluators` of the closure are then the method dispatch table for that object. A variety of object-oriented language that is clearly suited to constructing this way is the *actor* languages described in [**?**].

Reflection is commonly associated with object-oriented languages, perhaps largely because of the fame of the reflective object-oriented language SmallTalk80 [**?**]. In developing Platypus, I have been careful to avoid an object-based style, to show that reflection and object are entirely separate facilities in a language system.

One interpreter, many languages

This thesis is about a system in which languages are values that can be passed around—which inherently has the ability to be a mixed-language system. We have described in section **??** a system using one standard interpreter. How can we extend a single interpreter to handle a variety of languages? The ability to do this (and many other benefits) depends on a suitable abstraction for a *language interpreter*, and constructing that interpreter is a central part of this thesis.

5.3 An abstraction for programming language interpreters

Our abstraction for language interpreters is based around the parse tree abstraction for procedure representation. The language is arranged as a collection of named *operators*, where the names of the operators are those that appear in the nodes of the parse tree, and the definitions referred to by those names are the closures of the procedures used to implement each named language construct. Thus, a language—as we represent it—is a mapping from operator names to operator definitions. This mapping is done by binding operator names to operator implementations in an environment. We sometimes will refer to such environments as *languages* from here onwards, and any references to `languages` in programs are to this kind of value. In mixed-language interpretation, each part of the program must be interpreted in the appropriate language. We take

the closure as a suitable unit for linguistic atomicity (if nothing else, the syntactic considerations would make a finer grain cumbersome!) and so we include in each closure the language of that closure, stored as an environment binding operator names to the corresponding definitions.

Such a use of environments other than the variable binding environment(s) assumes the provision by the substrate system of multiple environments [**?**] and environments as first-class values. The way we may acheive this is described in section **??**.

As well as the language (environment of operators) an interpreter contains the *evaluator*, a procedure that glues the rest of the language implementation together. It is the evaluator that starts each step of program interpretation, by finding the operator name for the current node, looking it up in the language, and calling the result of that lookup, with the node as one of the arguments of the call. Operator definitions, in turn, use the evaluator to evaluate their sub-expressions as required.

Although unnatural for some languages (such as *awk*), this simple representation makes representation of many languages (perhaps most current languages) simple and concise, and I consider that it has proved its worth in practice. This is supported by the common use of Lisp as a basis for implementing other languages.

In the preceding chapters, we have used the term *interpreters* as a general term for what, from here on, we call *evaluators*, and what we had called the *standard interpreter* is now called the *standard evaluator*. The evaluator-and-language model is just one specific model for constructing an interpreter, and so interpreter is the more general term. We will still use it sometimes when referring to interpreters but not specifically to evaluators.

An `evaluator` is a closure called with two arguments: the evaluand that it is to process, and the level which provides the context (environment, language etc) for that evaluation. Since the closures in the argument level have closed over all the information needed to interpret the program it contains, no other information is needed by the evaluator. The program, arguments, environment, language, and even the evaluator itself are all included in the closure.

In section **??** we will see that this evaluation function is performed by one case of a more general level-based evaluator procedure. This is because the fundamental evaluator rôle is not the only part of a level's evaluator that must be thus changeable. The cleanest, most concise, implementation of a tower level evaluator turns out to be that in which each level has a general evaluator rather than an evaluator just for closures.

The interpretive closure

Reflective tower evaluation adds some new requirements to the conventional closure operation, since more information is now available in the current state of a program. To hold the extra information, we add some new components to our closures. The closures we use in this system contain all the usual parts of a closure:

- The procedure to evaluate
- The arguments with which the procedure was called
- The environment in which the procedure is to execute

and also two new parts:

- The `language` in which the procedure is to be understood. This is an `environment` (see section **??**) binding *operator names* (see section **??**) to the implementations of each operator
- The `evaluator` with which to process the procedure

The implementation of an operator is a procedure which interprets occurrences of that operator when applied to a level containing that operator as the operator of its continuation expression. The mechanism performing this application is explained in detail in chapter **??**. The operator definition procedure is itself represented as a closure, and hence starts a new tower. Like an evaluator closure, an operator closure takes as its first argument the level containing the closure on which it is to operate, and

Independence and interdependence of closures

This organization of the interpreter into evaluator and language brings several benefits:

- it separates the rôle of control of evaluation (the evaluator) from that of implementing individual operators, control structures and so on.
- it makes the language into an abstract data-type that is easy to implement;
- the interface between operator definitions and the rest of the system is simple and well-defined
- each operator definition is separate from all others, so it is easy to add new operators, and also to analyze the action of an operator in isolation, making assumptions about the way it is called, and proving those assumptions separately.

This leads to a very simple standard evaluator, with a clean structure to it, that is flexible for implementation of a variety of languages (as discussed in section **??**), and that makes it easy to modify and extend languages through reflection.

It also means that the same evaluator, being a parameterized interpreter, may be used to interpret programs in a variety of languages by providing the appropriate operator definitions. Here, for example, are some possible combinations. Note that the language must match the program—that is, it must provide all the operators that the program uses—whilst either evaluator may used with either language and program.

Also, different evaluators may be used with the same language, calculating the same results in a different manner. For example, a level could be evaluated strictly or lazily, with or without tracing or single-step and so on. All this can be done without altering or needing to understand the operator definitions. This is possible because the interface between evaluators and languages and operators is well-defined and fixed.

Modifying several interpreter levels allows changes to be composed together. For example, Facilities such as unusual evaluation orders and traced evaluation may be combined by adding independently several special evaluators. For example, tracing of a lazily evaluated program may be done by adding a tracing evaluator next to the program, and a lazy-evaluation evaluator next to that. Were the two added evaluators to be the other way around, the effect would be to trace how the lazy-evaluation works; thus, combination of tower levels is non-commutative. Such a change in the processing of one level may affect all lower levels. For example, it is impossible for a level to do strict evaluation if a level anywhere above it interprets its subject lazily. This is an example of a property being propagated pervasively through the tower. Likewise, the results of adding a tracing level will be affected by the programs of all levels below it—this really means that the tracedness affects all the lower levels, although the visible effect will be only in the tracing output—the semantics of the traced levels should not be affected.

Furthermore, since each closure has its own evaluator and language, the properties can be changed per closure, so individual closures can be interpreted differently, allowing such things as tracing of specific procedures or operators.

Including the language in the closure also makes inter-language calling identical to intra-language calling. Inter-language procedure calls depend on having a common data representation between the languages. This is covered in chapter **??**.

5.4 Writing interpreter components

Defining new operators is straightforward, because the implementation of an interpreter is divided into small parts (the evaluator and operators) with a simple, clearly-defined interface between them. This interface hides decisions about the implementations of other operators and the evaluator, while providing all the information that is needed to interact with them.

An operator definition, as bound to an operator name in a language, is a closure, called in the same way as an evaluator, that is, with the level it is to evaluate as its argument. Thus, both evaluators and operators are written as ordinary procedures, taking each one argument. The argument is the closure that is to be evaluated, and has all the information needed for the evaluation

closed into it, stored in a standard form. Operator and evaluator closures have no other interface to the rest of the system.

Since it is a closure, each evaluator or operator contains the definition of the language in which it is written, and contains an evaluator to evaluate it, evaluators and operators can be written in any language available on the system. A base language, containing operators shadowed by the meta-evaluator (see section **??**) is provided and is designed to be suitable for use in evaluator and operator definitions. When interpreted by the standard evaluator, procedures written in the base language are shadowed, and so run faster than any other part of the tower. The base language is described in chapter **??**. The base language operators provide basic flow control and provide for locating parts of a closure such as the argument list of the procedure being interpreted, and provide reflective and other primitive features.

A closure is an interpreter sufficient for interpreting the level below it if it provides all the operators needed by the evaluator of that level. To maintain also the integrity of the tower for reification, it must include a full set of reflective operators. When these reflective operators are provided in each level, code reflected into a level can then reflect into the next level, allowing reflective procedures to capture and pass back reified data through any number of tower levels:

Here, the actual action of `reify` has taken place through something running in the interpreter, but returning its result to the interpreted program.

Chapter **??** has more detail about requirements for languages used by evaluators.

This closure-based structure for calling parts of interpreters simplifies writing reflective interpreter components, because all the information needed is reachable through the level passed as the argument to that part of interpreter. Also, since each level is the base of a tower by virtue of having a pointer to its evaluator, the information in the level also includes all the levels above it.

We now give some example operator definitions. They are written as procedures each taking one argument, the level which they are to evaluate.

```
(defun if (level)
  (let ((expr (closure-continuation-expression
                (level-continuation-closure level))))
    (if (eval (substitute-expression level (second expr)))
        (eval (substitute-expression level (third closure)))
      (eval (substitute-expression level (fourth closure))))))

(defun + (level)
  (reduce #'+
   (map 'list
    #'(lambda (x) (eval
                   (substitute-expression
                     (level-continuation-closure
                        level) x))))))
```

5.5 The overall structure of the tower

Although each individual level is simple, the overall structure of the system is complicated. Not only is each level the base of a tower which stretches through and beyond the evaluator, but it also has in the language an environment of closures, each of which also starts a tower.

In fact, the structure is more complex than this, as each level of each of those side towers will start a new side tower from each of its operator definition closures, and so will each of those in turn—so as well as having infinitely many levels, a tower built on this basis has a number of branches—or new towers—coming into existence at each level. As can be seen in the text and diagrams of section **??**, and as shown again in section **??**, all of these are connected with the meta-evaluator just as the main tower is (and all those which are ever used must be grounded—as the details, presented later, of the meta-evaluator will show, in practice they all have the same meta-evaluator).

It is through the grounding of all these side towers that the whole tower is evaluable, and so we can see that all of these towers must use the standard interpreter at some stage. Since the retreating shadow technique described earlier (see section **??**) is used when needed, the towers do not need to rejoin into a single tower at the standard interpreter. Instead, as the standard interpreter is infinitely far away from each level, all the towers go off to the same infinity, in parallel lines.

The meta-evaluator arranges that these parallel lines do meet at infinity, or at least just past the horizon. This is a good way to think of the meta-evaluator. Although it makes the lines meet at infinity it also makes sure that that infinity

is just a little further than the furthest away that we can see or touch without moving from our present position. Although the diagram here shows all these towers as being of the same length and form, their lengths cannot really be compared (all being infinite) and their forms may vary as long as they are all grounded.

Remember also that the meta-evaluator is alongside all these towers, rather than just meeting them at their infinitely far far ends. One might perhaps choose to regard the meta-evaluator as the paper on which the towers are drawn.

There is the multi-dimensional form of reflection, mentioned in section **??**, in which a program can access its tower's meta-evaluator as the base of another tower, perpendicular to the first tower. This is useful for thinking about the implementation of the first tower, because it connects the tower to the tower's implementation in much the same way that tower reflection connects a closure to the implementation of its interpretation. Also, each of these grounded parallel towers described above meets its meta-evaluator (meta-tower) at all points along its length.

To include the meta-evaluator in the system, we must give up the first tower's pretence that there is no meta-evaluator, and provide a means for going from one tower to the next. This can be done in defining a type for towers, by making the type connect a tower with its meta-evaluator/orthogonal tower, and allowing any program to find the larger tower of which it is part, as described in section **??**.

Since these links lead to the next meta-level, access to further dimensions of the spiral of towers can be made possible if that is provided in the type of data used to represent towers.

5.6 What can we do using this model?

This model for interpretation and interpreters brings in several new possibilities.

- It allows fully transparent inter-language calling. This is possible without such a strongly structured model; it depends only on one area of the model, the common data representation for all languages: environment variables, local variables, procedure arguments and results, as well as the actual representation of values (integer, real, string, symbol etc), are what matter here. This in itself does not involve reflective techniques at all.

- It allows a language to be extended by the addition of new constructs (operators), usually completely independently of other constructs in the language. It also allows existing constructs to be redefined individually without altering other constructs.

- It allows the model of interpretation to be changed without reference to individual constructs. Thus, language-independent interpreter modifications (and other tools) may be written: for example, a version of the interpreter that steps through a program under interaction from the user may be written, that may be used with any language in the system.

- The interpreter, when reified or otherwise examined is in a more suitable form for manipulation that an *ad-hoc* interpreter procedure would be. Programs and interpreter of one language may be handled by any language with equal ease, subject to the operations the language provides on values of the appropriate types.

An unusual feature of this model of execution is the flexibility of its variable storage. The values list may be used as an open stack for stack languages, or used as though a framed stack for languages with stack frames. Using it for procedure arguments and results)as well as for the procedure's local workspace) is done in such a way that the arguments and results for a callee procedure in one language are handled correctly by the caller whatever the caller's language—even when the call goes between framed- and open-stack languages.

The environment variables are suitable for most procedural and functional languages, and are also usable by deductive languages such as Prolog to hold their bindings (instantiations) in. When a procedure in a deductive language calls one in a functional or procedural language, instantiations made by the former are visible as ordinary variable bindings to the latter. Likewise, a deductive procedure called from a non-deductive one will see as instantiations any non-local variable bindings that the latter (and its callers) may have made.

5.7 Summary of mixing languages

As well as being a reflective system, our system is a mixed-language one, making reified languages into part of the evaluation data that is available for manipulation by application programs running on the system.

To make the language a variable part of the context of a closure, we divide the interpreter into two parts, the *evaluator*, which is language-independent, and the *language*.

To do this, we need an abstraction for languages. The abstraction must be general enough to handle most languages reasonably well, and to handle all languages to some extent. Such an abstraction can be devised only in conjunction with an abstraction for the programs in the languages. For the programs, we choose to use parse trees, with each node being identified by its *operator* such as `if` or `+`, and for the languages, we use environments binding operator names to operator definitions.

This abstraction makes it easy to add new operators to a language, and also keeps separate the general evaluator, thus making it possible to redefine the evaluator independently from the language.

By making the `language` (the environment binding operator names to operators) of each tower level an explicit part of that level, we extend tower reflection from being a tool for reasoning about interpretation of programs to also being one for reasoning about languages and their interpretation.

"When I use a word," Humpty Dumpty said, in rather a scornful tone,
"it means just what I choose it to mean—neither more nor less."

Through the Looking Glass, Chapter 6

Chapter 6

Types, abstraction, and representation

We now look at the abstract types needed to support interpretation in a reflective tower system. Development of the type system for multidimensional towers was an important step in developing these ideas on reflective interpretation.

6.1 Representing the abstract: concretion and extension

In section **??**, we have looked at one aspect of the concept 'intension and extension'; we have seen it as applied to evaluation of expressions or interpretation of procedures. We now look at this topic in the more general sense of representing any abstract things in a computational (or otherwise linguistic) system.

The fundament of our mechanism for representing things is *Gödelization*—representing each word in a language by a digit in some numbering system (the natural numbers being the common case), and each instance of a linguistic construct by the sequence of digits that represent each of its components. (The technique comes from the work of the mathematician Gödel [**?**], who used it in describability and computability proofs.) Any way of representing something in computer memory must be a form of Gödelization, since whatever is represented must be represented by a pattern of bit groups, as that is all that is available.

In sections **??** and **??** the ideas of program as data, language and evaluator as data, and evaluation (an activity or process) as data were presented. All of these are built on forms of Gödelization, and presuppose the facility for representing programs and mappings as part of the application domain of a program.

Gödelization of processes

In looking at intension and extension, we saw that a procedural representation (which is intensional) for a process becomes a representation of a real (grounded) process only when an evaluator built of, or upon, an entire extensional definition of the process of interpretation is applied to it. (The representation of application (λ-substitution is a suitable model for application here) is the substitution of a number encoding a formal parameter with the number representing the actual parameter.) However, the evaluator must also have an intensional prescription (which is what is applied to the evaluee) as well as an extensional definition or implementation.

Thus, somewhere within the system there must be a link between the intension and the extension of the evaluator, that allows the evaluator in turn to realize the link between the intensional prescription of a procedure and the extension of it—a process performing that prescription. What is the nature of this link? Is it extensional? Yes, because a prescription of its function, *given an oracle* [**?**], can be written. How is the extension of the link projected onto the link's intension? By another such link—and this is part of the description of the link.

The example above, of representation of procedures, shows the representation of one abstract thing (a procedure) being used as a concrete implementation by a context-provider that already has a concrete (extensional) implementation itself. Although most of this thesis is concentrated on representing the abstraction of procedural interpretation, the same ideas apply more widely to representing abstract things, including, naturally, concrete things rendered representable linguistically by giving names (abstract) to things (concrete) in a computer (or other linguistic) system.

As already described in sections **??** and **??**, the link between the encoding of something as a Gödel number and the use of that number to mean that thing can be supplied only by the user (reader) of the Gödel number. Since we make the user—the supplier of extensionality—be the shadow (extension) of a further intensional description (Gödel number) into which the original number is applied (substituted), the user remains hidden: the meaning cannot be found from the intension alone. For example, given just the intension represented by the word 'skip', we cannot find its meaning. Given a provider of extension that maps this word onto something (an English dictionary, say; or perhaps a Norwegian one; the two will map 'skip' onto quite different extensions, although the word 'skip' is spelt the same way) we can then find an intension for this, if we choose to use the dictionary this way. If, on the other hand, we use the dictionary only to provide a textual substitution, we simply provide another intension ('small jump' in one case, 'store sjøfarende båt' in the other), which in turn either can be understood to represent an extension, or transformed by substitution into another intension, such as, respectively, 'små hopp' and 'large seafaring boat'.

In general, representation of the abstract requires a scheme for representing values outside the system by values inside it, with a mapping being defined between each representable value outside the system and the corresponding

representations within it.

The 'system' referred to above is a level of interpretation, to which an interpretation may be given in two ways: the extensional interpretation in terms of external values, as mentioned above; and an intensional interpretation given by another level of interpretation, to which the values in our original level are the outside values as referred to above. For example, the frequency of an electrical signal may be represented as a number of cycles per second, and that number may be represented in digital apparatus by a bit-pattern, which may be represented in electronic digital apparatus by a pattern of voltages.

In this sequence of abstraction→interpretation mappings, we see a tower of meta-circular definition appearing, much like the tower of interpretation. Each type of value at one level is represented by a value of some type at the level above, and these types are in some way related. (There is general review of types, the need for types, and the use of types in representation, in [**?**].)

Is this tower of types also a reflective tower? Yes, because the types used to describe a level of types may be brought into that level, and types may be moved up and down the tower. Meta-towers of types, describing the mapping of types between levels of types, may also be constructed, and brought into the towers. This reflection is not procedural—nothing happens in it; nor is it prescriptive—it does not prescribe instructions for doing something; it is representational and descriptive.

These towers of representation, like other towers, have their meta-levels linked by shadowing; an intensional type tower (described in terms of its structure, and then in terms of its structure's structure, and so on) may be described without infinite regression by an extensional type (describing what the type means to an outside observer).

Substituting description for prescription, we find that not only are these representational towers isomorphic to procedural towers, but also that each procedural tower is backed by a representational tower, which is more fundamental, in that all procedures require representation, but no representation requires a procedure.

There is a link between each level of the procedural tower and the level of the type tower that describes its types. This link, naturally, has a type element to it and also a procedural element: it is an environment (lookup table) mapping types to procedures for evaluating values of those types. This environment—the `type-evaluators`—is covered in more practical detail in section **??**.

Gödelizing the infinite tower

What lies beyond the type *tower*, as we bring that into our system (that is, Gödelize it—give it a Gödel number, or encoding)? What represents the representation of the representation? This is the meta-tower of the tower of types. Using this, we can continue our Gödelization of the abstract things underlying procedurally directed evaluation. We can also Gödelize this encoding itself, and yet still not describe in full its connection with the ground [**?**], because from

this intensional viewpoint, we can never create an extensional view or definition of the system.

In the discussion above, we have taken Gödelization as the basis for representing values of any type (including types). How can we describe a tower or a meta-tower, given that each tower in a meta-tower is infinitely long, and there are infinitely many of them? Does this require infinitely long Gödel numbers, or may we roll up the boring section of the tower, and encode this as such in the Gödelization, as is done in computer memory as shown in section **??**? Does such an approach even work in this rôle?

It is possible to bring these two approaches together, by allowing Gödel numbers for meta-towers to use not only the natural numbers for their digits, but also transfinite numbers (if the type we use to support the use of digit strings as numbers allows such a mixed polynomial as a number). In the first tower in the meta-tower we label the levels as $1, 2, 3 \ldots$. This is infinitely long, and we can write, in place of an infinite series of numbers (the boring section of the tower) a single transfinite number (the meta-tower shadowing the boring section of the tower). Thus, we now refer to the whole tower as ω.

The meta-tower of this first tower has level ω as its first level, and we label the following evaluator levels as $\omega + 1, \omega + 2, \omega + 3 \ldots$. Taking this meta-tower, and writing out *its* boring section in detail, we have another infinite series of numbers, and can do the same kind of substitution again, writing 2ω for in place of the series $\omega + 1, \omega + 2, \omega + 3 \ldots$, and starting a new tower, with evaluator levels $2\omega + 1, 2\omega + 2, 2\omega + 3 \ldots$.

This infinite series can itself be represented as a meta-tower (as represented by the diagrams in section **??**), and we label this meta-tower as ω^2, and its successive meta-evaluator levels (drawn as gray rods in the diagrams) are $\omega^2 + 1, \omega^2 + 2, \omega^3 + 1 \ldots$.

6.2 Requirements for the type system

Having explored the ideas behind the type system, and its connection with evaluation, we now look at what must be provided in the practical type system that we use in implementing the tower system. As concrete examples, we present several of the types used by the evaluator and meta-evaluator in Platypus.

Static and dynamic typing

We use a dynamic type system; a static type system would have difficulty with interpreters being able to pass around objects of arbitrary types for which their subject programs have called, particularly when the language being interpreted provides dynamic typing. Dynamic typing being more flexible than static typing, it also can be used to support statically typed languages without change.

Kinds of values

The values in a tower reflection system may be seen as being of two kinds:

- the values present in any ordinary computing system, such as numbers, characters, strings, pointers, structures, lists and arrays
- the values concerned with reflection, such as closures, levels, activation stacks, expressions, towers, value lists and environments.

To be able to perform computations on values of either kind, we must have operations handling values of the types concerned. Whether the values are concerned with reflection or not is not pertinent to how we handle them, and no types used here are inherently reflective.

There are two groups of types that we must consider, classified according to whether values of that type may be divided into several parts:

- simple types (such as numbers)
- compound types (such as vectors and structured records)

Simple types

In principle, there are two kinds of simple types (although in practice both are usually represented by numbers):

- those in which the possible values are related to each other in some particular way, and in which a value can be produced from other values (such as numbers);
- those in which values can only be created, and compared for equality (tokens).

Since tokens can be implemented through simple use of integers, we will not provide them separately. (In fact, to give tokens convenient textual names, we use *keywords*, as used in Common Lisp [**?**, Section 11.6]. We also use the substrate Lisp's symbols in general as tokens for lookup in environments.)

Likewise, logical values can also be represented as a range of integers, although their meaning is sufficiently special that we will could them appear as a distinct type (in fact, for convenience, they are implemented compatibly with Lisp, using the symbol `t` for `true` and the empty list `nil` for `false`).

Compound types

A compound type is one in which a value of that type may be separated into *elements*. Each element may be of either a simple or a compound type. We select an element of a value of a compound type by its *index*, which is of a simple type.

There are two kinds of compound type:

- those in which the index is a number (vectors, arrays);
- those in which (in this system) the index is a token (structured records)

Since, as noted above, tokens can be implemented as numbers, structured records can be implemented as vectors. In this system, the vector representation underlying a structured record is visible, although not used as the usual route to access the data in the record. In the Lisp version of the meta-evaluator, records are represented by defstructed objects in Lisp, while in the C version of the meta-evaluator, the array form of record is explicit, although usually concealed through a collection of macros. Such shifts in the nature of the same type are covered in more detail in section **??**.

Note that, like tokens, vectors may be compared for equality, but not compared for order.

6.3 The structure and content of our type system

The above considerations give us three fundamental types:

- truth values
- numbers
- vectors

With these, we can perform all the operations of conventional computing systems, apart from input and output, which we assume are done for us by operators at the next level up, which we will not try to analyze. We must also provide support for the types needed for reified state values. This is done through the types that we already have, as they are expressive enough for this.

Certain types are particularly important in the evaluator and meta-evaluator, as they describe the values that these procedures handle directly themselves. The rest of this section lists and explains these types.

The type type

The first type to describe in a list of types is, naturally, the type for values that represent types—that is, the type of the tags in our dynamic type system. The main use of these is to identify how to evaluate something. Since this may be different in any number of different contexts (such as different tower levels, or an evaluator and the corresponding meta-evaluator) it is neither appropriate nor even possible to build the evaluation technique into the type (it is not the same as class methods in an object-oriented system, as they have only the one context for each type of evaluation). Instead, we simply use the type of an object as a key to look up in an environment (such as a `type-evaluators` environment, as mentioned later in this section and in section **??**, and also in sections **??** and **??**).

It is appropriate, therefore, for types to be represented simply by names. Any information describing some aspect of the type may be found by looking that type name up in the appropriate environment.

The closure type

A closure is a structured record, containing these elements:

- *evaluator*: a (pointer to a) closure, containing the same kinds of elements as this one...
- *type-evaluators*: a structured record implementing an associative mapping from types (which are denoted by their names) to closures
- *language*: a structured record used to implement an associative mapping from operator names to closures
- *expression*: either compound or simple; this is in two parts, the continuation expression and the procedure expression, as explained in section **??**
- *values*: a vector of anything
- *environment*: a structured record implementing an associative mapping from names to values; there are really two environments, the lexical and the dynamic
- *original*: the inactive closure of which this is a copy
- *level*: a pointer back to the level which contains this closure

The `original` field exists to speed comparison in the meta-evaluator. For it to determine whether a closure is an instantiation of a closure that is shadowed by a kernel primitive, the meta-evaluator looks up the original field of the closure in the table (*shadow map*, an environment) of closures that it knows to be shadowed. The original field is copied when a closure is copied, and changed when the closure is changed.

The alternative to having an `original` field is to compare the contents of closures to determine whether they are the same—a potentially time-consuming task. An idealized implementation might do such a full comparison, in terms of whether two closures will have the same behaviour; however this is not necessarily possible to determine, since it cannot even be computed whether each of them will terminate. The `original` field gives us a very quick indication of whether the closure is still identical to the (possibly) shadowed one from which it was instantiated. This test is also fail-safe; it can indicate that the closures are different when they are still equivalent, but will never indicate that they are equivalent when they are not.

Closures have two rôles: when instantiated and made part of a tower they are building-blocks for the state of a computation. Before instantiation they represent a stored program, and are copied from this form into the running closures on the stack by the procedure call mechanism, as described in section **??**.

Many of the data structures used in Platypus contain many parts of the tower. In particular, the value list of a closure (its arguments, workspace and results) will contain whole levels of interpretation if the closure is an evaluator. Because of this, we print some of Platypus' data structures in very limited ways. For example, closures are printed like this:

```
#<{closure 244  (pr 1)
    expr: { EVAL-IN-CL { QUOTE
                         { PROGN
                            { LOAD "pl-in-cl.lisp"}
                            { LOAD "griss.lisp"}}}}
    vals:#<[7: 20
               (IF (EQUAL (LISP-VARIETY) "Platypus")
                   (PROGN
                     (LOAD "snark-interpreted.lisp"
                           "snark-interpreted.out")
                     (EVAL-IN-CL (QUOTE
                                  (PROGN
                                     (LOAD "pl-in-cl.lisp")
                                     (LOAD "griss.lisp"))))))
               #<File stream "griss.out">
               #<File stream "griss.lisp">
               NIL "griss.lisp" "griss.out"]> }>
```

The value list type

A *value list* is an extensible vector of values. When a procedure is called, the value list contains the arguments to the procedure. The procedure, while executing, uses it to hold local variables. When the procedure returns, it will have filled the value list with the results. (Like some Lisps [**?**], and PostScript [**?**], our language model allows for multiple results.)

Platypus prints value lists as shown in the closure displayed above.

The environment type

An `environment` is used to map names to values. In this system, we use an environment representation based on hash tables, with lists of previous values. This is known as *shallow binding*. In shallow bound systems, each symbol (name) holds its value directly; this is suitable only for systems with a very small number of environments (typically one or two, as in Lisp and other languages). Old (hidden) bindings are kept on a list, much as for deep binding.

In other systems, environments are represented as lists of name-value pairs, either as *property lists* or `association lists` in Lisp terminology; this is called *deep binding*. A third possible representation, combining some of the values of each, is described in [**?**].

Another possible representation of environments is procedural, as described in [**?**], and this appears as a possible alternative implementation of part of Platypus in section **??**, page **??**.

Deep binding is suitable for systems with many environments, but searching the binding list may be slow. Platypus uses many environments, such as the `language` and `type-evaluators` of each closure, and it uses them intensively (one or two lookups for every step of interpretation at each level concerned; see section **??**), so speed of lookup is critical. We use a representation using a *hash table* (as provided by Common Lisp, our substrate implementation language; see [**?**, Chapter 16]) to hold the current bindings (much like shallow binding, but not storing the values actually within the symbol) and association lists to hold the saved (hidden) bindings.

The avoidance of building a value slot or slots into each name (names are simply tokens, or symbols, which may be compared only for equality) allows us to use things other than proper names as the names, or keys, in environment lookup operations. This provides more flexibility—important in a mixed-language system, as, for example, in PostScript's dictionaries the keys may be of any type—and also allows more consistency between the evaluator and the meta-evaluator, as the shadow map maps closures to shadow-closures just as type-evaluators and language environments map names to closures—in effect a shadowed closure is a name for its shadow, in the sense that the appropriate environment (or context) is required to find from a meaning at one level the corresponding meaning in another level.

The expression type

There are several kinds of expression. A *compound expression*, often just called an *expression*, is a vector of values. Each value in it is called a *sub-expression*. Sub-expressions are normally constant, although they can by changed through reflection into the expression. Each expression contains

- a first sub-expression called the *operator* of the expression. When the expression is taken in the context of a closure (which it normally is) this is a name, naming an operator closure in the language of that closure
- other sub-expressions, evaluated under control of the closure named by the operator, as arguments to the operator

References to variables are also a kind of expression, usually appearing as not as top-level expressions but as sub-expressions of a compound expression. Two types are used in the implementations in this thesis:

- *local variable names*, which are distinctly tagged integers indexing into the *value list* from the extensible end;
- *variable names* which represent textual names of lexical and dynamic variables.

Other things, such as numbers and string constants, may occur as expressions, and are all usually treated as literal constants.

The printed form of expressions is as shown in the expressions appearing in closures and levels and towers in this section. Non-local variable names are printed as textual names, and local variables in the form `#<local-2>`, where the number of the local counts from the extensible end of the stack, such that the top of the stack is referred to as `#<local-0>`.

The level type

All evaluations occur in the context of a level. Every level always has a current closure, and it is in this that the evaluations within the level occur; this is also where most of the context is held. The level serves to hold the current closure of the level and the list of those saved by `funcall` operations, and also to connect the level with the tower of which it is part, and thence to the meta-evaluator (or meta-tower) within which that level belongs. (The links from one level to those above and below it, as discussed in section **??**, are part of the closure, not of the level, as it is appropriate for these to be closed into particular procedures, and thus to be changeable by reflection to those procedures.)

Also in the level is the `template closure`, used when instantiating an inactive closure (as described in section **??**) into this level.

The components of a level are as follows:

- *tower*: the tower of which this level is part.
- *call-record-stack*: The list of saved closures executing in this level. The front of the list is the current closure.
- *template-closure*: The closure used to fill in default fields when instantiating a closure.

The `tower` field is not needed for a one-dimensional tower, but is necessary when there is a choice of meta-evaluators, and the meta-evaluator for a tower is part of the reified state of the tower.

In Platypus, a level is printed like this:

```
#<{Level running 244:
  { LET-LOCAL
 { NIL
      { OPENIN #<local-2>}
           { OPENOUT #<local-4>}}
       { WHILE { NOT #<local-2>}
          { LET-LOCAL { { READ-FILE #<local-2>}}
             { PRINC "in: "}
             { PRINC #<local-0>}
             { TERPRI}
             { SHOW-STATE "file loop"}
             { LET-LOCAL { { EVAL #<local-1>}}
                 { PRINC "out: "}
                 { PRINC #<local-0>}
                 { TERPRI}
                 { WRITE-FILE #<local-2> #<local-0>}
                 { WHEN { EQ #<local-0> :QUIT}
                     { SETQ #<local-4> T}}}}}
       { CLOSE #<local-0>}
       { CLOSE #<local-1>}}
 on args}>
```

The tower type

Just as a level holds a stack of closures, a tower holds a string of levels, and also a link to the meta-evaluator. This link consists of three fields:

- *meta-evaluator*: The meta-evaluator of this tower, as explained in sections **??** and **??**;
- *type-shadow-map*: A map from type-evaluators—procedures that implement the evaluation of each type, as mentioned above in the section on the closure type—to their shadows, as described in section **??**.
- *operator-shadow-map*: The operator shadow map, mapping operator procedures to their shadows, also described in section **??**.

As well as the link to the meta-evaluator, there is a link to the next dimension of evaluation, the meta^2-evaluator, represented as the grey rod in the centre of the spiral in the diagram on page **??**, and as $\omega^2 + 1$ in section **??**. (A further mechanism, which I have not researched in detail, is required for access to the rest of the series that begins $\omega^2 \ldots, \omega^\omega \ldots, \omega^{\omega^\omega} \ldots$).

A tower is a complex data structure, and describing it in detail can take considerable space. In practice, it turns out that the most useful part of it to print is the program it is running; when working with meta-towers, it may also be useful standard evaluator (which will vary between meta-towers). A tower is printed like this in Platypus:

```
#<{tower with standard evaluator
      { LET    #<<evaluation-point>>
          { { TYPE-EVAL
                { LOOKUP
                    { TYPE #<local-0>}
                    { LEVEL-TYPE-EVALUATORS #<local-1>}}}}
        { IF TYPE-EVAL
             { FUNCALL TYPE-EVAL #<local-0> #<local-1>}
            #<local-0>}}
    and program
      { LET-LOCAL    #<<evaluation-point>>
            { NIL { OPENIN #<local-2>} { OPENOUT #<local-4>}}
          { WHILE { NOT #<local-2>}
              { LET-LOCAL
                  { { READ-FILE #<local-2>}}
                { PRINC "in: "}
                { PRINC #<local-0>}
                { TERPRI}
                { SHOW-STATE "file loop"}
                { LET-LOCAL
                     { { EVAL #<local-1>}}
                   { PRINC "out: "}
                   { PRINC #<local-0>}
                   { TERPRI}
                   { WRITE-FILE #<local-2> #<local-0>}
                   { WHEN { EQ #<local-0> :QUIT}
                      { SETQ #<local-4> T}}}}
          { CLOSE #<local-0>}
          { CLOSE #<local-1>}}
}>
```

Note that since this tower is being evaluated at the time, it has closures with their current expressions being different from their procedure expressions. The current expression is always a sub-tree of the procedure expression. It is marked in each of the expressions above with the marker `#<<evaluation-point>>`, which is not represented as a value in its own right; it is an artefact of printing a closure's two expressions together, being printed when recursive printing of the procedure expression reaches the same point (`eq` in Lisp) as the continuation expression.

6.4 Types and evaluation

The type system of a language is an important part of the language, perhaps as important as the range of statements available. Thus, it is important for a mixed-language system's type system to be flexible enough to meet closely the needs of many different languages, while having enough consistency of its own to allow the passing of values compatibly between procedures written in different

languages, as well as up and down the tower as described in section **??**. This is one of the reasons why it is important for systems such as Platypus to use dynamic typing; static typing is simply not flexible enough to support a system in which any level contains a dynamically typed language.

To make the evaluator flexible enough, it is parameterized by the `type-evaluators` field of the closures it evaluates; this contains an environment binding type names to the closures used to evaluate things of each type. The use of this field is described further in section **??**. There is a shadow map (see section **??**) corresponding to this and used by the meta-evaluator (see section **??**, page **??**). Types not mentioned in the type evaluator shadow map are evaluated by interpreted procedures, so it is not necessary for all types to be known ab initio to the meta-evaluator—new ones may be added at any time at any level.

Using this approach, not only do we make the evaluator extensible to cover the addition of new types, but it also takes on a more convenient form for reification and reflection, as the evaluator for a particular type may be found and replaced simply through access to the `type-evaluators` environment. At an earlier stage in the development of the evaluator, there was a Lisp `typecase` statement instead of the lookup and funcall with the `type-evaluators` environment; a change in the evaluation of one type required replacement of the whole evaluator.

6.5 Types and level shifts

Values may be passed up and down a tower. Since the meta-evaluator does not change the values it transfers from one level to another, values passed between levels keep the same *intension* (representation) while possibly having different *extensions* (meanings). For example, *bignums* may be numbers at a lower level, and arrays of numbers at some higher level. At the hardware level, which is the highest level, they, like all other values, will be words and bit patterns.

In C-Platypus, all types within the tower map onto quite different types in the meta-evaluator: all values map onto structures containing data words and associated tag (type) words; booleans map onto numbers, structures onto arrays, and so forth. This is an example of how the type shifts between levels of a tower are different to type shifts that go in and out of the tower.

In Platypus89, in which the substrate language is Common Lisp, the meta-evaluator and the tower contents both use the type system of Lisp, and there is no shift in the meaning of each representation (with the one exception that string-characters are used to index variable in the local stack frame, as they are distinctly tagged small integers—Common Lisp does not allow the definition of new varieties of atomic types). The use of this is shown in section **??**.

In Platypus, values normally have the same extensions at all levels of the tower, which has removed a possible source of complexity and of inaccuracy. All values also have the same intension (representation) both in the form in which a higher level uses them, and in the form in which a lower level obtains them through reifiers. This means that reifying and reflecting information do not

change the representation. This is an important design point in Platypus. Some reflective systems (such as SmallTalk-80 [**?**]) hold reifiable data in a different form from that in which it is passed back through reifiers. For example, a reifier for a stack frame object in such a system might construct the object from a real stack frame to pass it back to the program; the real raw stack frame might be in a unsuitable form (perhaps outside the type system) for handling as reified data. In the corresponding example in Platypus, the real stack frame is passed back by the reification of a stack frame. To make this practical, the stack frames must be kept in a form suitable both for program execution and for manipulation as reified data.

There are advantages and disadvantages of translating the data between formats like this. Among the advantages are that the reified form might not be suited for efficient program execution, in which case separating the representations allows the form used by the evaluator—the reflected form—to be designed purely with efficient evaluation in mind; and that levels (and other structures) that have to be realized (as described in sections **??** and refunrollmech).

Whether or not the mapping between intensions at adjacent tower levels is the identity mapping may be used as a way to classify level shifts into one of two forms. Level shifts in which this mapping is an identity are closer together than those for which reifiers must alter the representation, as less work needs to be done to move information between the levels. Thus, it may be expected that the first level shift in each tower in a meta-tower (see section **??**) may well change the representation, but others will not. At the lower end of the meta-tower, this corresponds to the extensional shift between the problem domain—the external real-world meanings—and the computational domain of manipulable values. In general, changing the extensional meaning is useful in implementing a tower (the representation may change between the tower and its meta interpreter) but is not particularly useful within a tower, where it is more useful to pass data between levels readily.

As levels share a type system, and can pass data to each other, they also share a *structural field*—they can share data with no need to transform it as they pass it amongst themselves. And yet, they also each have their own structural fields, as the level interpreted by each interpreter is the structural field of the level above it and as each level can access all other levels, the whole tower (or meta-tower) is the structural field of each component—and this includes that component itself.

Whatever the form of the shift between intensions at different levels, there must be a defined mapping to do the shifting. Without this, the tower would be broken for purposes of reification and reflection, and could only be used for interpretation, imposing a one-way restriction at one of the level shifts. This is done deliberately in implementing the meta-evaluator in C-Platypus, to ensure that the meta-evaluator always remains hidden from the tower.

In making representations compatible between levels, we also make them compatible between the languages of each level, mentioned as a requirement in section **??**. This may require some flexibility in matching the underlying representation to each language. For example, some languages, such as Lisp, do not

have a distinct boolean type, and others, such as C, do not have bignums (digit string arithmetic), and yet it might be appropriate to provide for these in the base language and the standard interpreter. Fortunately, the common grounds of computability and computer architecture have pushed language designers' intuitions toward a reasonably consistent central set of types (such as integer, boolean, string, array, record).

With a common type system, routines written in different languages can pass data between themselves without having to make any modification or translation to the data. This keeps inter-language calling equivalent to intra-language calling.

6.6 Summary of types, abstraction and representation

The ideas behind the towers' type system are important in understanding tower reflection. Types are an essential part of the way we represent values, and the mapping from one tower level to the next is a representation of a value in one system by a value in another. The basis for representing values in a computer is Gödelization, in which digits in numbers denote words in a language.

The type system we use must allow the representation of procedures and of procedural evaluation, as well as the representation of the application's problem domain. It must also be possible to represent the infinite towers and meta-towers used in reflective evaluation.

The system must provide operations on types concerned with reflection (that is, types for objects representing parts of the tower) as well as for the types of objects normally handled by an interpreter. We divide types into two kinds: simple and compound.

A few types are of particular importance in a reflective tower. Closures are the central type. Other important types include expressions, environments and value lists.

As information is moved between levels, its representation may be changed, although in Platypus it is not changed. The meaning of the same information may be different at different levels even when the representation is the same.

Although the meaning and representation of information does not normally change between levels of a normal tower, it may well have to change in going between the tower and the meta-evaluator that implements the tower.

> The shop seemed to be full of all manner of curious things—but the oddest part of it all was that, whenever she looked hard at any shelf, to make out exactly what it had on it, that particular shelf we always quite empty, though the others round it were crowded as full as they could hold.
>
> "Things flow about so here!" she said at last in a plaintive tone, after she had spent a minute or so in vainly pursuing a large bright thing, that looked sometimes like a doll and sometimes like a work-box, and was always in the shelf next above the one she was looking at. "And this one is the most provoking of all—but I'll tell you what—" she added, as a sudden thought struck her. "I'll follow it up to the very top shelf of all. It'll puzzle it to go through the ceiling, I expect!"
>
> But even this plan failed: the "thing" went through the ceiling as quietly as possible, as if it were quite used to it.
>
> *Through the Looking Glass, Chapter 5*

Chapter 7

The standard evaluator

The essential rôle of the evaluator is to evaluate the expression of a level in a context provided by the information in that level. To do this, it must

- find the type of the expression
- look the type up in the current closure's `type-evaluators` environment to find a type evaluator
- apply that type evaluator to the expression and the level (the level being passed in to provide the context for evaluating the evaluand expression)

and then, if the expression is a compound expression (like a list in Lisp), it must

- find the current operator, from the expression tree of the level it is processing,
- look that operator up in the current closure's `language` to find an operator definition
- apply that operator definition to the expression and the level

A very simple function proves to be all that is necessary both for interpretation of the level below it and for preservation of integrity up the tower. This chapter presents and explains that function.

7.1 A general evaluator

The evaluator function is complicated slightly by being both an expression evaluator (like Lisp's `apply`) and a general evaluator (like Lisp's `eval`), used for example, for variable lookup. This was not the case in early stages of this work, but turned out to be the cleanest way of making control of evaluation easy to reflect into. Without this, certain changes (for example, between different forms of variable binding) would be much harder to reflect in to the system,

and could even involve two levels of reflection instead of one (which potentially takes a considerable penalty from performance, as the interpreter's interpreter is interpreted by the meta-evaluator, instead of the program's interpreter). An alternative to making the evaluator be a general evaluator is to have several specific evaluators as fields of each closure; a tidy form of this is to have a map from types to evaluators—in effect a language whose operators are type names and whose operator implementations are type-specific evaluators. This map is the `type-evaluators` field of the closure structure, as mentioned in sections **??** and **??**. It is onto this form which, after experimentation, the implementation in this thesis settled.

7.2 The structure of each level

Using the model of procedure calling described in section **??**, we make towers in the following overall form:

```
(defstruct tower
  meta-evaluator
  operator-shadow-map
  type-shadow-map
  base-level
  standard-evaluator-closure)
```

where the `meta-evaluator` field is the procedure that makes the tower run, and the `shadow-map`s are used by the meta-evaluator as described in section **??**, as is the `standard-processor-closure`. The `base-level` is the first level in the tower—the application level that the tower eventually interprets.

Within the tower, the form of a level is:

```
(defstruct level
  tower
  call-record-stack
  template-closure)
```

The `tower` component allows anything that has access to the level to access the meta-evaluator (and hence the meta-tower) of the level. This is one of several cycles of reference within the tower, that make it possible to go from one part of a reifier's result to another, as well as being used by the evaluator and meta-evaluator in evaluating the tower. The `call-record-stack` is the succession of saved procedure activations, which are saved as closures, and the `template-closure` is a closure which is copied when a new closure is to be made in that level, before filling in any of its slots with more specific values. For example, it has as its evaluator by default the standard evaluator of that tower.

In each level of the tower, the call stack is made up of closures, each of the

following form:

```
(defstruct closure
  evaluator
  type-evaluators
  language
  procedure-expression
  continuation-expression
  values
  lexical-environment
  dynamic-environment
  original
  level
  number)
```

The `evaluator` closure, the `type-evaluators`, `language` environments, the `procedure-` and `continuation-expression`s, and the `values` and `lexical-` and `dynamic- environment`s components of the closure are as already explained. The `original` is the closure of which this one is an instantiation. It is used by the meta-evaluator to find whether the closure is one that it can interpret directly, as explained in sections **??** and **??**. If the closure is changed by reflection, its `original` field is altered to point to the closure itself, so that, if the closure had been shadowed, it will no longer be recognized as being shadowed (since the shadow will no longer apply). The `level` is the level of which this closure is part. Not only does this give access to other parts of the level, but also through the level it gives access to the tower and thence to the meta-tower. The `number` is there to identify the closure, for the implementor's convenience. It is a reliable way of testing whether two closures are the same. The number is issued from a counter when the closure is created, and when a modified copy of the closure is made, the new one gets a different number.

7.3 The standard evaluator code

The standard evaluator is a procedure which embodies very little evaluation strategy and no language-specific features. Unlike a Lisp evaluator, it does not evaluate the arguments to the procedures that it interprets, as explained in section **??**.

As mentioned in section **??**, the standard evaluator is also the general evaluator. To combine the rôles of tower level evaluator and general evaluator, the evaluator switches on the type of its argument. Tower levels are processed, literals are returned unchanged, variable references are looked up in the environment, and expressions are evaluated by saving the old expression in the context level and recursing to evaluate the context level with the new expression temporarily in place of the old one. The switching is done by looking up the name of the type of the argument, in an environment (the `level-type-evaluators` of

the `closure-level` of the closure) defining how to evaluate each type of object. The values bound in this environment are closures taking as their arguments the thing to evaluate and the level in which to evaluate it. This is very similar to languages, which bind operator names (effectively node sub-type names) to specific evaluator closures.

An earlier version of these routines did not have the `type-evaluators` environment, but used a single routine containing a Lisp `typecase` form, in which a fixed set of evaluand types were handled directly. This made it hard to change specific parts of the evaluation strategy through reflection; the new system is at a better granularity for manipulating parts of the interpreter, just as the "environment of operators" view of languages is easier to handle and extend than a single procedure for handling all types of expression or statement.

Here follows the code of the standard evaluator, written in the dialect of Lisp that is used as the base language of the system (see chapter **??**) with the additional syntax of labels in square brackets for purposes of explanation (also referred to in a later chapter).

The standard evaluator itself

The standard evaluator is a simple procedure which selects other procedures to evaluate its argument according to their type. This means that it has very little evaluation strategy built into it; the strategy for each type of argument is defined in a separate procedure, and these procedures are accessed via the `level-type-evaluators` environment in the level—a form convenient for reifiers to change the evaluation of particular types of value.

Some of the accessor macros in the functions presented here appear to present as part of the level fields which the previous text has explained as belonging to closures. These accessor macros (such as `level-language`) refer to the corresponding part of the closure at the top of that level's call stack.

The defining form `def-unclosure`, which is explained in section **??**, constructs a closure record, but does not close in all of the usual parts of a closure; those which are not defined at this time (such as the dynamic environment) are added to the active copy as the closure is instantiated.

```
(def-unclosure standard-evaluator (anything background-level)
  ;; "The standard standard evaluator."
  (let* ((type-eval
          (lookup (type-of anything)
                  (level-type-evaluators background-level))))
    (if type-eval
        (funcall type-eval anything background-level)
      anything)))
```

The standard evaluator is called with two arguments, the thing to evaluate

and the level which to use as the context for that evaluation.

It finds the type of the evaluand and looks it up in the `level-type-evaluators` environment of that level to produce the closure used to evaluate objects of that type. It then calls this closure. If the closure is not supplied, the evaluand evaluates to itself.

A more flexible alternative to this strategy, used in a further refinement of these routines presented at section **??**, is for each environment to contain, as well as its set of bindings, a value to return for any unbound names. This allows the `if`s and their `else` clauses be removed from the general evaluator and from the expression (list) evaluator, and also makes it easier to specify (and change reflectively) the default actions in these cases. Such flexibility, and fine granularity, are to be desired in reflective interpretation systems. Although the larger number of smaller procedures makes for more overhead in procedure calling, the potentially smaller amount that must be changed to implement evaluation features reflectively means that a larger amount will still be shadowed, so at a small cost to the speed of the fully shadowed system, the overall speed of reflective evaluation is likely to be better than that of the system with coarser granularity and fewer procedure calls.

The following functions are called to evaluate particular types of evaluand. They are bound to the type names by the `level-type-evaluators` environment of the level that uses them. Each one takes the evaluand as its first argument and the level in which to evaluate it as the second argument.

The symbol evaluator

As mentioned in section **??**, local variables are represented not by symbols but by indices into the value list. Symbols are used for type and operator names, and to name non-local variables. It is the non-local variables that are implemented by the symbol evaluator.

The symbol evaluator handles both lexical and dynamic environments; if a symbol is bound dynamically, that binding is used, otherwise the lexical binding is used. (This is easily changed by use of a reflector to re-bind the symbol `symbol` in the `level-type-evaluators` environment of the tower.)

```
(def-unclosure eval-symbol (symbol background-level)
  (cond
   ((eq symbol t) t)
   ((keywordp (the symbol symbol))
    symbol)
   (t
    (let* ((dynamically-found
            (lookup symbol
                    (level-dynamic-environment
                       background-level))))
      (if dynamically-found
          dynamically-found
        (lookup symbol
                (level-lexical-environment
                   background-level)))))))
```

The symbol evaluator has some Lisp-specific code in it, although these features may be used from other languages, and indeed perhaps should, for compatibility. The symbol `t` is treated specially, as are keywords. These two special kinds of symbol always evaluate to themselves.

If the symbol is neither `t` nor a keyword, it is looked up in the dynamic environment, and, if that fails, in the lexical environment, which are stored in the active closure of the level. (The reference to `t` here is Lisp-specific, and is introduced for the implementor's convenience.)

The list evaluator

The list evaluator evaluates expressions consisting of an operator and its arguments (if present). Although its arguments are different, it is similar to Lisp's `apply` procedure. It also implements the implicit funcall used by Lisp and some other languages—a feature which a reifer could remove by rebinding the `list` entry in the `level-type-evaluators` environment of the level. It implements the implicit funcall by tagging `funcall` onto the front of the expression if the operator is not found. The current closure's language's definition of `funcall` will then be used to do the work of the procedure call.

```
(def-unclosure eval-list (list background-level)
  (if (null list)
      nil
    (let* ((operator-name
            (expression-operator list))
           (operator
            (lookup operator-name
                    (level-language background-level))))
      (if operator
[operator] (funcall operator list background-level)
[funcall] (with-changed-level background-level
                (:continuation-expression (cons 'funcall list))
            (standard-evaluator
                 background-level
                 background-level))))))
```

List evaluation is central to the evaluator, as this is where operators and procedures are applied to their arguments.

As with symbols, a Lisp-specific feature appears: by an effect of the Lisp type system, `nil` appears as a list, and always evaluates to itself.

If the list is not `nil`, the operator name of the expression is extracted, at `[operator]`, and is looked up in the language of the level to produce a closure, which is evaluated to implement that operator.

Note that this is very similar to the action of the type lookup and evaluation in `standard-processor` above. The operator may be seen as being the type of the expression.

If the operator is not defined, the funcall operator is used to create a new stack level in which to try to run a function of that name. This is commonly useful, as most languages allow calls to be made without an explicit call operator; it is however to some extent a Lisp-specific feature.

The call to funcall is made by making a temporary modification to the current level with the name `funcall` prepended to the expression, at `[funcall]`. Such temporary changes are made using the construct `with-changed-level` which takes as arguments a level on which to work, a list of changes to make to components of that level, and a block of code to run having made those changes. After running the argument code, it undoes the changes made at the start, and returns the result of the argument code. It is an operator within the tower, and in the meta-evaluator, it is a Lisp macro, which is presented and explained in section **??**. Using the existing level rather than a fresh copy has two advantages: it means that changes made by reflection within the argument code persist outside the dynamic extent of this construct; and superfluous levels are not created (they would add to the load on the garbage collector).

The string-char evaluator

Our implementation uses string characters not as literal characters but as local variable references. (This is because the underlying language, Common Lisp, does not provide for distinctly tagged integer types. Characters were a convenient type to purloin for variable indices.)

```
(def-unclosure eval-stringchar (char background-level)
  (nth-value (char-int char)
             (level-values background-level)))
```

For efficient access to the values list, we use string characters for the indices into it. (This is because in the underlying Lisp system this is the only distinctly tagged small integer type; we want ordinary integers to evaluate to themselves, as literal constants.)

The local variable references evaluator

String characters suffice for most variable references, but we provide a mechanism for using integers in general for this. The presence of two local variable mechanisms rather than one brings no overhead (apart from the behaviour of the environment mechanism, if that is slower for environments containing more bindings—see section **??**), since it is just another binding in an environment.

```
(def-unclosure eval-lvr (lvr background-level)
  (nth-value
   (local-variable-reference-slot-number lvr)
   (level-values background-level)))
```

Local variable references beyond the range of character numbers are referred to by the number stored in a local variable reference structure. This is very rarely used!

The level evaluator

This is present largely for historical reasons; various parts of the evaluator passed levels around (similar, in some ways, to continuation-passing style [**?**]), but most of these are now work in other ways. The evaluator is retained as a way of splicing two levels together and evaluating the result. It is still used in calling external interpreters, where levels do meet in such a manner.

```
(def-unclosure eval-level (arglevel background-level)
  (standard-evaluator
   (level-current-expression arglevel)
   arglevel))
```

A level is split into two parts to make it appear in the appropriate form to pass back to the central evaluator routine.

The closure evaluator

The closure evaluator constructs a level using the expression and language of the argument closure (which should be kept together) and the other components from the level providing the context.

```
(def-unclosure eval-closure (closure background-level)
  (with-changed-level
   background-level
   (:evaluator (closure-evaluator closure)
     :language (closure-language closure)
     :procedure-expression
           (closure-procedure-expression closure)
     :continuation-expression
           (closure-continuation-expression closure))

   (call-meta-evaluator
        background-level)))
```

The other evaluators just pass the arguments straight through:

```
(def-unclosure eval-string (string background-level)
  string)

(def-unclosure eval-integer (integer background-level)
  integer)

(def-unclosure eval-nil (the-nil background-level)
  the-nil)
```

A closure is evaluated by splicing it into the current level, preserving the parts of the closure that must be kept together; the expression and the language must match because the expression is written in the language.

7.4 The essence of interpretation

The functions `standard-evaluator` and `eval-list`, roughly equivalent to Lisp's `eval` and `apply`, are very similar in structure, and a symbol evaluator (shown here for a slightly simpler system) can be constructed that is very similar to these two.

By extracting the common parts of the procedures, It is possible to construct a procedure which performs any of their functions, being passed as arguments suitable structure accessor procedure as well as the arguments for any of the procedures above.

Using this function to create the general evaluator yields the following code:

```
(def-unclosure standard-evaluator (thing level)
  ;; "The standard evaluator."
  (boojum thing level 'type-of 'level-type-evaluators))
```

`standard-evaluator` is the general evaluator, which selects an evaluator according to the type of its argument.

If called with an evaluand of a type for which there is no evaluator defined, the evaluand will typically evaluate to itself, the level-type-evaluators environments returning the identity function as the `unbound` value.

The in-line insides of the previous form of the evaluator have been replaced by a call to a function very similar to the old in-line code, but parameterized to allow them to be used in other ways in the evaluator. For reasons that may become clearer later, this function is called the *boojum* function:

```
(def-unclosure boojum (thing level selector env-selector)
  ;; "Parameterized evaluator kernel for mixed languages."
  (funcall (lookup (funcall selector thing)
                   (funcall env-selector level))
           thing level))
```

The arguments `thing` and `level` are as for the evaluator presented earlier in this chapter. The two extra arguments are structure or attribute accessor functions: `selector` selects an attribute of the thing to evaluate, which is used as the key for finding a more specific evaluator; and `env-selector` finds a part of the level in which to use that key for lookup.

Unlike the evaluator presented earlier, there is no explicit default case handling (for example, for types of evaluand for which no evaluator is defined). This is provided by having the `lookup` function return a suitable default value on being called with an argument that is not bound in that environment. This is a property of the environment used, and requires the environment system to allow the `unbound` value for an environment to be specified as part of the environment.

`boojum` is called in the following ways to make up parts of the evaluator:

```
(def-unclosure eval-list (list level)
  (boojum list level 'car 'level-language))
```

`eval-list` evaluates parse-tree nodes consisting of an operator with a (possibly empty) list of argument sub-expressions. The `unbound` value in level-language environments should be the `funcall` function—which can look much like `eval-list` above:

```
(def-unclosure op-funcall (list level)
  (boojum 'funcall level 'identity 'level-environment))
```

The symbol evaluator is defined without using `boojum`, although it could use it:

```
(def-unclosure eval-symbol (symbol level)
  (lookup symbol (level-environment level))
  ;; could have been:
  ;; (boojum symbol level 'identity 'level-environment)
  )
```

This only allows for one environment, rather than the dynamic and lexical arguments of the evaluator above. However, they could be provided by binding each name to a function that must be evaluated to return a value (functional representation of environments, mentioned in section **??**, is sometimes used in experimental Lisp systems[**?**]), and having a the `unbound` value in the first environment return a value that makes a call to the lookup in the second environment. These functions may, of course, be based on calls to `boojum`.

The rest of the functions do not have any need to call `boojum`, and are just as in the other version of the evaluator.

In all the evaluator code using `boojum` above, only the following functions are used:

- `funcall`
- `lookup`
- `type-of`
- `level-type-evaluators`
- `car`
- `level-language`
- `identity`

- `level-environment`
- `aref`
- `level-value-list`

Most of these are structure accessors, so if we regard those as being just two operators (`type-of` being distinct from normal structure accessors), this reduces it to

- `funcall`
- `lookup`
- `type-of`
- `structure-access`
- `identity`
- `aref`

which is a total of six distinct operators, of which two (`funcall` and `lookup`) could be fairly complicated, and the rest are very simple.

It is my contention that the `boojum` function represents a refined principle of interpretation, going (in terms more from philosophy than from Lisp) from a value of some kind to a description of how to find the meaning of values of that kind, and applying that meaning function to the value, in context, to reach the meaning. It is, in fact, a form of the essence of sentence, which, on further reflection, may be also the sentence of essence.

A specialized form of `boojum`, `snark`, is presented in section **??**.

7.5 Evaluation strategies

Because the evaluator is parameterized by the rest of the closure of which it is the evaluator, the code presented in section **??** is all that the structure of our reflective system requires of the evaluator (other evaluators may add features such as tracing, but the skeletal function is the same). The evaluator does not, by itself, do recursive descent of the expression tree. Sub-expressions are passed to the evaluator (generally, but not necessarily, the same evaluator) by the operator definitions when they need their arguments evaluated. Note that, unlike Lisp, we do not evaluate functions' arguments for them automatically. The evaluation is more like Lisp's special form evaluation. This is because, while for function definition it is normally desirable to have the arguments evaluated, here we are defining language constructs, and will usually require more control over evaluation. Conditionals are the classic example of a construct with such requirements. An alternative to this evaluation strategy is to quote argument sub-expressions, with quotes that are not stripped by evaluation—the *handles* of 3-Lisp [**?**]. These two strategies are equivalent, since the stripping of

handles is done by explicit evaluation. For convenience, Lisp-like evaluation for defined functions may be introduced by a suitable function-defining macro (see section **??**, page **??**), which inserts the argument-evaluation code (see section **??**) at the appropriate point.

7.6 The evaluator as a link between levels

The evaluator links the parts of a level, since it is the part that calls the other parts. The linking is as much a protocol between the components as an active part of the computational mechanism. Since the evaluator of one level runs at the level above, it also determines the protocol for communication of information and control up and down the tower.

It would be possible to have levels of a different form in a tower, so long as they support the same protocol. In this case, the only requirement on each interpreter level is that it must provide an evaluator that can be called to interpret the level. For consistent support of reification and reflection, each tower level should use the standard format of closure in full.

Relaxing the rules to require only the maintainance of interpretability along the tower allows greater diversity. For example, a level written in a language difficult to represent with a parse tree and an environment of operators could use some other form of closure, as long as that closure has an evaluator, and can process the level below.

Since such non-standard closures do not allow the level above them to provide reification data of the commonly expected type (that is, a normal closure), they spoil any assumptions about the universal applicability of the operations we provide for the handling of closures. The usual form of closure appears to be flexible enough, and we will refer to that form only from here on.

7.7 How Lisp-like operators evaluate their arguments

The operators that always evaluate all their arguments (just as Lisp functions do) may do so by calling a general expression evaluator provided in the system, that evaluates all the argument sub-expressions of a given expression—that is, all those other than the operator.

To evaluate each argument, the evaluator of the current level is called. This is done through the form `evaluate-by-evaluator-of-level`, which in practice allows a short cut to be taken in common cases of the evaluation. This form is presented at section **??**, page **??**.

The argument expression evaluatoris provided as an operator in the base language in Platypus. For efficient execution, it is shadowed by a corresponding procedure in the meta-evaluator. Its code is as follows:

```
(defun eval-sub-exprs (expr level)
  "Evaluate each sub-expr (except the first) of EXPR in
the context of LEVEL. The result is a list."
  (let* ((results nil))
    (dolist (x (expression-tail expr))
            (push
[eval]       (evaluate-by-evaluator-of-level
              x level)
             results))
    (nreverse (the list results))))
```

This may be called by operators which require all of their arguments evaluated in left-to-right or unspecified order. For example, arithmetic operators typically will do this. The actual evaluation of each expression occurs at [eval], which evaluates a sub-expression in the context of the current level, using the standard evaluator, which is assumed to be available as an operator (or a function substituting for an operator) in the level. The results of the evaluation are pushed onto a consed stack, and this is reversed to make the overall result, which is a list, leftmost sub-expression's result first, suitable for use as the last argument to a Lisp `apply` call.

Operators requiring explicit control of sub-expression evaluation may call the same function used by [eval] directly:

```
(defun op-if (expr level)
   (if (evaluate-by-evaluator-of-level
            (nth 1 expr) level)
      (evaluate-by-evaluator-of-level
           (nth 2 expr) level)
     (evaluate-by-evaluator-of-level
          (nth 3 expr) level)))
```

7.8 Summary of the standard evaluator

The evaluator is the kingpin of a level. It links the parts of the level to each other by using them to evaluate the level, and links adjacent tower levels by making it possible to shift data from one level to another. Its form is tied to the form of the tower level type.

Each evaluable infinite tower (in the scope of this thesis) eventually reaches a repetitive stage (termed the *boring* stage by [**?**]), the procedure running in each of these identical levels being known as the *standard evaluator* level.

The standard evaluator is a fairly small and skeletal procedure, needing, in its most refined form, only six operators in its definition, most of those being for structure handling. The rest of the evaluator is defined separately, partly in individual operator definitions, and partly in some general evaluator procedures

that may be called by operator definitions. These general procedures are used for evaluating arguments to operators. To do this, they invoke the evaluator and language mechanism to do the evaluation in the appropriate context.

The concise form of the standard evaluator may be seen as a distillation of the essential matter of a programming language interpreter (independently of any particular language). This may be refined further to a procedure which implements several of the main parts of the evaluator. This procedure consists of three procedure calls and one environment lookup.

Who is as a wise man? and who knoweth the interpretation of a thing?

Ecclesiastes 8:1

Chapter 8

Mapping linguistic and semantic features onto the reflective system

Having produced a model of computation on a reflective theme, we now examine its use in modelling existing computing systems and thereby in re-implementing existing languages with the addition of reflective features and mixed-language working.

Our reflective core system is concerned mainly with program structure and control, and makes no attempt to cover such facilities as arithmetic. These must be assumed to be available at ground level. We can build on such ground level facilities, but do not attempt to describe them, instead leaving them to be described by conventional mathematical means (for arithmetic) for example. Likewise, input and output are not covered by our system, but must be made available in some form from the ground level. These facilities tend to be similar in most languages; they depend not so much on the language as on the host computer system. However, input could be seen as a form of reflection, putting into the system data from outside it, and output as a form of reification, copying data from inside the system for things outside it to use. If such communication is done with another such system, it may be seen as a form of co-tower relationship, as mentioned in section **??**.

As well as implementing the immediately visible constructs of a language, reflective operators may implement and control the infrastructure of a language system, such as the binding of values to symbols.

8.1 Procedure application

We take *procedure* (or *function*) *application* as the most fundamental facility to describe. Like Lisp, we can claim a distant connection between our procedure

(closure) application and lambda reduction.

Although, as explained in section **??**, our procedure application does not implicitly evaluate arguments, it is in many ways similar to that of Lisp or Scheme. As well as using closures as procedures we can use them as continuations [**?**] and as meta-continuations [**?**]. A meta-continuation is to a tower as a continuation is to a stack. It represents the frozen state of a tower, much as a continuation represents the frozen state of a process. We will use the term meta-continuation both for the continuation of a simple tower, and for that of a meta-tower of any dimensionality.

The result returned by a reifier is always either a meta-continuation or part of one. Because meta-continuations are values of an ordinary structured type, programs can manipulate (or create) them just as they do any other data; this manipulation is not reflection, and has no effect on the live state of the tower. Reflection occurs when a meta-continuation is explicitly made into the new tower state. By calling and passing meta-continuations, co-routine-like towers (co-towers) may be implemented. Indeed, any form of flow control may be done by reifying, modifying and reflecting meta-continuations, although this is not necessarily the best or cleanest way of implementing features; if it maps poorly onto the implemented language's own model of computation, it may be quite obscure, and not make it at all easy to handle the reified data.

8.2 Reflective and non-reflective calls

There are two classes of procedure call, between which it is important to distinguish:

- the *non-reflective call*, which adds an activation record to the stack within the current level
- the *reflective call*, which adds a level to the tower

In this way, the tower is the call stack for reflective calls.

In implementing Lisp on such a reflective substrate, non-reflective calls can be used to implement Lisp's procedure calls, while reflective calls can be used to implement Lisp's special forms. This is covered in detail in [**?**] and [**?**].

It is natural, in writing programs in the base language of such a system, to make all calls reflectively, so that each procedure called can be reified as a full tower level in its own right, with all the information being available directly. Also, intuitively this seems right; it makes the power of reflective calls available at every call, as well as organizing the arguments into a regular form, in which the argument passed is always the overall state. The callee then extracts the information it needs from agreed places in its single argument, which for a traditional call means from the top of the value list.

For the design of new languages, the distinction between reflective and non-reflective calls seems artificial, and the provision of non-reflective calls unnecessary, when reflective calls are always available. Using just the one form of

call throughout makes the tower into a simpler structure than is possible with two types of call. Unfortunately, it is not possible to abandon non-reflective calling and still implement most existing languages smoothly, because their designs typically assume that everything exists at the same level; while this would not prevent our reflective model from being used for interpretation, it would make the results of reifiers be a poor match to the language's own model of computation, because extra tower levels would appear that would not map onto any level shift apparent in the source language. Since one of the aims of this development is to allow programs to extend the languages in which they are written, this is worth avoiding.

Procedure calls may done implicitly by the standard interpreter, using the `funcall` operator, if an operator definition is not found, as explained in section **??**. When this happens, a level is constructed which will run the *funcall* operator on the procedure to be called. The default funcall action (which this operator is expected to provide) should evaluate the arguments to the procedure, thus making the normal calling by call-by-value. Call-by-name is nearer to the reflective calls mentioned above, as the information structure of the argument will not have been collapsed down into its simplest form, but will be in its original form, which names the value.

If this automatic evaluation of arguments is required, it must be provided by the routines that read programs from the input files and convert them into parse trees. It is very simple to embed each procedure in the user program in a standard piece of code which calls a procedure which evaluates each element of the calling expression, putting the results onto the value list in turn, before calling the procedure body.

The procedure calling mechanism in our system is in general based on that of Lisp, while also being designed with other languages in mind. It is a fairly primitive operation: features such as choosing different procedure bodies according to the form of the argument list (useful for ML, for example) are not provided; they must be moved into the bodies of the procedures that need them. This implies that some re-arranging may be required to go from source code in such languages to our common parse-tree form of procedure bodies. While this may make the reified forms of procedures less natural in some cases, it also makes them common, so that a program in any language can understand data representing reified programs from any other language. With a suitable choice of operators, it should usually be possible to find a reasonable compromise; for example, in ML, procedure bodies might always have as their top-level operator something that chooses which part of the procedure to run, according to the arguments list, and have as the sub-expressions of that top-level expression an alternation of guard clauses and original definition bodies (a bit like Lisp's COND construct). Worse mismatches than this occur with non-deterministic languages, as covered in section **??**.

Open-stack languages are easily accomodated, as the design of the stack shares the local data section between caller and callee—a little like *register windowing* in CPUs such as the SPARC. For languages which view the local data as a stack, the top item is what other languages regard as the first argument.

This conveniently allows procedures with a variable number of arguments to use the earlier arguments to decide how many arguments to use. It does, however, mean that such things as Lisps's `&rest` and `&optional` argument control must always be implemented with the help of the caller, so that the extent of the arguments to a particular call is always known; it is not possible to say "the rest of them" or "all of them" where a local stack is shared without distinction between all levels of procedure call; some form of marker must be used.

This is a problem which requires further investigation, as it is undesirable for a caller to have to know whether a particular callee requires a marker at the end of its argument list. Probably the best solution is for the marker to be used always, in which case the `funcall` procedure and any equivalents (such as `apply`) must always supply it (which is no trouble) and all procedures must understand it in receiving their argument lists. It is then appropriate that they leave it there, to mark the end of the result list—some languages, including Common Lisp [**?**,] and PostScript [**?**,] allow multiple results—for the `funcall` procedure to remove when the callee has returned. Non-local control transfers (`throw`, `longjump`) must also be able to work with these markers.

8.3 Flow control

In a language in which all calls are by value, it is not possible to implement a flow control construct using a procedure written in that language (at least, at that level of interpretation) except by quoting its arguments (in Lisp terminology)—in which case the new construct is not on a par with those initially built into the language, which require no quotation.

The few traditional language designs and implementations that have addressed this problem have used two techniques: *macros*, and *call-by-name* (*fexpr*s, in Lisp). In both of these techniques, the body of the construct is not evaluated before being passed to the procedure, but is passed as a piece of program text. A macro transforms that text to another text, which is used in its place, while a fexpr performs directly the action denoted to it by the text.

In many languages, macros are defined in a part of the language that is separate from the rest of the language; the `#define` construct of C is a good example—it may even be handled by a program separate from the C compiler.

These macro languages are unusual in having no interpretation semantics of their own—they are semantic parasites on their host languages. This separation may be taken as a form of mixed-language programming.

Lisp, however, uses its main language for *macro expander functions*; to tailor further its suitability for this, it has provided a feature specifically to help with this, the *back-quote*.

Conditionals

The implementation of conditional operators requires conditional operators in the language used for the implementation, and so conditional operators of some

kind must be present in the ground interpreter. The ground-level conditionals may be described in conventional terms:

$$(\text{if } abc) \equiv \lambda(abc).b \text{ when } a = \text{true} \tag{8.1}$$

$$(\text{if } abc) \equiv \lambda(abc).c \text{ when } a = \text{false} \tag{8.2}$$

but the aim of our system is not to make these attempts at absolute statements but to describe each level in terms of another level. So instead of the conventional description above, we use our intensional description:

in which the fundamental conditionality is transferred to another level, and thence infinitely far away. Thus, it, like input, and random numbers, is a feature that we can use, but cannot generate. So, to use conditionality, a closure must be linked with integrity over conditionality to a grounded interpreter that provides conditionality (a conditionality termed `ef` (`e`tensional `if`) by Smith).

Since we want the standard ground interpreter to be very simple (so that it is a minimum pinning-down to Turing-computable reality) we will make it provide only one form of the conditional:

if a
  then b
  else c

and other levels and languages can build their own conditional forms from this as needed, such as Lisp's `cond`.

Iteration

Iteration, involving destructive assignment to state, is often regarded as undesirable in language descriptions, and so represented with cleaner tail-recursion. We can use tail-recursion, or we can represent it as side-effects. Since in representing it as side-effects we must use an iteration construct in the level above, we present a paradox by saying that at one level it is iteration by side effects, but at the next level it could just as well be represented indirectly by tail-recursion.

In the terms introduced in section **??** and described in [**?**] and in section **??**, if we represent iteration as iteration, we use jumpy continuations, whereas if we make each loop into a tail-recursive procedure without converting the tail-calling to tail-jumping, we can use pushy continuations. Since a pushy continuation with tail-reflection removed is equivalent to a jumpy continuation (see section **??**), for normal program interpretation this does not make much difference: programs will behave the same way.

In practice, converting iteration into tail-recursion may make reflection into loop control variables more obscure, because it separates loop bodies into subprocedures (*in-line lambdas*, in Lisp terminology). In practice, real iteration in

the interpreter is a closer match to the semantics of many target languages, and so we will generally use that. Also, it may be confusing to find the wrong number of call levels on the control stack, although programs handling reified program data (perhaps through a library) must be able to understand the equivalence between tail-recursion and iteration.

Jumps and continuations; non-local exits

Using the grand jumpy reflector (the reifier that returns a simple snapshot of the whole of the current state and code—see sections **??** and **??**) that assigns the entire state of the system from an arbitrary value that an arbitrary level has produced, we can implement any kind of flow control structure. For example, a `GOTO` may be implemented by capturing the state, changing the continuation expression of the current closure (see section **??**) and making that modified state the new state. A non-local exit may do likewise, but also take some stack frames away entirely before changing the expression of the current continuation closure of the last remaining one. A reflective system permits another form of non-local exit in which several levels of the tower may be spanned. This exit may carry with it, to the catcher of the exit, information from further into the tower or meta-tower, thus being both an exit and a reifier. This can be used to pass back to a program errors in the interpretation of that program.

8.4 Variables, bindings and assignments

Assignments are straightforward, since we can change any part of the state of the system through reflectors. To make an assignment to a variable, we simply write into the appropriate slot in the structure holding the current level's data. Each level's data is stored in the values-list and environments of the closure representing that level.

Local variables are referred to by indices into the value list (from the growable end of it) and non-local variables by symbols which must be looked up in the environment (which could be a hash table, for example). Binding a local variable means making a new local variable to hold values referred to by that name, while binding a non-local variable means storing its old value in a saved bindings chain associated with the environment's lookup table.

Variable access in conventional languages is often compiled in-line, and is perhaps a good example of how levels could be merged ('mixed' [**?**] [**?**]) to produce programs with the reflection partly compiled out—variable access, although here defined in terms of reflection, is an operation that can be done non-reflectively, and so is a candidate for compiling out on a suitable combined reflective-and-plain system, as suggested in section **??**.

8.5 Types

On the whole, we have assumed that a dynamic typing system (see chapter **??**) is available for use throughout the system, and that it is provided by the ground level, and is consistent throughout the system.

Static typing is, of course, possible; the values are still tagged as normal, but a particular language's input parser may have checked the types as it builds the parse tree which we execute.

It is, in general, desirable to keep the same type system for all levels (for ease of passing data between levels at reflection and reification) but it is possible for a level to implement special typing requirements for the level below it. which is acceptable except in first-dimension level shifts, where it makes the underlying implementation repeatedly translate data as it moves it between levels—work which is best avoided when much data must be passed between levels.

In a shift in the second dimension, that takes us to the system that implements non-reflectively the first tower, it may be commonly useful, and desirable, to change the type system to help with modelling the representation scheme used in the first tower.

The reason for this difference is in the meaning of each kind of level shift. The usefulness of changing the type system is for implementing the type system of another tower, whereas within a tower it is useful to have the same type system throughout, to make it simpler to pass data between levels.

8.6 Objects and messages, and actors

Reification and reflection are useful in object-oriented systems, so that an object processing a message can invoke further methods by sending other messages to its *self* (that is, to the same object). This self-reference is a form of reflection. In the language structure represented here, we can model each object or actor by a closure, which closes the programmed actions (methods) of the object with the environment within which it originated. When the actor receives a message, the closure has its `expression` modified to contain the message, and is then evaluated.

8.7 Parallelism

Languages with parallel evaluation constructs [**?**] may be implemented using our model of interpretation, with operators that use jumpy reflectors to switch evaluation contexts. For example, to implement Occam's [**?**]`par` construct, a `par` operator could be written to create new contexts (threads of execution) and put them onto the list of contexts being executed, and the operators that rendezvous between threads (input, the `?` operator, and output, the `!` operator) would perform the context switching required; `?` would transfer control to the thread from which the input will come, and `!` would transfer control to the thread which will receive the output.

Our evaluation framework has not been designed with genuine parallel evaluation in mind, but it could be added in the same way that it can to any other serial execution system.

8.8 Declarative languages

Declarative languages do not map well to the model of interpretation that we use here. One way of implementing them on it is to *procedurize* the programs, so that they run in a manner more like that of procedural languages [**?**] [**?**]. For example, the clauses

```
fish(A) :- chondroicthyes(A);
fish(A) :- osteoicthyes(A);
```

procedurizes to make

```
(defun fishp(A)
  (or (chondroicthyesp a)
      (osteoicthyesp a)))
```

However, with the need for backtracking and cuts, the translation is more complicated than this—one approach is to use Continuation Passing Style [**?**] [**?**] [**?**]. Another is to write an *evaluator* that implements them in any convenient manner, providing suitable procedurized operators in its *language*—it need not use these operators for execution (it would have alternative means for doing the real evaluation), but should provide them to allow program analysis and compilation. Closures in this form should still support the protocol required for tower levels, so that they can be used as evaluators, and can be interpreted by ordinary evaluators. There are further problems arising from the nondeterminism commonly found in declarative languages. It is in principle possible to match this with the rest of the system by writing a special version of the evaluator, which handles collections of results (non-deterministic results) as though they were single values, and makes some kind of translation when transferring data in out of this world. However, this is almost certainly of very limited usefulness—for example, what should the `+` operator do when asked to add two such non-deterministic values (possibly with each of them having a different number of possible deterministic values)? A better solution is probably to provide extensions to such languages to improve their interface with conventional languages, for example using streams [**?**, Chapter 21] or generators [**?**] or some other form of lazy evaluation to work through successive possible results.

Another approach requires specialized libraries for programs that want to call across this divide—bearing in mind the specialized application that such programs are likely to have, this may well be appropriate (or at least unobtrusive) but it is not the purely transparent cross-language calling that we generally

hope to provide.

One way in which Platypus does facilitate cross-calling between unification languages (a subset of declarative languages) and procedural languages is that the environment of the continuation closure may be used as the unification environment, and the value list is a suitable common form of argument/result list. When a procedure in a unification language calls one in a procedural language, the instantiated unification variables are available to the procedural procedure as normal dynamic variables, and when a procedural language procedure calls a unification one, bindings made by the procedural code appear as instantiated variables to the unification code.

8.9 Changing the implementation of features dynamically

As well as being useful for the implementation of language features, reflective programming techniques may be used to alter the implementation as the program runs. As an example of this, I implemented a procedure to evaluate a form using deep binding instead of the semi-shallow binding that Platypus normally uses. There were three parts to this change: new implementations of the binding and assigning operators (`let` and `setq` in the Lisp-like base language); a new implementation of the type evaluator for symbols (variable names); and an adaptor function to evaluate a form given as its argument, with conversion of environments between the standard representation and the new one happening before and after the evaluation itself. Some of the code used is presented in the following paragraphs.

Before reading this re-implementation of variable access, it may be instructive to consider how much effort would be involved in making the same change to a non-reflective evaluator, or to a reflective evaluator that does not use the `evaluator`, `type-evaluators` and `language` structure used in these experiments.

First, the new lookup and binding procedures that are reflected in. A lookup procedure, using association lists, is defined as follows, to be inserted by reflection. (The definition is in two parts, to avoid duplicating code. The first routine allows for environments to be represented either by association lists or by the original form of environments, to avoid having to scan the entire meta-tower system below the current point to convert all environments. It would be more typical of our reflective system to use an environment instead of a typecase, but that would make it more complicated to set up than is warranted for a presented example.)

```
(defun new-lookup (item env)
    (typecase env
         (environment
            (lookup item env))
         (list
            (cdr (assoc item env)))))
```

This routine is analogous to the normal symbol lookup routine presented in section **??**:

```
(defun new-symbol-lookup (symbol background-level)
  (cond
   ((eq symbol t) t)
   ((keywordp (the symbol symbol))
    symbol)
   (t
    (let ((dynamically-found
           (new-lookup symbol
                   (level-dynamic-environment
                          background-level))))
      (if dynamically-found
          dynamically-found
        (new-lookup symbol
                (level-lexical-environment
                       background-level)))))))
```

and a binding procedure, also using association lists, is defined:

```
(defun new-bind (name environment value)
  (cons (cons name value) environment))
```

These are reflected in using something that calls the following procedure. First, it takes apart the data structures used, and saves the old values in variables:

```
(defun eval-with-typevals-and-ops (level
                                   typeval-bindings
                                   opbindings
                                   form)
  (let ((the-closure (level-current-closure level))
        (the-old-language (closure-language the-closure))
        (the-old-type-evaluators
             (closure-type-evaluators the-closure))
        (the-language (copy-language the-old-language))
        (the-type-evaluators
             (copy-language the-old-type-evaluators)))
    (set-closure-language the-closure the-language)
```

then it adds new bindings in the `language` and `type-evaluators` environments:

```
(dolist (this-binding opbindings)
        (let ((opname (car this-binding))
              (opform (cdr this-binding)))
          (bind opname the-language opform)))
(dolist (this-binding typeval-bindings)
        (let ((tyname (car this-binding))
              (tyform (cdr this-binding)))
          (bind tyname the-type-evaluators tyform)))
```

After that, it evaluates its argument form, and restores the old values of the `language` and `type-evaluator` environments:

```
(let ((result (eval-in-level form level)))
  (set-closure-language
       the-closure
       the-old-language)
  (set-closure-type-evaluators
       the-closure
       the-old-type-evaluators)
  result)))
```

This provides a general facility for any reflective changes that modify the handling of a particular type of sub-expression or other value; these changes often require both alterations to the evaluator and to existing operators, and also possibly the addition of some new operators.

To use this for changing the implementation of non-local variables, it is called from the following procedure:

```
(defun eval-with-alternate-bindings-by-type-evals (form)
  (progn
    (eval-with-typevals-and-ops
     (current-level)
     '( (symbol . new-symbol-lookup) )
     '( (lookup . new-lookup) )
     (cons 'eval-with-alternate-environment-representation
           (cons form nil)))))
```

This constructs a form to evaluate its original argument form using `'eval-with-alternate-environment-representation`, which converts the environments between the two representations:

```
(defun eval-with-alternate-environment-representation (form)
  (progn
    (let ((closure (level-current-closure (current-level)))
          (old-lex-env (closure-lexical-environment closure))
          (new-lex-env (environment-to-alist old-lex-env))
          (old-dyn-env (closure-fluid-environment closure))
          (new-dyn-env (environment-to-alist old-dyn-env)))
      (set-closure-lexical-environment closure new-lex-env)
      (set-closure-fluid-environment closure new-dyn-env)
      (let ((res (eval form)))
        (set-closure-lexical-environment closure old-lex-env)
        (set-closure-fluid-environment closure old-dyn-env)
        res))))
```

8.10 Evaluation order—strict or lazy

In Platypus, evaluation order is largely defined by the operator definitions, which call the evaluator to evaluate each argument at the appropriate point. It is, in principle, possible to write an evaluator that normally returns lazy results, probably as closures to be evaluated later (or on a separate computer, to provide parallelism; see [**?**], [**?**], [**?**], [**?**]) and which evaluates things only when necessary. However, because Platypus's model of evaluation allows side-effects anywhere (including the very general side-effect of jumpy reflection, as described in section **??**), this does not in itself provide a universal means for lazifying any existing language! It could, however, be used to implement languages known not to allow side-effects.

It is possible to get round the problem of implementing languages with side-effects, using the power and flexibility of an interpretive system based upon metacontinuations. This may be done by returning lazy results (futures) but keeping a list of them (that is, of futures that need further evaluation to become proper values) in a variable of the evaluator of the level above that containing the lazified program. Whenever a side-effect is about to occur (the evaluator must monitor for this, by catching all use of primitive (shadowed) operators that can produce side-effects) all the stored lazy results are evaluated further toward their real results, and any that finish their evaluation can be dropped from this list. This way, consistency is guaranteed when side-effects happen, although the system is otherwise lazy.

If a lazy evaluator were installed at some level in the tower, all levels below that would become lazy, as no evaluation would occur there until needed. This is an example of how changes at one level may be pervasive through to all lower levels (which have no control over the matter—see section **??**).

8.11 Special substrate features for specific languages

Some languages, particularly those that are primarily part of a special-purpose system, and only secondarily programming languages (for example, expansion languages for programmable text editors, and graphics languages such as PostScript, require special features in their implementations that would not be present in more general-purpose languages. For example, an editor language typically will have one or more buffers in which to hold text, each with a current point, and PostScript has a current graphics state (transformation, colour, linewidth, etc), and usually an underlying imaging system for painting onto a bitmap.

What is the most appropriate place to store such special substrate data? It is not in named variables at the application program level—the namespace must be kept clear for the application program to use in its entirity—but neither may it make the generic evaluator become specialized. The solution is to put it at the evaluator's level, using variables named by agreement between the operators concerned (such as editor-specific or graphics-specific operators)—observe in the code at section **??** that the core evaluator routines do not use any non-local variables at their own level.

Difficult substrate features

Some languages require features that are difficult to integrate with the rest of the system—for example, PostScript's access permission flags which are a part of each of its [compound?] values. Theoretically, this could be implemented tidily by changing the substrate on which the evaluator runs, so that the evaluator works in a world with the relevant underlying type system, but this does not seem very practical. For most purposes, such incompatibilities seem quite minor, and often it may be simplest to tolerate them—for example, to say that all values always allow reading and writing. (In the trial implementation of PostScript in Platypus, we cheat, through good chance, on PostScript's executable and non-executable (literal) arrays, by representing them as arrays and expressions (Lisp lists) respectively—thus mapping a non-standard distinction onto a standard one not otherwise made by that language.)

The need for such substrate features is often connected with the need for special types of value. For such types, it may be appropriate to compile procedures for handling their values, as described in section **??**.

8.12 Summary of building languages with reflection

Our mixed-language interpretation is designed to allow many languages to be built on top of it. Languages which can be converted readily to a procedural

form are most suitable for this: procedural and functional languages are easiest, declarative and rule-based languages are harder.

We assume handling of such types as numbers to be made available underneath the implementation of reflection. Reflection does not help to describe these, anyway, so nothing would be gained were it possible to include them in the reflective system.

Most conventional language features map readily onto a reflective mixed-language architecture. Occasionally there is a mismatch, such as it being natural to try to make all calls reflective (which builds a tower level for each procedure call).

Using jumpy reflectors (that assign to parts of the state, without saving the old values on a stack) to change specific parts of a tower allows very natural implementation of many common language features such as jumps, calls and assignments.

Reflective features may be used to group together parts of a system, such as all the operators of a language, for interpretation in a particular way.

As well as any languages implemented on top of the reflective system, there is a base language which provides reflective facilities and some simple flow control and calling operators. This is sufficient for running the rest of the system, so long as all parts of the system are connected with integrity to the base language.

Reflection allows new features to be added to conventional languages, including extreme examples such as a non-local exit that goes right out of several levels of interpretation.

Procedure calls are to some extent built into the evaluator, but other features are not so much so. Our procedure calling is naturally call-by-name, but call-by-value may be implemented easily on top of this; such a facility is provided in a form that is useful to many language implementations.

"*Must* a name mean something?" Alice asked doubtfully.
"Of course it must" Humpty Dumpty said with a short laugh.

Through the Looking Glass, chapter 6

Chapter 9

A suitable language for the standard interpreter

9.1 Requirements

What are the requirements for a language for the standard interpreter? Within the requirements, we can devise a variety of possible languages, but we will aim for something that is simple, and expressive for a wide range of programs, with particular emphasis on interpreters.

Computational completeness

The first requirement is that the language must be able to express all Turing-computable functions. This is usually satisfied through common-sense in designing the language. It is possible to devise computationally complete languages that are poorly expressive, for example the Turing Machine [**?**]. However, we will provide a range of operators more expressive than the Turing Machine (and working on a basic type system which supports mapping of abstract types to it rather better than does the Turing Machine Tape). Here are some kinds of operators that we expect to find in a serious programming language (and even in a small example language):

- binding
- function application
- recursion
- conditional execution
- operations on objects that are specific to types of particular types (such as arithmetic), in particular: comparison for equality and ordering; addition, disjunction; multiplication, conjunction;

- optionally: iteration (which may be done with recursion and conditionals)
- assignment
- jumps
- stack manipulation

Provision of operators of all these kinds is sufficient for the implementation of a wide variety of programming languages, through interpreters written in a reasonably expressive style. Support for algorithmic and functional styles is particularly good, and implementation of other styles of language, while not so succinct, is not particularly cumbersome. Declarative and pattern-matching languages are the worst fit to our model, and require pre-processing into a suitable procedure-based representation, as explained in section **??**.

I have presented the base language as a form of Lisp, partly because, having simple syntax, it is a convenient notation for simple new procedural or functional languages, and partly because the semantics of the base language, and the operators available, are generally Lisp-like. This follows from Lisp's origins and its evolution toward language and other symbolic processing.

Reflectiveness

Our other major requirement for the base language is that it must support reification and reflection. This needs only two operators:

- `grand-jumpy-reifier` returns the tower's state
- `grand-jumpy-reflector` sets the tower's state

An implementation for each of these is given in section **??**.

Other reifiers and reflectors can be built on top of these, as explained in section **??**, through use of the type system and the function application operator, for example the main operations on meta-continuations as described (under different names) in section **??**:

- `grand-pushy-reifier` runs a function with the tower's state as its argument
- `grand-pushy-reflector` runs a function in a tower constructed from its argument.

Completeness of operators

As mentioned in section **??**, the language for a closure must provide all the operators used in the expression of that closure. This is no new requirement introduced by the tower. The tower of levels with languages has only made it possible for this condition not to be satisfied. In a conventional language, all the operators in the language are always there. Only in a system where the language can be reflected into can this condition be broken.

Therefore, the base language must include at least all the operators needed to implement the standard evaluator. This is an extension of the idea of structural integrity, but applied to interpretation, rather than to the handling, of reified values. To be useful, it should also include a variety of operators typically useful in implementing operators of other languages, such as structure field accessors for the data types returned by reifiers.

It is not necessary for all operators of the base language to be shadowed by the meta-evaluator, but in practice (for efficiency) they all are. (Operators not included in the base language may also be shadowed. The only link between inclusion in the base language and being shadowed is that a computationally complete subset of the base language, and enough of the reflective operators to reify and reflect the entire tower state, must be shadowed.) [**?**] contains a description of how to derive a minimal set of grounded operators.

9.2 The implementation of the base language

Like those in other languages, many of the operators in the base language will need to evaluate all their arguments, but do not need to specify in what order to evaluate them. Arithmetic operators are a good example of this. In the explanation below, we will use

```
((lambda (a b c) (+ a b c)) 1 2 3)
```

as an example.

The most concise way to implement such a family of operators is to split each operator into three parts:

- *An argument evaluator*, `evaluate-sub-expressions` (described in section **??**), which evaluates each sub-expression of the expression being interpreted for which the original operator was used. In the example above, this will take the `a b c` of `(+ a b c)` as the list of sub-expressions to evaluate. Since the lambda construct makes local bindings, `a`, `b` and `c` are local variables, that is, indices into the value list. The value list contains `1 2 3` in the positions `a`, `b` and `c`, and so the result of the sub-expression evaluation is `(1 2 3)`.

- *Primitive operators* such as `prim+2`, capable of doing the basic action on objects of the relevant type, but not of evaluating their arguments.

- *The operators themselves*, such as `+`. These first call `evaluate-sub-expressions`, and then map their corresponding primitive operator along the result of this, and return the result accumulated at the end of the mapping. Thus, `+` will calculate (through iteration, here unrolled for explanation) `(prim+2 (prim+2 1 2) 3)` as it accumulates its result.

In the experimental implementation *Platypus* (see section **??**) the base language and its shadows are set up by a group of Lisp macros `platypus-defprim`, `platypus-def-control-prim`, `platypus-def-lispy-prim`, and `platypus-def--lispy-expr-prim`, which both define the code to be interpreted within the tower, and name (or even define) the Lisp function to be used as the shadow outside the tower.

Since the operators structured in this way call more rudimentary operators such as `dyadic-add`, these more rudimentary ones may be provided as operators in their own right; they may be used directly for implementing some languages. Since they do not evaluate their arguments, the arguments must be fed to them in a fixed manner—non-evaluation of arguments means they cannot even be given through varying local variable indices. The values to be processed must be placed at the end of the values list—the top of the stack—and the operator called. It removes its arguments from the values list by treating it as a stack and popping them from it, performs its essential action, and puts any results onto the end of the values list by pushing them as onto a stack. This form of calling makes these operators suitable for use directly in a FORTH-like language such as PostScript—this is done in the implementation of PostScript used here.

9.3 The real set of primitive operators

In practice, many more operators than strictly necessary may be provided as primitive (shadowed), for efficiency and to make better use of the richness of the substrate system. The operators provided in Platypus include flow control, evaluation, arithmetic, data structure manipulation, reification, reflection, and input and output.

9.4 Operators for reification and reflection

In this section, we look at adding reflective operators to a language, taking Lisp as our example language. All of this applies to any other language used in our experimental system; Lisp is the most convenient example language.

The same reflective operators may be provided in any language. Furthermore, since we make all languages equivalent and transparently cross-callable, and interpretable by the same interpreter, by having common formats for program, language definition, and state, a program P_a written in language L_a may reify a sub-program P_b (perhaps a library routine called from P_a) written in L_b, and will receive it in the same form as that it would receive the representation of itself from a reifier. The operators used will be different, but if P_a does any analysis of the program, it may use the definitions of the operators of the language L_b (which is built into the closure of P_b) to find what each operator does.

Adding reflective facilities to an existing language

Since Lisp's calls are the same as our non-reflective calls, a common way to add reflective features to a Lisp system is to add a special form that does a *reflective call*, in which a procedure is applied to some otherwise hidden part of the Lisp system (such as the *environment* or the *stack*) along with any other arguments which are given as normal in the calling form.

This gives us reifiers such as

```
call-with-environment(procedure args env)
```

which is called as

```
(call-with-environment procedure arg1 arg2 ... argn)
```

but as a callee has an environment and a list of the original arguments supplied as its arguments, with the interpreter interposing the extra information. In effect, the interpreter executes

```
(call-with-environment procedure arg1 arg2 ... argn)
```

as if it were

```
(apply
       procedure
       *current-environment*
       (list arg1 arg2 ... argn))
```

where `*current-environment*` is a reifying variable—a variable handled specially by the evaluator, holding part of the information used by the evaluator in evaluating the program—holding the current environment.

Here is a sample piece using this style of reifier:

```
(defun print-environment (reified-environment arglist)
  "Print out the environment to a stream.
This expects to be passed two arguments: the environment
of its caller, and the argument list with which its caller
thought it called it. That argument list should have one
element, a stream to which to print."
  (format
     (car arglist)           ; should be a stream
     "The caller's environment is ~S~%"
     reified-environment))



(defun env-to-file (fn)
  "This uses a reifying call to pass its environment
to a function which will print out that environment."
  (with-open-file (str fn :direction :output)
    (call-with-environment print-environment str)))
```

The general case of such a reifier is `call-with-level proc (level args)`, which is called as `(call-with-level proc arg1 arg2 ... argn)` but as callee receives the level at which it is running and the original argument list, as if it were called as: `(call-with-level *current-level* proc args)`.

Since the call frame contains a level, the *call frame* (activation record) of such a reflective call is in effect a tower level, into which information has been reified from the calling level.

When using activation records as tower levels, the link to the lower levels is an ordinary local variable/parameter in the stack frame. The next lower level for an interpreter must be held in variables of the interpreter anyway, even in a non-reflective system, as the interpreter must store somewhere the program it is interpreting. In our experimental implementation, the link to the lower levels is always in the same place in the level: it is in the second argument position in the value list of the interpreter, just after the interpreter's evaluand.

In this form of reflective interpretation, every call (transfer of meaning) to a lower level must pass that lower level to itself as part of its parameter list. This is easily accomplished, as the calling interpreter has that information available for its own use already, although possibly in a different form. All that it needs to do is include the data in the stack frame that it builds, so that it appears as if given as an argument to the call.

For reflection, one of a second range of special forms, complementary to the first one, and characterized by the form `(eval-with-<context> <procedure> <context>)` (where `<context>` is whichever part of the context we wish to reflect), calls a procedure given as one of its arguments, with parts of the context surrounding it being taken from its other arguments. For example, we could have reflective operators such as `(eval-with-environment form environment)`, which evaluates `form` using `environment` to provide any free variables needed by `form`, and `(eval-with-arglistform arglist)` called `funcall` in Lisp, or, to be as general as possible, `(eval-with-level form level)`. For example, this function

```
(defun another-level-cons (d e)
   (eval-with-level
      '(cons d e)
      (construct-funny-level)))
```

returns a cons cell which was constructed by the `cons` operator of a different level.

Although these additions to Lisp can provide full reflective facilities, they present no organized model of non-reflective and reflective calls. Also, they often (although not necessarily) work in terms of the internals of a normal Lisp system, which, meta-circularity notwithstanding, is not particularly suited to manipulating language elements as data values. For example, to represent environments we might conveniently use a-lists, but an abstract type for environment, with appropriate operators (bind, unbind, lookup, assign, boundp) would be more appropriate. This is a consequence of the poor support for data abstraction (that is, just cons cells!) that classical Lisp provides. It seems appropriate to find a model of computation, and a type system to support it, designed more specifically for reflective evaluation, and not built around the facilities of one particular language. Such a system would have a data type representing the state of a computation, in which type grand reifiers would always return their results, and in which grand reflectors would always take their arguments. Each field of this type would also be of a particular type, and these types would be few and of simple well-defined characteristics.

The type we will use to represent the state of a computation is the interpretive closure, as described in chapter **??**, and the types of its fields, as described in chapter **??** are interpretive closure, environment, expression (or parse tree) and values list.

Jumpy and pushy reflection and reification

Instead of the two special calls `eval-with-level` and `call-with-level`, which always start new levels of interpretation, we can use alternative, and simpler, forms of reifier and reflector. In these operators, the state is represented as a tower. The reifier, which is a procedure taking no arguments, returns the state

of the system as its result, and the reflector takes one argument, a tower, which replaces the current tower.

These jumpy operators differ from the reflective operations described in section **??** in that they *assign* to the state rather than *bind* the state. The pushy reflective operators may be implemented on top of the jumpy ones with the addition of a stack.

The grand pushy reflector may be implemented as the following macro, which takes something to evaluate and a level in which to evaluate it, and performs the evaluation in that context. The variable `*tower*` belongs to the level of the current level's evaluator. The action of reading it is a grand jumpy reifier, and writing it, a grand jumpy reflector.

```
(defmacro eval-with-level (form level)
  `(let ((old-levels (tower-levels *tower*)))
     (setf (tower-levels *tower*)
           (cons ,level old-levels))

     (eval ',form) ; this is implicitly in
                   ; the context of *tower*

     (setf (tower-levels *tower*)
           old-levels)))
```

A matching reifier is implemented by the following macro, which takes a procedure to call and an argument list, and splices onto the start of the argument list the level within which it will call that procedure:

```
(defmacro call-with-level (function &rest args)
  `(funcall ,function
            (car (tower-levels *tower*))
            ,@args))
```

As demonstrated by the procedures above, when used on a conventional architecture (that does not provide stack pushing as a primitive) these assigning (jumpy) reflective operations are more primitive than the binding (pushy) ones, in the sense that they may be used as part of the internals of the pushy operators. However, given an interpreter outside the tower, the pushy operators may also be implemented as primitively, in that they are simply functions that produce or consume an extra argument before calling a given function, and where the meta-evaluator handles that argument. (However, on a conventional computer system, all reflective operators are actually jumpy at the exact point of reflection, as there is a point when control switches from one context to another, regardless of whether the old context has been pushed onto some kind of stack, or otherwise stored.)

Each of these two approaches to handling reified execution state uses one of

two kinds of meta-continuations, as explained in sections **??** and in [**?**]: *pushy* meta-continuations, which bind the tower state, and *jumpy* meta-continuations, which assign to the state. In activating a pushy continuation, no information is lost, as the old state is kept on a stack of continuations; while a jumpy continuation discards information, simply replacing the old state with the new one. These terms may be used to describe both control flow within one level (where pushy continuations are procedures to be called, and jumpy continuations are labels to be jumped to) and flow of meaning between levels, where pushy and jumpy meta-continuations are the two forms of reflection that we have just described.

Pushy and jumpy meta-continuations may be mixed within one tower, and even within one level. Viewed as tower-constructing devices, they have rather different forms. A jumpy meta-continuation destroys tower levels, by replacing a string of them destructively with a shorter string, or creates them, by replacing one level with a string of several levels.

A pushy meta-continuation cannot destroy or create levels in the same tower in this sense, but starts a new tower, orthogonal to the first. (This is as described in section **??**.) The analogy for this in (non-meta) continuations is that a jumpy continuation modifies a sequence of actions within a procedure, while a pushy continuation starts a fresh sequence in another procedure. Just as tail-recursive pushy procedure continuations can be transformed into iterative continuations, tail-recursive pushy meta-continuations can be transformed into iterative metacontinuations. Functions for these transformations are given in [**?**].

From here on, we will use more systematic naming for reflective operators. The reifier called `eval-with-level` above will now be called `grand-pushy-reifier`, and the reflector `call-with-level` will be called `grand-pushy-reflector`. The jumpy reflective operations, not given as named procedures above, but used implicitly in-line, will be called `grand-jumpy-reifier` and `grand-jumpy-reflector`. Here is the code for the two grand jumpy reflective operators:

```
(proclaim '(special *tower*))

(defun grand-jumpy-reifier ()
  *tower*)

(defun grand-jumpy-reflector (new-tower)
  (setq *tower* new-tower))
```

Reifying and reflecting specific parts of a level

The grand jumpy or pushy reifier and reflector are the only reflective operations that we need. However, most of our reflective actions will copy the current tower, make some change, and make the changed tower become the current one.

To make this kind of activity more convenient, and also to avoid unnecessary work, we provide some more specialized jumpy reflectors, for changing individual components of the current tower.

Changing one part of a level's state usually maps to one operation in a typical programming language. For example, alterations to the value list can implement assignment to variables or binding of variables; changing the continuation expression implements a jump or a call. Assignment to the interpreter has no equivalent in conventional programming languages, although there are commonly statements to make new procedure definitions (usually statically) and to implement non-local flow control (long jumps).

Providing separate reflectors for each part of the state stored in the closure thereby models cleanly the actions that a typical interpreter must implement. We also provide a collection of procedures for handling the reified values. These not only make handling these values easier, but also help to maintain consistency in the tower—not a theoretical consideration, but helpful in avoiding an easy way of crashing the system.

The reflectors (and, less importantly, reifiers) which affect only part of the state are as follows:

- `set_procedure_evaluator` and `procedure_evaluator`
- `set_current_evaluator` and `current_evaluator`
- `set_procedure_language` and `procedure_language`
- `set_current_language` and `current_language`
- `set_procedure_expression` and `procedure_expression`
- `set_current_expression` and `current_expression`
- `set_procedure_values` and `procedure_values`
- `set_current_values` and `current_values`
- `set_lexical_environment` and `lexical_environment`
- `set_dynamic_environment` and `dynamic_environment`

The reifiers above may be built on top of the `grand-jumpy-reifier`, and select a field of the the resulting record. The reflectors are different: they must find the part of the structure that the corresponding reifier would return, and modify just that part. In terms of conventional programming language technology, this is finding the left hand side value (abstract address) of the reifier's result and then through it assigning the new right hand side value.

This is more efficient than finding the whole reified object by `grand-jumpy-reifier`, changing just one part and reflecting the whole thing back in with `grand-jumpy-reflector`. In some forms of Lisp [**?**, section 7.2], efficient code for these operations can be generated automatically through the use of `defsetf`, producing an interface presented as `setf` forms.

Integrity

A requirement met implicitly by providing a set of whole-tower reflective operators is that reflective integrity should be preserved in going through a level that uses this language. A procedure shifted up to execute into such a level can always return information back to its home level.Therefore, information may always be passed up and down the tower by reflecting procedures to run in different levels as long as the language at each level has at least the facilities of the minimal base language. This requirement must be met anyway for other reasons: were a level not to include facilities equivalent to those of the base language, it could not form part of a integral string of interpreters (this is a circular argument) and so the integrity of the tower would be lost at that point. This would make it into two orthogonal towers, as described in section **??**. However, it would still have structural integrity for reification (as described later in this section), and so lower levels would be able to access levels which they are not interpreted by at all. Thus, such a tower would no longer be grounded, as its connection with the umbrella interpreter would no longer have integrity—it would be two towers for purposes of interpretation, but only one for reflection.

Structural integrity for reification, that is, for access to remote levels through reifiers in one's own level, is ensured through the structure of each level, rather than through the language.

Integrity of groundedness through this level is met through the computational completeness of the language. It is grounded because it can, in its own right, compute anything that is computable. Its groundedness does not depend on that of any other levels.

Inserting and removing levels of evaluation

The most important of the reflected data manipulation procedures are one to insert a new evaluator just above the base of the tower, and a complementary one to remove an evaluator from just above the base of the tower.

Since an evaluator is an ordinary closure, it is guaranteed that when inserted above an existing, interpretable (grounded) evaluator, it can be interpreted by the previous first evaluator, and also that it can interpret the old application.

Here are the procedures for inserting and deleting evaluators from the end of the tower:

```
(defun add-evaluator (evaluator)
  (let* ((our-tower (grand-jumpy-reifier))
         (new-level
          (make-evaluator-interpretation-level
           (car (level-call-record-stack our-tower))
           evaluator)))
    (push new-level
        (level-call-record-stack our-tower))
    (grand-jumpy-reflector our-tower)))

(defun remove-evaluator ()
  (pop (level-call-record-stack (grand-jumpy-reifier))))
```

An operation on the tower that preserves its integrity is one that replaces a string of levels by another string of levels that also has integrity. For the string to have integrity each interpreter must be able to interpret its lower neighbour, and so the new sequence must fit the interpreters above and below it correctly:

which we can guarantee by taking those end interfaces, labelled *Algol language* and *Algol program* entirely into the operation, replacing not only the link between them but those levels themselves. This way, we never try to link levels which will be incompatible, but can always insert an extra level in between as a buffer, with the appropriate language definitions for the previous evaluator. (The ability to do this depends on an evaluator being provided written in the new language.) Unfortunately, this may add more levels of interpretation, and so perhaps should be avoided in practice, for efficiency.

In practice, common tower manipulations change only one level, adding or removing an interpreter between two that remain unchanged:

an operation which is referentially transparent to all lower levels. Although the intensional meaning has been changed, the extensional meaning is still the same, because a transformation that preserves integrity is one that does not destroy the correctness of the previous interpretation.

An example of the usefulness of this is adding and removing tracing of evaluation, by adding and removing an evaluator that traces what is being processed.

Handling collections of closures

To change *en masse* how a group of closures (such as those implementing the operators of a language) are interpreted, the closures may be grouped by giving them the same evaluator. This makes them share a tower level together. Then, all that group of closures can have their processing affected by changing that one evaluator.

For example, all the operator closures of the language of a closure could be given the same evaluator, and then all activity in that language could be traced by tracing that evaluator. (To trace an evaluator i_1 which is interpreted by an evaluator i_2, we insert an evaluator i_t, which traces what it processes, between i_1 and i_2.)

Grouping closures in this manner can cause harmless dimensional anomalies in the tower structure. The closures that have been grouped share a tower level because they have the same evaluator; yet they may also be at distinct tower levels for other reasons, such as one being part of the interpreter of another. Thus, an evaluator may exist at more than one level, and so the same tower can have more than one number of levels between the base and the umbrella, as shown in the following diagram:

The same reification and reflection operators may be made into a part of any language; their action is always the same. Some languages will require some amount of packaging around the bare reflective operators, as they may not provide means for handling such data directly (they might have to present it in terms of a procedure library for handling reified data), or they may prefer to present the data in some form that matches the language's natural model of computation more closely.

Unrolling the infinite tower

As mentioned in sections **??** and **??**, the boring section of the tower is kept as a circular reference, which could in principle be followed indefinitely. If, however, we were to allow a program to do so via the reifiers, it would be hard to detect how many levels of the tower it had climbed before eventually changing something. So, to simplify the tracking of realized tower levels, we make the shadow versions of the the operators reifying and manipulating tower components to do some extra work that is invisible from within the tower (unless the program within the tower calls for reification of the meta-tower, as described in section **??**. These are the extra actions needed:

- Operators for reifying structures, and for accessing parts of structures, that return a value from higher in the tower than their caller (for reifiers) or their argument (for structure field accessors), and for which the result type may be `closure`, compare the value found with the standard evaluator (see sections **??** and **??**. If the value is the same (`eq` in Lisp terminology), a copy (one level deep, not a tree copy) of it is returned instead, otherwise the value is returned itself. The closure is also lookup up in the tower's shadow maps, and if it is found to be shadowed, a copy is returned likewise. In the copy, the `original` field points to the closure of which it is a copy (see section **??**), and so the meta-evaluator recognizes that this is still part of the boring section of the tower, and so shadows its evaluation. This copying ensures that the standard evaluator and the shadowed operators

can never be changed—it is never actually held in any place reachable from any variable of any program within the tower.

- Operators for modifying fields of closures set the `original` field of the closure to the closure itself, so that it is no longer recognizable as a shadowed closure if it originally was one, and when the meta-evaluator evaluates the closure, it will do so by interpretation instead of by shadowing.

  The code used for doing these actions is part of the meta-evaluator, and so is presented in the meta-evaluator chapter, in section **??**, page **??**.

Reflective operators that reach the substrate

There are a few operators that reflect down (or up!) to the substrate language. The main one of these is `eval-in-cl`, which takes an argument form that it passes to the evaluator of the substrate language, which is the `eval` procedure of Common Lisp. The result of the evaluation is then passed back as the result of this Platypus operator. This operator was provided so that the benchmark suite for Platypus could also run the same tests in Common Lisp automatically, for comparisons of the speeds. It could also be used for a form of reflection, right through to the substrate, to ask for compilation of Lisp forms that could then be installed in the shadow map to make new primitives. (This is discussed further in section **??**.)

There is also a `break` operator, for use as a breakpoint, that runs a read-eval-print loop in the substrate language. When the user quits from the loop, possibly having reflected some changes into the system after reifying and displaying some information, this operator returns.

The `time-now` and input-output procedures are also in some sense substrate system reflective operations.

9.5 Summary of the base language

A language for use with the standard interpreter in the boring section of the tower must be powerful enough to support both the standard interpreter and the procedures that will run on it, which will typically be operators for other languages.

The implementation of the base language has two parts: the operators themselves, and their shadows, which are run at the next meta-level in the tower. (The last meta-tower is run in the substrate language on which the whole reflective system is built, and it is there that all operator definitions are eventually evaluated.)

The language should provide operations typically needed by interpreters, and those needed for reification and reflection. It is also desirable that the base language be reasonably expressive.

As well as the fundamental reifiers and reflectors, it is convenient to provide some jumpy reflectors that assign only part of the state; these are not only more

efficient, but also more expressive of many common language features that they may be used to model.

In practice, we provide many more operators in the base language than are strictly necessary. ([**?**] explains how to work out which operators are necessary.)

Summary of reflective operators

Operators for reflection may be added to an existing language. With our model for mixed-language interpretation, the same operators will work for any language.

Reflective operators (reifiers and reflectors) are of two kinds, *jumpy* which move data between program-as-agent and program-as-subject without automatically creating new levels of interpretation, and *pushy* which create new levels either providing data from the program-as-subject or using it to create a new (or modified) program-as-subject. Jumpy operators are more primitive than pushy operators, in that (on a conventional architecture) they may be used to implement pushy operators, whereas, within one level of interpretation, pushy operators may not be used as the primitive on which jumpy operators may be built (other than by considerable wasted work).

One form of reflective operator is the *grand reflective operator* which reifies or reflects the entire state of the system. However, it is more efficient, as well as often more convenient, to reflect into just the part of the state required, and so reflectors that set only specific parts of the state are also worth providing in a practical system.

Reification of programs is homogenous between languages. The same reifiers (and reflectors) may be used in any language, and the values returned have closed into them all the linguistic information needed to understand the value in any way that might be required.

A fool hath no delight in understanding, but that his heart may discover itself.

Proverbs 18:2

Chapter 10

A model meta-evaluator

Having already described the standard interpreter we now look at a model for the meta-evaluator. The meta-evaluator stands in for an infinite string of meta-circular standard interpreter levels, and in so doing implements the entire first dimension of the tower. To do so, whenever it is about to call a procedure—whether that is the evaluator, a type-evaluator or an operator—which is shadowed, it must call the shadow procedure (which runs in the meta-evaluator's own level, but performs actions as though it were in the meta-evaluatee's level) instead of the original procedure.

10.1 Requirements for the meta-evaluator

There are two major requirements for the meta-evaluator:

- it must be able to do everything that the standard interpreter can do;
- it must support level-shifting within its own level, in such a way that it can realize on demand levels for use in the tower that it implements.

Since we are not requiring it to conform to the structure of a tower level itself, we can observe that

- it may be a whole interpreter (evaluator and language in one);
- it must provide operations on the fundamental types on which the standard interpreter's type system is based.

Most of the operations of the meta-evaluator will be directly equivalent to those of the standard interpreter, and the forms of the two will be similar. Reflective operations do not translate so directly into operations of the meta-evaluator because they do level-shifting, which is not possible when there is only one level. The operations which cause level-shifts in the standard interpreter must translate to non-reflective actions in the meta-evaluator. All these actions affect variables of the meta-evaluator, which represent the state of the tower.

The meta-evaluator has one important variable, let us call it `*tower*`, whose value is the tower which it interprets. This is the argument with which the meta-evaluator was called.

A non-reflective operation in the tower works on the contents of the base of the tower. For example, arithmetic operators handle values in the `values` element of the current closure at the lowest level of the tower, and flow control operators affect the stack of sub-functions (sub-expressions) implicit in the `expressions` elements of the stack of closures at that level.

Reflective operations may change the contents of the tower more significantly. For example, the `grand-jumpy-reflector` operation sets `*tower*` to a fresh value that has been produced from somewhere within the (old) contents of `*tower*`. Likewise, `grand-jumpy-reifier` assigns to a location (such as the value part of a variable binding) stored within the current value of `*tower*`. These may be used, in conjunction with a stack in the meta-evaluator, to implement pushy reflection.

This description would suffice were the meta-evaluator to be positioned at the end of the tower. A *lightning bolt* [**?**] is needed to reach from the top of the tower (where the ground is) toward the base of the tower. This must cut through the infinite repetitive (boring) section of the tower:

However, as well as being at the infinitely remote upper end of the tower, the meta-evaluator has to be alongside the whole tower, so that it can reach through the infinite section of the tower, to shadow things at the lowest level at which it is possible:

The lightning here travels along the tower through the meta-evaluator, which can conduct it from one level to another as needed. As already discussed, it strikes the highest point on the tower that is not mimicked by the meta-evaluator; before this, it is just travelling through open space, and does not touch anything.

Because of the meta-evaluator being beside the tower, its connection to the tower is more complicated. Indeed the tower is more complicated than that, as it has a castle of towers arising from the operator closures in the language of each level, thus:

where, at any one time, exactly one of the towers drawn small on the diagram—the one starting at the closure for the current operator—is active.

The following diagram shows that the meta-evaluator is alongside everything, and that it contacts (shadows) any tower in the castle wherever a tower in that tree reaches meta-circular standard interpretation:

Here, the meta-evaluator brings the lightning's power to any part of the tower, whether in the main tower or in a side tower.

10.2 An introduction to Platypus

Platypus is a non-reflective, single-language implementation of a reflective, mixed-language interpreter. (Its name originally stood for Practical Language Algebra with TYpes, Procedures, Unification and Sending, although the latter two features are not implemented in this version of it.)

There are now three implementations of Platypus: an early prototype written in Cambridge Lisp, a version in C written for speed, and a more readable and concise version in Common Lisp. Both of the latter two are referred to below. Example code throughout the thesis is drawn largely from the Common Lisp implementation, *Platypus89*.

Platypus is a shadowing interpreter, as described in sections **??** and **??**, and has to implement:

- all the operators needed in the standard interpreter that it mimics;
- level-shifting, including the creation of new levels on demand;
- auxiliary functions, such as storage management, input and output.

10.3 The pull of efficiency

My thesis is that the reflective facilities described can be implemented efficiently enough (compared against a conventional system) that a programming system using them is useable in practice, at least for prototyping. Hence the intensively used parts of Platypus are written with efficiency in mind. It is also essential that everything required in the reflective system is provided correctly—correctness cannot be compromised to gain speed. Therefore, there is a division of Platypus' internals into two parts:

- those that are mapped onto by the standard interpreter: these are visible and hence have to fit exactly to the abstract model of interpretation used by Platypus;
- those that are private to the meta-evaluator: these are not visible to programs running on the system, and so can be implemented with any programming techniques that will provide enough speed.

This division is particularly noticeable in the C version. In *Platypus89*, the equivalent procedures inside and outside the tower could be produced from the same source, with one branch being held as a lambda-expression inside the tower, and the other compiled to machine code by the substrate Lisp system. (In practice, the sources are similar but not identical for most of the code, for historical reasons. Some of the compiled code is generated mechanically from the non-compiled code definitions, particularly for Lisp-like functions that evaluate all of their arguments.)

10.4 Representation strategies

There are two ways in which we could store tower components:

- in exactly the form in which they are seen through reifiers;
- in a form convenient to the meta-evaluator, which must be converted to and from the form seen within the tower.

In all versions of Platypus, to be sure of the correct representation of reified state to the application, we use the former approach: everything is pre-reified—only ever kept in its reified form—rather than reified on demand. This means we that have to write the meta-evaluator to handle all the reifiable structures in their reified form throughout. The only values which are not always kept in a reified form are those that have no reified form, such as the internals of file stream descriptors.

In practice, it turns out that this works well; representations suitable for use with reified data are also suitable for use by the evaluator and meta-evaluator, and there are no points at which some other representation would have been obviously more suitable.

However, were translation at reification and reflection to be used, it would occur at very specific points (the reifiers and reflectors). Much of it would be simple, although there are possibilities for complicated reifiers and reflectors, ranging from building of reified stack frames to de-compilation at reification and compilation at reflection [**?**]. Because of the translation occuring at such well-defined places, the possibilities for errors introduced by this feature are restricted, and the possible effects on the rest of the system are reduced.

10.5 Mapping

Objects in the tower world that have special meaning to the meta-evaluator, such as the closures representing functions which the meta-evaluator evaluates directly itself, must be mapped onto the corresponding objects in the meta-evaluator. In the C implementation of Platypus, this mapping is done by a pair of *hash tables*, using a perfect hashing scheme—that is, hash table lookups take a constant time. This is made possible by special allocation of memory while initializing the system (see section **??**), such that there is something special about the addresses of the heap objects that have special meanings to the meta-evaluator.

The more significant of the two hash tables in the C implementation, as far as reflective interpretation is concerned, is the *operator map*. This maps from objects inside the tower to functions in the meta-evaluator. If the object is a closure representing a primitive operator, that is, one which is shadowed by the meta-evaluator, and is being called from the standard evaluator, this mapping succeeds and returns the meta-evaluator function. If the object is not a primitive operator closure, it is mapped to a distinguished null value.

The other hash table, called the *location map*, is concerned with garbage collection. It is used to update C variables pointing to distinguished heap objects (such as nil) as the garbage collector moves things around.

In the Common Lisp implementation, Platypus89, the hashing for the operator map is done by normal Lisp hash-tables. There is no location map in this implementation, as we use Lisp's garbage collector, and all levels share one conceptual undivided address space (and type space!), and so our meta-evaluator variables pointing into the tower are updated automatically as part of the normal action of Lisp's garbage collector.

The meaning of the map

As mentioned in section **??**, the operator map is in some ways analagous to a language. Just as a language maps operator names (which are symbols, or tokens) to operator closures, the operator map maps operator closures at one meta-level to those at another.

Now, since the meaning of a name is not inherent in the name, but only in the context (environment) in which the name is de-referenced, any kind of value (such as an operator closure) may be used as a name. Thus, both operator maps and languages translate operator names (operator extensions) of some kind to operator closures (operator intensions) so that the intension of the operator may be used to fulfil, in that context, the extension of the operator.

10.6 How the meta-evaluator works

Meta-circular calls to the standard interpreter are recognized with a single comparison operation. When the meta-evaluator begins to evaluate a level, it compares the evaluator of that level with the standard evaluator.

- If they are the same, the meta-evaluator evaluates the level itself;
- If they are different, the meta-evaluator builds a new real tower level in which to run the evaluator of the level with the level as the argument to the evaluator. (The new level was already there in principle, but was not realized in memory). The meta-evaluator then recurses to evaluate this level.

When the meta-evaluator is interpreting a level directly, each time it tries to apply an operator or type-evaluator, on behalf of a standard interpreter level which it is shadowing, it first looks it up in the appropriate map.

- If it finds the null value (that is, the operator or type-evaluator is not mapped), it climbs the tower, by realizing a new level of the tower to run the evaluator of that operator or type-evaluator, and recursing to interpret it;
- If it finds a non-null value, it calls the shadow function (in the language of the implementation) denoted by that value, with the current level as

its argument. The whole tower is accessible through this level, since each level has its evaluator built into its structure, and holds its evaluee, if any, in the argument list of its current closure.

This is almost identical to the action of the standard interpreter within the tower, except that there is a conditional level-shift in a one-level-deep second tower dimension, and when this level shift is taken, the operator or type-evaluator shadowing map is used much like a language.

The actions of the C and Lisp implementations are very similar. As far as the ideas of reflective interpretation are concerned, they are equivalent. In the implementations, the main difference between the evaluators is in the mapping mechanism—and this is a small difference, and in either implementation, it could be done some other way. The most notable difference between the substrate systems is that Lisp provides a lot of useful things (such as memory management) that had to be written into the C version directly.

10.7 Summary of the model meta-evaluator

The boring part of each tower is not really evaluated, but its evaluation is mimicked by the *meta-evaluator* of that tower. The meta-evaluator has two rôles: it implements finitely the infinite tower, and it stands in for any number of levels.

To do this, it has to be able to absorb level shifts, to stop them going any further along the tower. It does this by realizing new levels, (and abandoning old ones), on demand, when it must extend the non-boring part of the tower, and shadowing things itself when still on the boring section.

The code of the meta-evaluator can be similar to that of the standard evaluator, with the addition of some level-shifting code that would not, within the tower, be allowed to exist within a single level, because it is capable of generating (realizing) levels[1].

The meta-evaluator is alongside the tower (from the tower's point of view) and both alongside and above the tower (from the meta-evaluator's point of view).

The meta-evaluator runs beside the lowest tower level that it can, that is, one level above the highest one that is not mimicked by the meta-evaluator. It follows this boundary by climbing up to new levels as it realizes them, and climbing back off them when they are no longer needed.

There are two approaches to how the meta-evaluator should view data within the tower, in terms of how each type of data is represented, and whether each type appears as the same type inside and outside the tower, or as distinct types. In this thesis, we hold the data in the same form in both, and hence, for example, stack frames do not have to be re-encoded when reified or reflected.

The meta-evaluator must have a map from *shadowed* operators in the tower to *shadowing* operators in the meta-evaluator. In a system with only one di-

[1] Within the tower, each procedure may be in only one level at a time.

mension to the tower, this map must be visible to the meta-evaluator but not to the tower.

If the meta-evaluator implements the storage system of the tower, it must also have a map from distinguished objects within the tower to variables in the meta-evaluator. If the meta-evaluator and the tower share a storage system, such a map is not needed.

"And part of the roof came off, and ever so much thunder got in—and it went rolling round the room in great lumps—and knocking over the tables and things—till I was so frightened, I couldn't remember my own name!"

The White Queen, Through the Looking Glass, Chapter 9

Chapter 11

The meta-evaluator

In this chapter, we present a meta-evaluator for towers of the form described in this thesis. There are two implementations of the meta-evaluator, the earlier of these two being written in C, and the other in Lisp. They are presented in the other order, however, since the Lisp one is more important, and has superseded the C version.

The Lisp implementation has two versions, the second being derived from the first by moving common code from several procedures into a separate procedure, which is the kernel of this form of the meta-evaluator.

In section **??** we look at *Platypus89*, the Lisp implementation of Platypus, which is written for conciseness, readability and programming aesthetics. A full listing of Platypus89 is given as Appendix A of this thesis.

In section **??**, we look briefly at *C-Platypus*, the C implementation of Platypus. This is written for efficiency. Although this line of development was abandoned in favour of the Lisp-based approach, some of the ideas explored in this version may be useful in future developments of reflective interpreters, and so are still presented.

11.1 Platypus89 - a tower evaluator in Lisp

The meta-evaluator of *Platypus89* is a Lisp function looking much like the standard interpreter. The listing given here is produced directly from the real sources (using a mixed-language style of LaTeX and Lisp!). As with other listings from the real source code, the Lisp syntax has been modified to allow labelling of points in the code with text in square brackets.

The standard meta-evaluator itself

The general tower meta-evaluator function is very similar in structure to the standard evaluator, but it must also perform level shifting when required.

```
(defun evaluate-anything (anything background-level)
  "Evaluate ANYTHING in the context of LEVEL."
  (let* ((cont (level-current-closure background-level))
         (evaluator (closure-evaluator cont)))
[is-on-standard-evaluator?]
    (if (not (eq (closure-original evaluator)
                 (tower-standard-evaluator-closure
                  (level-tower background-level))))
[interpret-evaluator]
        (let ((evaluator-interpretation-level
               (make-interpretation-level
                background-level
                evaluator)))
          (evaluate-anything evaluator-interpretation-level
                             evaluator-interpretation-level))
[evaluator]
      (let ((inside-tower-type-eval
             (env-lookup (type-of anything)
                         (closure-type-evaluators cont))))
        (if inside-tower-type-eval
            (let ((outside-tower-type-eval
[type-shadow-lookup] (env-lookup (closure-original
                                  inside-tower-type-eval)
                                 (tower-type-shadow-map
                                  (level-tower
                                     background-level)))))
              (if outside-tower-type-eval
[shadowed-type]  (funcall
                    outside-tower-type-eval
                    anything background-level cont)
              (let ((type-interpretation-level
                     (make-interpretation-level
                         background-level
                         inside-tower-type-eval)))
                (evaluate-anything
                 type-interpretation-level
                 type-interpretation-level))))
          anything)))))
```

The first decision to make in the meta-evaluation of a level is whether the the level can be interpreted directly by the meta-evaluator, or whether the level's evaluator must be interpreted. The rule for this, as explained in section **??**, is that the level can be interpreted directly if its evaluator is the standard evaluator, and otherwise its evaluator must be interpreted by the meta-evaluator. This decision is taken at `[is-on-standard-evaluator?]`. If the level cannot be evaluated directly, the meta-evaluator recurses to run it, at `[interpret-evaluator]`. This realizes (see section **??**) a level of the tower,

which previously existed abstractly but had no separate representation in the implementation. The realization is done in `make-interpretation-level`, which allocates a new level and fills it in such that when interpreted it will interpret the given evaluator with the original level as the evaluator's argument.

If the level can be evaluated directly, this is done at `[evaluator]`, a point which has a direct equivalent in the standard evaluator. The type of the evaluand is looked up in the `type-evaluators` environment of the level to find a closure to instantiate and apply to evaluate that kind of evaluand. If no way of evaluating it is found, the evaluand is returned unchanged, at `[type-unknown]`. Otherwise, the `original` of the type-evaluator is looked up in the `type-shadow-map` of the tower, at `[type-shadow-lookup]`. As explained in section **??**, the `original` field may be used to determine whether a closure is an instantiation of a particular closure (usually a shadowed one). If a closure is modified (other than in the values list), its `original` field—not accessible to be changed independently—is changed automatically to point to the closure itself, rather than that of which is was an instantiation, and so it will not be treated as an instantiation of that one any longer. If the type evaluator is shadowed, the shadow is called, at `[shadow-type]`. Otherwise the type evaluator must be interpreted, in a modified copy of this levl, made and evaluated at `[non-shadowed-type]`. For a system with a one-dimensioned tower, the shadow procedure is in the substrate language, Common Lisp. It will be one of the procedures presented below.

The list evaluator

The list evaluator is like that within towers (including the all-too-Lisp-specific part for handling `nil`!), but must also look for the shadow of an operator, and, if it has one, use that, otherwise realizes (see section **??**) a new level to interpret the operator definition, and interprets that level using the current evaluator—this is done using the Platypus form (or Common Lisp macro) `evaluate-by-evaluator-of-level` (presented in section **??**) rather than as a direct call, so that this procedure may be called from places other than `evaluate-anything`:

```
(defun meta-eval-list (list background-level curr-cont)
  "Evaluate LIST in the context of BACKGROUND-LEVEL.
CURR-CONT is the current continuation closure, which the caller has
already extracted and so might as well pass along."
  (if (null list)
      nil
    (let* ((operator-name (expression-operator list))
           (operator (env-lookup operator-name
                                 (closure-language curr-cont))))
      (if (null operator)
[funcall]
          (with-changed-level
              background-level
              (:continuation-expression (cons 'funcall list))
            (evaluate-anything background-level
                               background-level))
[operator]
        (let ((shadow-definition
[shadow-lookup] (shadow-lookup
                  operator
                  (level-tower background-level))))
          (if (null shadow-definition)
[non-shadowed-op]
              (let ((operator-interpretation-level
                     (make-interpretation-level
                      background-level
                      operator)))
                (evaluate-anything
                 operator-interpretation-level
                 operator-interpretation-level))
[shadowed-op]
            (funcall shadow-definition
                     list
                     background-level
                     curr-cont)))))))
```

Extraction of the expression, operator and operator definition of the current continuation closure of the level are just as in the standard evaluator, as is the automatic conversion of unknown operators into procedure calls.

Once an operator has been found, the action is similar to that of the standard evaluator, but with the addition of code to handle shadowing of some operators by primitive definitions. The split between shadowed and non-shadowed operators is, in terms of the standard evaluator, entirely within the `(funcall op anything)` at `[operator]`. In principle, the standard evaluator and the meta-evaluator could be produced from the same source code, using an operator/procedure which could be described as `shadowable-funcall`; within the tower, this would be a normal funcall, but in the meta-evaluator, would have

the extra rôle of finding and using the shadow versions of callee procedures that have shadows.

At `[shadow-lookup]`, the shadow map for the tower is looked up. This shadow map must match the tower's meta-evaluator, and describes which operators the meta-evaluator has shadows for. Ideally, it should be a field of the value stored in the meta-evaluator field of the tower.

If there is no shadow for that operator, the meta-evaluator recurses to interpret the interpretable definition of the operator. This is done at `[non-shadowed-op]`.

If a shadow has been found, it is called directly (that is, to run at the level of the meta-evaluator), at `[shadowed-op]`. This is written as a macro partly to give it a more descriptive name, and partly to hide even more optionally compiled tracing code! It is, in the underlying Common Lisp, simply a `funcall`.

The symbol evaluator

Symbol evaluation is very similar—there is no shadowing to do here either!

The code below is simplified slightly from what a real system should do, in that if `nil` is returned by the first lookup, the second lookup is performed. This should really be done using a multiple-value return [**?**, Section 7.9] with the first value being the result of the lookup and the second value being the success code. However, there is another approach to this, using a distinct return value (which the next meta-level might not make available for general use at this level). This is described in section **??**.

Like Common Lisp, our dialect of Lisp provides *keywords*—symbols which always evaluate to themselves—and they are handled at `[keyword]`. (Keywords are printed with a colon preceding the symbol name.) They are generally useful as simple token values; for example, in the list of changes given to `with-changed-level`, they are used to name the parts of the level to change.

```
(defun meta-eval-symbol (symbol background-level curr-cont)
  "Evaluate SYMBOL in BACKGROUND-LEVEL."
  (cond   ((eq symbol t) t)
[keyword] ((keywordp (the symbol symbol))
           symbol)
          (t
           (let
               ((dynamically-found
                 (env-lookup symbol
                             (closure-fluid-environment
                              curr-cont))))
             (if dynamically-found
                 dynamically-found
               (env-lookup symbol
                           (level-lexical-environment
                            background-level)))))))
```

The string-char evaluator

Evaluation of local variable indices (string characters) is just as in the evaluator inside the tower. Since the listings here present the working code of the system, they use for efficiency a Common Lisp feature that does not affect the semantics of the program: the declaration `the`, which allows the compiler to make assumptions about the types of expressions' results. A `fixnum` is a number which is an integer and which need not have heap memory allocated for it—that it is, it is not a double and not a `bignum` (a number stored as a string of digits). (This requires it to be small enough to fit, along with its tag bits, into one machine word.) Telling the compiler explicitly that values are of this type permits the use of efficient number handling code, with, for example, addition instructions that are compiled in-line, instead of tests on the types of the arguments, and calls to a general-purpose addition procedure to deal with any cases that the in-line code cannot handle directly itself. (In addition to this, some parts of the system are compiled with an implementation-specific proclamation, `(fixnum-safety 0)`, which tells the compiler that *all* arithmetic is to be performed with `fixnum` operations (which are typically inlined) rather than calling general arithmetic functions.)

```
(defun meta-eval-stringchar (char background-level curr-cont)
  "Evaluate CHAR in the context of BACKGROUND-LEVEL.
CURR-CONT is the current continuation closure,
which the caller has already extracted and so might
as well pass along."
  (declare (ignore background-level))
  (nth-value (the fixnum (char-int char))
             (closure-values curr-cont)))
```

The local variable references evaluator

Local variable reference structures also involve no shadowing:

```
(defun meta-eval-lvr (lvr background-level curr-cont)
  "Evaluate LVR in BACKGROUND-LEVEL."
  (declare (ignore background-level))
  (nth-value (the fixnum
                  (local-variable-reference-slot-number lvr))
             (closure-values curr-cont)))
```

The level evaluator

Levels are handled exactly as inside the tower (perhaps a little surprising, considering their close involvement with the model of reflective evaluation):

```
(defun meta-eval-level (arglevel background-level curr-cont)
  "Evaluate ARGLEVEL in BACKGROUND-LEVEL."
  (declare (ignore background-level curr-cont))
  (evaluate-anything (level-current-expression arglevel)
                     arglevel))
```

The closure evaluator

The same goes for closure evaluation:

```
(defun meta-eval-closure (closure background-level)
  "Evaluate CLOSURE in BACKGROUND-LEVEL."
  (with-changed-level background-level
      (:evaluator
        (closure-evaluator closure)
        :language
        (closure-language closure)
        :procedure-expression
        (closure-procedure-expression closure)
        :continuation-expression
        (closure-continuation-expression closure))
      (call-meta-evaluator background-level)))
```

The macro `with-changed-level` is explained in section **??**.

The other evaluators just pass the arguments straight through:

```
(defun meta-eval-string (string background-level curr-cont)
  "Evaluate STRING in BACKGROUND-LEVEL."
  string)

(defun meta-eval-integer (integer background-level curr-cont)
  "Evaluate INTEGER in BACKGROUND-LEVEL."
  integer)

(defun meta-eval-nil (the-nil background-level curr-cont)
  "Evaluate THE-NIL in BACKGROUND-LEVEL."
  the-nil)
```

This is all of the significant code of the meta-evaluator. Much of the rest of Platypus89 is made up of the operators and their shadows.

Summary of the meta-evaluator code.

In the small amount of code presented above are all the semantically significant parts of a mixed-language meta-tower-based evaluator—and perhaps more than is strictly necessary for that function. The apparently complex goal turned out to have an essentially simple solution, refined to be just what is required for the generalization of what it is for something to be an evaluator for procedural and functional languages.

Is it possible to refine this further, to derive a function which describes the essence of tower-reflective interpretation in the context of a mixed-language system?

Several parts of this are directly equivalent to parts of the standard evaluator, and the explanations given in section **??** apply precisely to these points too. These include those marked as `[funcall]`, `[operator]`, and `[evaluator]`.

11.2 The essence of reflective tower evaluation

Just as it is possible to refine the standard evaluator to a form in which several parts of the evaluator are calls to a suitably parameterized function, it is also possible to do this with the standard meta-evaluator.

This parameterized function, the *snark*, is this:

```
;;;; Is this the ultimate Lisp function?
(defun snark (thing
              level         selector
              env-selector  shadow-selector
              cont)
  "Parameterized meta-tower evaluator kernel
for mixed languages."
  (funcall (lookup (lookup (funcall selector
                                    thing)
                           (funcall env-selector
                                    level))
                   (funcall shadow-selector
                            level))
           thing level cont))
```

It is very similar to `boojum`, but with a second selection and lookup. It is an optimized form of a `boojum-boojum` function:

```
(defun boojum-boojum (thing
                      level         selector
                      env-selector  meaning-selector
                                    shadow-selector)
  "Parameterized meta-tower evaluator kernel
for mixed languages."
  (funcall (lookup (funcall meaning-selector
                            (lookup (funcall selector
                                             thing)
                                    (funcall env-selector
                                             level)))
                   (funcall shadow-selector
                            level))
           thing level))
```

the optimization being that `meaning-selector` is always the identity function. The `snark` function is used in the following procedures to make the new version of the meta-evaluator. This is a replacement for the code presented in section **??**, but it uses `snark` for each of the appropriate parts of the evaluator, instead of the ad-hoc evaluation functions there shown. It may be instructive to compare this with the code presented on page **??**, to see the similarity between `snark` and the last part of that version of `evaluate-anything`. The general tower meta-evaluator function is very similar in structure to the standard evaluator presented on page **??**,, but it must also perform level shifting when required.

In this case, `snark` is used to do the switching on the argument type, and the level shift is performed conditionally, to get the type evaluator either interpreted or mimicked as appropriate.

```
(defun evaluate-anything (anything background-level)
  "Evaluate ANYTHING in the context of LEVEL."
  (let* ((cont (level-current-closure background-level))
         (evaluator (closure-evaluator cont)))
    (if (not (eq (closure-original evaluator)
                 (tower-standard-evaluator-closure
                           (level-tower background-level))))
        (let ((evaluator-interpretation-level
[non-shadowed-eval] (make-interpretation-level
                       background-level
                     evaluator)))
          (evaluate-anything evaluator-interpretation-level
                             evaluator-interpretation-level))
[shadowed-eval]
      (snark anything
             background-level
             #'type-of
             #'level-type-evaluators-function
             #'level-type-shadow-map-function
             cont))))
```

It would be possible to simplify this evaluator further, with added flexibility, but at the expense of speed, by using another map to go from the evaluator in the tower to its shadow, the meta-evaluator. In such a system, this new map (which replaces the `if` in the code presented above) would map the standard evaluator to a procedure which calls `snark` to switch on the evaluand type, as at `[shadowed-eval]`, and anything else (via the map environment's `unbound` value) to a procedure which makes a new level in which to interpret the evaluation, as at `[non-shadowed-eval]`. The use of the extra map would, naturally, be implemented by another call to `snark`. The further flexibility that this brings is that it would then be possible for more than one possible evaluator to be shadowed by more than one meta-evaluator.

The expression evaluator

The expression evaluator uses `snark` to dispatch on the operator name and to do the level shift, if necessary, to get the operator definition interpreted or mimicked as appropriate.

In this version of the expression evaluator, there is no code to convert unknown operators to function calls. This may be done by having the `unbound` pseudo-binding in the language environment bind unknown operators to a procedure which converts them to function calls and tries again.

```
(defun meta-eval-list (list background-level curr-cont)
  "Evaluate LIST in the context of BACKGROUND-LEVEL.
CURR-CONT is the current continuation closure, which
the caller has already extracted and so might as well
pass along."
  (if (null list)
      nil
       (snark list
              background-level
              #'expression-operator-function
              #'level-language-function
              #'level-operator-shadow-map-function
              curr-cont)))
```

The symbol evaluator could use `snark` with the identity function as the key selector for use on symbols, and have the `unbound` binding in the lexical environment bind to a function that looks the name up in the dynamic environment. Then, as `snark` always evaluates the result of its lookups, this inheritance between the two environments is provided—at the cost of each binding being of a procedure that returns the intended value of the binding—rather like a continuation-passing style of environment implementation.

```
(defun meta-eval-symbol (symbol background-level curr-cont)
  "Evaluate SYMBOL in BACKGROUND-LEVEL."
  (cond   ((eq symbol t) t)
[keyword] ((keywordp (the symbol symbol))
           symbol)
          (t
           (let
               ((dynamically-found
                 (env-lookup symbol
                             (closure-fluid-environment
                                       curr-cont))))
             (if dynamically-found
                 dynamically-found
               (env-lookup symbol
                           (level-lexical-environment
                                       background-level)))))))
```

The remaining functions are as in the other version of the evaluator, as they have no particular use for calling `snark`.

This is all of the significant code of the compact meta-evaluator, occupying only 83 lines of openly formatted Lisp, in 11 procedures—perhaps rather shorter than an ad-hoc versions of such a function would be!

This is shorter than I originally expected such a program to be, and is quite a small part of the overall system. An analysis of the size of the system is presented in section **??**.

11.3 Summary of the meta-evaluator code.

Is it possible to refine this function further? There is one thing that can be done: to pun on the names of structure components, making an isomorphism between the union of all possible evaluands (or their types) and the structural field wherein they exist and are evaluated. Doing this, the key selector and the environment selector take on the same names for each pair. Are there any parallels to this in other programming systems? It is similar in some ways to instances and classes in object-oriented system such as SmallTalk-80 and CLOS.

Structural punning apart, the boojum/snark appears to be the most refined form of evaluator that can be derived along these lines. Seen in more general terms than Computer Science, it does seem to make sense as a description of performing an action according to instructions: use some key aspect of the situation and some key aspect of the context to find an action, and perform that action, possibly moving to the plane of abstraction necessary for handling that action.

How does this fit in with general models for action, for example the deliberation model of Doyle [**?**], as re-presented in [**?**]? This is examined further in chapter **??**.

The data stack representation in the meta-evaluator

In Platypus89, the data stack is represented as a single extensible vector, the *values list*, which is shared between all closures that refer to it. Its division into stack frames is an abstraction visible only in the operators of languages, and a language may treat it as an open stack, as does PostScript, for example.

There are several advantages to this form of stack, which I consider to be a correct choice of stack design here.

- Open-stack (eg FORTH, PostScript) and framed-stack (eg Lisp, Algol) languages can use it interchangeably, and can cross-call transparently using it.
- It is easy to pass a variable number of arguments, although either an end-of-list marker or a count must be passed, unless the extent of the argument-list is implied by the values of some of the arguments. This provides, for example, for Lisp's `&rest`, and for PostScript's `array` operator, and its overloaded transformation operators such as `rotate`.
- It is easy to reify—and the structure is simple to manipulate.
- It is efficient—there are no stack frames to allocate and collect, only a pointer or index that represents the end of the stack.

- Multiple results can be returned, as required for PostScript and Common Lisp.
- The only assumption that it echoes is that there is a conventional call-return stacking—an assumption made throughout the rest of the system anyway.
- There are no "live, but returned-from" stack frames to manage—this stack organization forces operators to store the data off the stack at the appropriate time.

One point that might seem unusual or counter-intuitive about the correct implementation of this kind of stack is that Lisp/Algol-like languages must take their first argument as being the first on the stack–that is, the *highest*-indexed in terms of the stack implementation. This gives compatibility with languages like PostScript, even to the extent of `&rest` working compatibly between them and other languages.

The only drawback with this ordering is that if arguments to a function are to be evaluated left to right, as Common Lisp and many other languages require, the results cannot be pushed onto the stack in the order in which they are evaluated. The solution to this is to access the value list as an indexable array rather than as a stack while doing this. The auxiliary function for funcall, `funcall-helper`, which is presented on page **??** in section **??**, uses the following macro to evaluate the argument sub-expressions from left to right but to push their results right to left:

```
(defmacro eval-and-push-sub-exprs (expr level)
  "Evaluate each sub-expr (except the first) of EXPR in
the context of LEVEL.  If optional extra argument START
is given, that specifies where to start in the sub-expr
list, with 1 being the usual place, that is, it is how
many initial sub-exprs not to evaluate. The results are
pushed onto the stack of LEVEL."
  `(with-values-of-level
    ,level

    (let* ((the-exprs ,expr)
           (the-length (length the-exprs)))

      (when (>= (+ the-length (the-value-list-last))
                (the-value-list-max))
       (extend-value-list (the-value-list) the-length))

      (do* ((x the-exprs (cdr x))
            (n (- the-length) (1+ n)))
           ((endp x))
           (setf (the-nth-value n)
                 (evaluate-by-evaluator-of-level
                  (car x) ,level)))
      (incf (the-value-list-last) the-length))

    nil))
```

It would, in principle, be possible to use the same stack mechanism to implement the stack of saved closures in each level (the return stack, in conventional terms) and also to implement the stack of levels that, along with the shadowing system, makes up each tower. This would reduce the amount of code needed, not only to implement the tower, but also in programs that use reified data.

11.4 Other parts of the meta-evaluator

The central procedure of the meta-evaluator is similar to that of the standard evaluator, and there are other parts that are analogous to some in the standard evaluator. These include the operators, and procedures called by the operators for common purposes such as the evaluation of arguments.

As well as the parts that do have analogues in the standard evaluator, there are those that handle the tower manipulation—constructing levels and shifting between them.

The Platypus operator (or Lisp macro, in the meta-evaluator) `evaluate-by-evaluator-of-level` calls the evaluator of the current level, or, if that evaluator is the standard evaluator, calls the meta-evaluator directly. It is used where procedures called by the evaluator need to call the evaluator. Going via the evaluator of the current closure of the current level is preferable to

calling `standard-evaluator` or `evaluate-anything` directly, as it allows such procedures to be called from places other than the standard evaluator and meta-evaluator. The code of `evaluate-by-evaluator-of-level` is is follows:

```
(defmacro evaluate-by-evaluator-of-level (thing level)
  "Evaluate THING in the tower of LEVEL
using the evaluator of that tower."
  `(let* ((our-level ,level)
          (evaluator (level-evaluator our-level)))
     (with-changed-level
      our-level
      (:continuation-expression ,thing)
      (if (= standard-evaluator-closure-number
             (closure-number evaluator))
          (evaluate-anything ,thing our-level)
        (call-meta-evaluator our-level)))))
```

Creating and using towers and levels

The procedures presented here create and maintain towers and the levels that make up the towers. They are in two groups, those handling whole towers and those handling levels.

Procedures for handling towers

The following procedures set up and start towers. Starting a tower means applying its meta-evaluator to it; starting a tower in the real evaluator (of which these routines are part) means applying something one stage further away than the meta-evaluator, as all levels of meta-tower that support that tower must also be started at the same time.

A tower is a complex data structure with many cycles of references; `new-tower` and `funcall-shadow`, and many of the procedures presented further below, are used in the maintenance of this structure.

Much of the manipulation of the tower is done in the core of the meta-evaluator, but it can also be done by operators, including some that might not normally be regarded as reflective. For example, `funcall-shadow` is unusual in being an operator that manipulates the tower structure explicitly (in the form of adding a call record to the call record chain of a level)—although this *may* be done by any procedure in the system, whether or not it is an operator, and whether or not it is shadowed, as these actions are simply the manipulation of a data structure.

Some structure access operators, for manipulating fields of tower components, may sometimes be in effect reflective, if the components that they alter are parts of a live tower, as reifier results are—reifiers in Platypus do not take a copy (snapshot) of the tower, but return references to parts of the real tower.

When this is the case, operators such as `set-closure-evaluator` are jumpy reflectors. As described in section **??**, the shadow versions of such operators take special actions to ensure that shadowed procedures are never altered—a copy is always taken at the appropriate place and it is that copy that is altered.

Towers are created by the function `new-tower`, which takes as its arguments the meta-evaluator, the base level program, and the standard procedure. The tower that it sets up has the base level already realized, complete with a reference back to the tower. If there are already some evaluator levels attached to the base level, they are retained, and these levels are given references back to the tower just as the base level is.

```
(defun new-tower (meta-eval base stand-proc)
  "Make a new tower, with META-EVAL as its
meta-evaluator, BASE as the program it is to run at its
base level, and STAND-PROC the evaluator it is to use
as its standard evaluator. Any evaluators BASE may
already have are kept, and STAND-PROC is put above them
all."
  (let ((the-tower
         (make-tower
          :meta-evaluator meta-eval
          :base-level base
          :standard-evaluator-closure stand-proc)))
    (do* ((this-one base
                    next-one)
          (next-one (closure-evaluator
                     (level-current-closure
                      this-one))
                    (closure-evaluator
                     (level-current-closure
                      this-one))))
         ((eq next-one stand-proc) nil)
         (when (null next-one)
               (setf next-one
                     stand-proc
                     (closure-evaluator
                      (level-current-closure
                       this-one))
                     stand-proc)))
    the-tower))
```

`new-tower` creates a tower with a given meta-evaluator, application program, and standard evaluator. This tower is suitable for passing to `run-tower`.

```
(defun run-tower (tower)
  "Evaluate TOWER."
  (begin-define-in-tower tower)
  (prog1
      (call-meta-evaluator (tower-base-level tower))
    (end-define-in-tower tower)))
```

`run-tower` is the real substrate language procedure that starts the evaluation of a tower. It sets the definition context (for such things as `defun`) to be that tower, and then passes control to the tower's meta-evaluator, and, after setting the definition context back to what it was, returns the result from the meta-evaluator.

Procedures for handling levels

These macros and procedures are for creating (or realizing—see section **??**) new levels. They are used in the meta-evaluator when it has to do level shifts when a evaluator, type-evaluator or operator must be interpreted.

```
(defmacro make-interpretation-level (level evaluator)
  "Make a level with the context from LEVEL
to evaluate an instance of EVALUATOR with an
argument of LEVEL.  EVALUATOR is a closure."
  `(new-level
    ,level                              ; level
    ,evaluator                          ; closure
    (list ,level ,level)))              ; arguments
```

`new-level` is the central routine for creating new levels ready for parts of the evaluator to shift the evaluation into them. It takes as arguments:

- A `level`, from which a tower is found. The new level is put into the same tower as this argument level. This level also provides the template closure used to fill in parts of the instantiated closure that the level will start running.
- A `closure` to instantiate, providing the expression that the new level is to run. The rest of the instantiated closure (apart from the argument list, that is, the initial value list) is made from the template closure which is part of the level supplied as the first argument.
- A `argument list` which is used to fill in the initial contents of the value list of the current closure of the new level.

`new-level` creates two linked structures, a level and the current closure for that level, and completes the cycle of references between them.

The code of `new-level` is as follows:

```
(defun new-level (level closure args)
  "Construct a level to go in the tower of LEVEL
made by instantiating CLOSURE with ARGS for its
arguments."
  (let* ((tower (level-tower level))
         (template (level-template-closure level))
         (the-new-closure
          (make-closure
           :evaluator
            (closure-evaluator template)
           :language
            (closure-language template)
           :type-evaluators
            (closure-type-evaluators template)
           :procedure-expression
            (closure-procedure-expression closure)
           :continuation-expression
            (closure-continuation-expression closure)
           :values
            (convert-to-value-list args)
           :lexical-environment
            (closure-lexical-environment template)
           :fluid-environment
            (closure-fluid-environment template)
           :level
            nil                 ; filled in later to complete a circle
           :original
            (closure-original closure)
           :number
            (incf closure-counter)))
        (the-new-level
         (make-level
          :call-record-stack
          (list the-new-closure)
          :tower tower
          :template-closure (closure-original closure))))
    (setf (closure-level the-new-closure) ; complete the circle
          the-new-level)
    the-new-level))
```

Modifying levels

The macro `with-changed-level` is used in a variety of ways. It modifies its argument level as specified by a list of changes, evaluates its body forms, and undoes the changes it made to the level, before returning the result of the last of the body forms.

In effect, this is a rebinding of some fields of the level, the scope of the rebinding being the body arguments of this macro. However, the rebinding is

not done within the level being modified, but in the meta-evaluator handling that level—in this case, in the stack of the substrate system.

This is important not only for intuitively correct evaluation of the interpretive tower, but also for its efficient evaluation. It is correct, because some of this work is done behind the scenes by the meta-evaluator, and is hidden from the tower being worked on; it is efficient, because it avoids creating temporary tower levels and throwing them away after use—the original level is re-used, and the temporary storage is on the substrate system's stack.

The coincidence of correctness and efficiency goes further than that, too: were temporary levels to be made for use in the evaluation, changes made to those levels through the use of reflection would be discarded unless copied back into the levels from which the temporary levels were copied.

The disputable point here is not whether this technique should be used, but where the saved data should be stored, in the version that runs on the substrate system. (Obviously enough, when run on an ordinary meta-evaluator, the saved data may be kept on that evaluator's stack.)

The expansion of `with-changed-level` can be quite complex; its macro-expander procedure calls the following procedure, `make-change-savers`, to generate pieces of the code to return. It takes a list of change descriptions, each of which consists of a list containing a keyword indicating what to change, and a form indicating what to change it to. It returns a list of lists, each of which contains (in order):

- the name of a variable in which to save the old value,
- the name of a temporary variable in which to store the new value while other changes may be being calculated—they all take effect simultaneously as far as the evaluand tower is concerned
- the form to evaluate to get the new value,
- and the structure field description of what to change in the level (in a form suitable for passing to Common Lisp's `setf` [**?**, section 7.2]).

The elements of these lists are built into the result of the `with-changed-level` expander.

```
(defun make-change-savers (change-list)
  "Shadow function for expanding with-changed-level."
  (let ((the-savers nil))
    (let ((evaluator-value
           (member :evaluator change-list
                   :test #'eq)))
      (when evaluator-value
            (push
             `(old-evaluator new-evaluator
               ,(second evaluator-value)
               (closure-evaluator the-cont-clo))
             the-savers)))
    (let ((language-value
           (member :language change-list
                   :test #'eq)))
      (when language-value
            (push
             `(old-language new-language
               ,(second language-value)
               (closure-language the-cont-clo))
             the-savers)))



    (let ((type-evaluators-value
           (member :type-evaluators change-list
                   :test #'eq)))
      (when type-evaluators-value
            (push
             `(old-type-evaluators new-type-evaluators
               ,(second type-evaluators-value)
               (closure-type-evaluators the-cont-clo))
             the-savers)))



    (let ((procedure-expression-value
           (member :procedure-expression change-list
                   :test #'eq)))
      (when procedure-expression-value
            (push
             `(old-procedure-expression new-procedure-expression
               ,(second procedure-expression-value)
               (closure-procedure-expression the-cont-clo))
             the-savers)))
```

```
(let ((continuation-expression-value
       (member :continuation-expression change-list :test #'eq)))
  (when continuation-expression-value
        (push
         `(old-continuation-expression new-continuation-expression
           ,(second continuation-expression-value)
           (closure-continuation-expression the-cont-clo))
         the-savers)))



(let ((values-value
       (member :values change-list :test #'eq)))
  (when values-value
        (push
         `(old-values new-values
           ,(second values-value) (closure-values the-cont-clo))
         the-savers)))
(let ((lexical-environment-value
       (member :lexical-environment change-list :test #'eq)))
  (when lexical-environment-value
        (push
         `(old-lexical-environment new-lexical-environment
           ,(second lexical-environment-value)
           (closure-lexical-environment the-cont-clo))
         the-savers)))



(let ((fluid-environment-value
       (member :fluid-environment change-list :test #'eq)))
  (when fluid-environment-value
        (push
         `(old-fluid-environment new-fluid-environment
           ,(second fluid-environment-value)
           (closure-fluid-environment the-cont-clo))
         the-savers)))
the-savers))
```

`with-changed-level` has a macro expansion in which the subject level and the parts involved in the change are bound to variables, using `let*`. Within the body of the `let*`, the modifications are made, the argument body evaluated, and the modifications reversed, before returning the result of the last of the argument body forms.

```
(defmacro with-changed-level (level changes &body body)
  "Modify LEVEL as specified by the keyword arguments in CHANGES, and
run BODY.  Then change the changed parts back."
  (let ((change-savers (make-change-savers changes)))
    `(let* ((the-level ,level)
            (the-cont-clo (car (level-call-record-stack the-level)))
            ,@(map 'list #'(lambda (saver)
                             `(,(first saver) ,(fourth saver)))
                   change-savers)
            ,@(map 'list #'(lambda (saver)
                             `(,(second saver) ,(third saver)))
                   change-savers))
       ,@(map 'list #'(lambda (saver)
                        `(setf ,(fourth saver)
                               ,(second saver)))
              change-savers)
       (let ((result (progn ,@body)))
         ,@(map 'list #'(lambda (saver)
                          `(setf ,(fourth saver)
                                 ,(first saver)))
                change-savers)
         result))))
```

Examples of the use of `with-changed-level` may be found in several places in the code presented in this thesis, including at section **??**.

11.5 Operators and their shadows

Each operator of the base language (as described in chapter **??**) is shadowed by a procedure in the meta-evaluator. These could in principle be compiled from the corresponding definitions within the tower, but (for historical reasons) most of them are written separately. They are largely generated by Lisp macros, as there is an appreciable amount of standard code at the start of each, to pick apart the level that is passed in as an argument, and, in the case of operators that evaluate all their arguments (such as arithmetic operators, `cons`, etc), to perform the argument evaluation.

There are one or two parts to the definition of each shadowed operator:

- the definition of the version that is kept within the tower, which also includes a reference to the compiled Lisp code to use outside the tower;
- the Lisp code that is compiled to make the shadow definition; for some operators, this can be compiled from the definition that goes inside the tower.

An example of the kind with a separate Lisp function is the primitive `if` operator, which has the Lisp helper function:

```
(defun if-helper (expr level cont)
  "Interpret a LEVEL that wants to do an if-then-else."
  (let* ((condition (second expr))
         (if-case (third expr))
         (else-case (fourth expr)))
    (if (evaluate-by-evaluator-of-level condition level)
        (evaluate-by-evaluator-of-level if-case level)
      (evaluate-by-evaluator-of-level else-case level))))
```

and the definition in the tower:

```
(platypus-def-control-prim
   if                     ; name
   if-helper              ; helper function to use
   (condition then else) ; argument names for definer
                          ; macro to generate for use
                          ; in procedure body
   ;; procedure body (to run in tower for non-shadowed
   ;; interpretation)
   (if (eval-in-level condition int-level)
       (eval-in-level then int-level)
     (eval-in-level else int-level)))
```

Platypus89 operators that are very like existing Lisp functions, and need all their arguments evaluated anyway, may be defined in one piece, as follows:

```
(platypus-def-lispy-prim
  cons        ; name within tower
  cons        ; name of primitive Lisp function to call
  (a d)       ; argument list
  (cons a d)) ; code to run inside or outside the tower
```

The operator `+` is one of those defined to evaluate all its arguments. Its definition is as follows:

```
(platypus-def-lispy-prim + + (a b)
                         (+ a b))
```

In section **??** it is mentioned that reflectors and structure accessors that might return data consisting of closures higher in the tower must check that what they are returning is not the standard processor—in which case they must make and return a copy of it instead. They use the following procedure to do this:

```
(defun unroll-closure (closure)
  "We take a copy of the evaluator if it would be the
standard evaluator, so that the user program can't get
at the standard evaluator to modify it. Otherwise we
return the closure as it is."
  (if (eq closure standard-evaluator)
      (let ((the-closure (copy-closure closure)))
        (setf (closure-number the-closure)
                    (incf closure-counter))
        the-closure)
    closure))
```

It is used in the structure accessor that definitely returns data from further up a tower:

```
(defun closure-evaluator-function (closure)
  "Return the evaluator of CLOSURE."
  (unroll-closure (closure-evaluator closure)))
```

and in anything that might modify a closure (accessed through some other means, such as as a component of a language) in case that closure is the standard evaluator:

```
(defun closure-type-evaluators-setter-function (closure type-evaluators)
  "Set the type-evaluators of CLOSURE to TYPE-EVALUATORS"
  (let ((modifiable-closure (unroll-closure closure)))
    (setf (closure-type-evaluators modifiable-closure)
               type-evaluators
          (closure-original modifiable-closure)
               modifiable-closure)
    modifiable-closure))
```

`funcall` is a particularly important operator, as it is part of the general evaluation mechanism. It evaluates its arguments, and creates a new closure by instantiating the closure held in its first argument. The remaining evaluated arguments are placed on the end of the values list (pushed as onto a stack) and the closure placed on the end of the list of closures run at that level (that is, pushed onto the return stack). The current meta-evaluator is called to evaluate the new closure—that is, to continue evaluation with the new closure as the current evaluation point.

Note that since many languages require their arguments to be evaluated from left to right, but pushing them in order of evaluation leaves them in the

wrong order on the stack, we must in effect collect the evaluated arguments into a list which we reverse before pushing its contents onto the value list. In practice, this is done more efficiently by moving to the new end position of the stack, and filling the values in in reverse order, as explained on page **??**.

Here is the code of `funcall-helper`:

```
(defun funcall-helper (expr level cont)
  "Interpret a LEVEL that wants to do a funcall."
  (let* ((values (closure-values cont))
         (nvalues (value-list-length values))
         (callee-name (second expr))
         (callee (evaluate-by-evaluator-of-level
                  callee-name
                  level))
         (instantiated-callee
          (if (not (closure-p callee))
              (error
               "funcall-helper: Platypus function ~S ~
                 (derived from functor ~S in level ~S) ~
                 is not a closure~%"
               callee callee-name level)
            (copy-closure callee)))
         )



    (setf (closure-number instantiated-callee)
          (incf closure-counter))
    (eval-and-push-sub-exprs (cdr (expression-tail expr))
                             level)
    (let ((nvalues-incl-args (value-list-length values)))

      (setf (closure-values instantiated-callee)
            values)
      (setf (closure-fluid-environment instantiated-callee)
            (closure-fluid-environment cont))

      (setf (closure-evaluator instantiated-callee)
            (closure-evaluator cont))



      ;; Put the new call record onto the stack of the
      ;; current level
      (push instantiated-callee
            (level-call-record-stack level))
```

```
      ;; Let the underlying evaluator continue
      ;; evaluating. This is like handing the new
      ;; code-vector address over to the program
      ;; counter to be run, in a conventional machine.
      (let ((funcall-result (call-meta-evaluator level)))

        ;; put the activation stack back to normal
        (pop (level-call-record-stack level))

        ;; pop the args
        (pop-below values nvalues-incl-args nvalues)
        funcall-result))))
```

Since procedure calling is so central to the system, for efficiency another operator, `funcall1`, is provided that does a `funcall` an extra level lower down the tower. This is suitable for use as the entire body of the in-tower definition of `funcall`. It evaluates its arguments to get to the level need, to pass them on to `funcall-helper`, which it then calls. Its implementation is as follows:

```
(defun funcall1-helper (expr level cont)
  "Interpret a LEVEL that wants to do a funcall1, that is, to
interpret the interpretation of a funcall. funcall1 is provided for
use by interpreters only, really."
  (let* ((res nil)
         (cont-values (closure-values cont))
         (cont-value-1 (nth-value 1 cont-values)))
    (let* ((lcc (level-current-closure cont-value-1))
           (lcx (closure-continuation-expression lcc)))
      (setq res (funcall-helper
                 lcx
                 cont-value-1
                 lcc))
      res)))
```

and its use in the in-tower definitions is this:

```
(platypus-defprim funcall funcall-helper (fun &body args)
                          (funcall1 fun args))
```

Another use of this procedure is that made by the standard evaluator and the meta-evaluator to convert unknown operators into `funcall`s:

```
(platypus-defprim
   :default
   convert-missing-op-to-funcall-helper
      (fun &body args)
 (funcall1 fun args))
```

The stack-based operators are defined in the same way as other operators, but use in their helper procedures the macros `with-popped-args`, `push-results` and `with-popped-args-push-results`, which are explained on page **??**. Here are some examples; not that they are similar to their Lisp-style counterparts, but simpler in that they do not evaluate their arguments but just take what is provided for them on the stack:

```
(defun ps-add-helper (expr level cont)
  "Interpret a level that wants to do a PostScript add."
  (with-popped-args-push-results
   level
   (a1 a2)     ; the args
   ((+ a1 a2)) ; the results
               ; arbitrary body follows, may be empty
   ))
```

The `if` procedure, which has no "else" clause in PostScript:

```
(defun ps-if-helper (expr level cont)
  "Interpret a level that wants to do a PostScript if."
  (with-popped-args
   level
   (condition proc)
   (if condition (with-level-interpret-ps-block level (cdr proc)))))
```

Almost all of the stack-style operators may be defined in this manner, making their definitions generally succinct. It would, naturally, be possible to define them in terms of stack-like operators, as is done in the PostScript-in-PostScript implementation described in [**?**].

11.6 Definition of operators

The operators and their shadows are defined using the macros and procedures presented in the following pages.

Definitions in towers

We need to make each definition in the context of a particular tower, to get various bits of background for definitions.

```
(defvar *platypus-definition-towers* nil
  "A list of towers open for defining things in. New
definitions go in the topmost one of these. This is a
pushy equivalent to CL's jumpy in-package system.")

(defmacro current-definition-tower ()
  "Return the current definition tower."
  '(car *platypus-definition-towers*))
```

Lisp to Platypus expression conversion

In Platypus' closures' expressions, local variables are referred to not by a name (in the sense of a Lisp symbol) but by a number which is stored either as a Lisp character value (as described in section **??**) or, if it will not fit in the range of numeric values denotable by characters, in a local-variable-reference structure. The procedure `lisp-to-platypus-expression` takes in a lambda expression with formal parameter names, in which Lisp `let*` constructs may be used, and converts formal parameter and local variable references into the character or structure form.

Whenever a `let*` form is found, it is converted into a `let-local`, which does whatever its operator definition in the language at run-time provides: it should evaluate its subexpressions, and append (push) them onto the value list. The `let*`ted variable names are then added to the read-time local variables and formal parameters list to be passed to recursive calls of `lisp-to-platypus-expression`— this list is started with the formal parameter list on the outermost call to `lisp-to-platypus-expression`.

This may be used for languages other than Lisp, as long as their parsers produce locally scoped variable declarations in the same form as Lisp's `let*`s.

```
(defun lisp-to-platypus-expression (the-args the-body)
  "Convert lambda-body-like expression with args
THE-ARGS and body THE-BODY from Lisp format to Platypus
format."
  (typecase
   the-body
   (list
    (if (eq (car the-body) 'quote)
        the-body
      (if (eq (car the-body) 'let*)
          (let* ((bindings-list
                  (without-fluids (cadr the-body)))
                 (new-body (cddr the-body))
                 (all-the-args the-args)
                 (bind-exprs (map 'list
                                  #'(lambda (binding)
                                      (progn
                                        (push (car binding)
                                              all-the-args)
                                        (lisp-to-platypus-expression
                                         all-the-args
                                         (cadr binding))))
                                  bindings-list)))
            (let ((new-expr (cons 'let-local
                                  (cons bind-exprs
                                        (lisp-to-platypus-expression
                                         all-the-args new-body)))))
              new-expr))
        (map 'list #'(lambda (x)
                       (lisp-to-platypus-expression the-args x))
             the-body))))
   (symbol
    (if (member the-body the-args)
        (as-local-variable-index the-body the-args)
      the-body))
   (t
    the-body)))
```

Making closures from parse trees

The procedure `make-closure-with-expression` takes an expression, in the form produced by `lisp-to-platypus-expression`, and a tower, and returns a closure ready to run that expression and suitable for instantiation in that tower. Each tower includes a template closure, from which new closures by default inherit components that are not specified explicitly when the closure is created. The template closure normally must have as its evaluator the standard-evaluator that is used in that tower, so that closures instantiated in the initial

context of the tower will be shadowed by the meta-evaluator of that tower.

```
(defun make-closure-with-expression
             (expr &optional (tower (current-definition-tower)))
  "Make a closure for the current definition tower or
for a specified tower, with EXPR as its expression, and
the TOWER (if not the default one) as the second
argument."
  (let* ((template (tower-template tower))
         (closure (make-closure
                   :evaluator (closure-evaluator template)
                   :type-evaluators (closure-type-evaluators template)
                   :language (closure-language template)
                   :continuation-expression expr
                   :procedure-expression expr
                   :values nil
                   :lexical-environment (closure-lexical-environment
                                         template)
                   :fluid-environment (closure-fluid-environment
                                       template)
                   :level (closure-level template)
                   :original nil
                   :number (incf closure-counter)
                   )))
    (setf (closure-original closure)
          closure)
    closure))
```

The fact that this closure is a static closure (see section **??**) is indicated by its `closure-original` field being `eq` to this closure itself.

Closures produced by instantiating this closure will have this closure as their `closure-original` field.

Lisp-style defining forms

Platypus provides several defining forms in the syntax of its Lisp-like base language, corresponding approximately to `defun` in Common Lisp. There are variants for defining ordinary functions, operators, and primitives—that is, shadowed operators. There are several ways of defining shadowed operators, according to whether they require all their arguments to be evaluated automatically before their provided code body is entered, and according to whether a named existing Lisp procedure is to be used as the procedure to wrap into the whole shadow code body, or whether the shadow code is to be generated automatically directly from the code that it shadows.

There is much in common between all these kinds of defining forms, such as the need to translate parameter and local variable references to the form described in section **??**, and they are implemented as macros which call the

procedure `platypus-defun1` to do the common parts of their work.

There are two optional (keyed) arguments to this macro, which indicate whether the form being defined is to be shadowed (`primitive`), and whether it is to be bound in the language (`define-as-operator`) or in the environment (as a callable procedure). The callers of `platypus-defun1` use various combinations of these arguments, the only combination not used in practice being a shadowed callable procedure—all shadowed procedures are normally run directly from the evaluator, as operators, rather than being called through the `funcall` operator.

```
(defun platypus-defun1 (fun-name fun-args fun-body
                                &key primitive define-as-operator)
  "Define a platypus function, called FUN-NAME, with
args FUN-ARGS and body FUN-BODY. Keyword argument
PRIMITIVE, if specified, specifies a lisp function with
which to implement this function. This is mapped in the
shadow map of the current definition tower."
  (let* ((the-body (if (and (> (length fun-body) 1)
                            (stringp (car fun-body)))
                       (cdr fun-body)   ; throw away docstring
                     fun-body))
         (the-name (if (listp fun-name)
                       (cadr fun-name)
                     fun-name))
         (the-env (if (listp fun-name)
                      (car fun-name)
                    (if define-as-operator
                        (tower-language
                         (current-definition-tower))
                      (tower-environment
                       (current-definition-tower)))))
         (closure (make-closure-with-expression
                   (lisp-to-platypus-expression fun-args the-body))))
    (when *print-platypus-defuns*
          (format t "Setting defun in env: ~A~%"
                  the-env))
    (platypus-set-environment-variable the-env the-name
                                       closure)
    (when (not (null primitive))
          (when *print-platypus-defuns*
                (format t "Setting primitive~%"))
          (platypus-set-environment-variable
           (tower-operator-shadow-map
            (current-definition-tower))
           closure primitive))
    nil))                               ; value to return from macro
```

Defining ordinary procedures

This is the simplest use of `platypus-defun1`. At this stage in the definition process, the procedure body must be a single form, rather than an implicit `progn`.

```
(defmacro platypus-defun (fun-name fun-args
                                   ;;&body
                                   fun-body)
  "Define a platypus function, called FUN-NAME, with
args FUN-ARGS and body FUN-BODY."
  `(platypus-defun1 ',fun-name ',fun-args ',fun-body))
```

Defining non-shadowed operators

`platypus-defop` is similar to `platypus-defun`, but it makes the definition in the language of the initial context of the tower, rather than in its general environment.

```
(defmacro platypus-defop (op-name op-args
                                  ;;&body
                                  op-body)
  "Define a platypus operator, called OP-NAME, with
args OP-ARGS and body OP-BODY."
  `(platypus-defun1 ',op-name
                    ',op-args
                    ',op-body
                    :define-as-operator
                    t))
```

Defining ordinary primitives

`platypus-defprim` is the simplest way to define a shadowed primitive. It is similar to `platypus-defun`, with the addition of the `prim-name` argument, which is the name of the shadow procedure to run when interpreting this procedure and finding that it is eligible for shadowing (see sections **??** and **??**).

```
(defmacro platypus-defprim (fun-name prim-name fun-args
                                    ;; &body
                                    fun-body)
  "Define a platypus function, called FUN-NAME, and
implemented by the lisp function called PRIM-NAME, with
args FUN-ARGS and body FUN-BODY. The primitive named by
PRIM-NAME is called with one argument, the level to
interpret."
  `(platypus-defun1 ',fun-name ',fun-args ',fun-body
                    :define-as-operator t
                    :primitive ',prim-name))
```

Defining control primitives

The following function is for use in a macro-expander, `platypus-def-control-prim`, which is presented after it. It takes an expression template, which is a list of names for successive sub-expressions of the expression, and produces a `let`-binding list for binding those names to those parts of the expression. If `do-eval` is true, the code returned has the sub-expressions evaluated in the context of the level, accessible in the context in which this binding code is run, under the name given by `level-name`, and the results of those evaluations are bound; otherwise the un-evaluated sub-expressions are bound throughout.

For example, the description of the arguments for `if`, `(condition then else)`, given the name `expr` for the expression will produce the list:

```
((condition (nth 1 expr))
 (then (nth 2 expr))
 (else (nth 3 expr)))
```

whereas the description of the arguments for `cons`, which needs its arguments evaluated for it, will expand to:

```
((a (eval-in-level (nth 2 expr) level))
 (b (eval-in-level (nth 3 expr) level)))
```

where `level` is passed in as the name of the level in which to evaluate the arguments.

```
(defun make-control-arg-splitter (level-name
                                  expr-name
                                  arg-names
                                  do-eval)
  "Make a let-binding-list, in which the argument expr
bound to EXPR-NAME is split into its successive part
and bound, part by part, to the names in ARG-NAMES. If
the third argument, DO-EVAL, is non-nil, each argument
expr is wrapped in a call to eval."
  (do* ((args arg-names (cdr args))
        (arg (car args) (car args))
        (arg-index 1 (1+ arg-index))
        (let-list nil))
       ((endp args) (nreverse let-list))
       (if (eq arg '&body)
           (progn
             (setq args (cdr args))
             (setq arg (car args))
             (push (if do-eval
                       `(,arg (eval-in-level
                               (nthcdr ,arg-index ,expr-name)
                               ,level-name))
                     `(,arg (nthcdr ,arg-index ,expr-name)))
                   let-list))
         (push
          (if do-eval
              `(,arg (eval-in-level
                      (nth ,arg-index ,expr-name)
                      ,level-name))
            `(,arg (nth ,arg-index ,expr-name)))
          let-list))))
```

The results of `make-control-arg-splitter` are used in the following macro-expander:

```
(defmacro platypus-def-control-prim (fun-name prim-name fun-args
                                                &body fun-body)
  "Define a platypus function, called FUN-NAME, and
implemented by the lisp function called PRIM-NAME, with
args FUN-ARGS and body FUN-BODY. The primitive named by
PRIM-NAME is called with one argument, the level to
interpret. If interpreted, the body has the parts of
the expression of the current continuation closure of
the argument level split out into the appropriate named
variables as specified by fun-args."
  `(platypus-defun1 ',fun-name '(int-expr int-level int-cont)
                    '(let* (,@(make-control-arg-splitter
                               'int-level
                               'int-expr
                               fun-args
                               nil))
                       ,@fun-body)
                    :define-as-operator t
                    :primitive ',prim-name))
```

Defining Lisp-like primitives

The Lisp-like primitives are those that require all of their arguments to be evaluated before entering the provided code body. The evaluation of the arguments is done by some code which the following macro wraps around the provided code body. The extra code calls `eval-sub-exprs`, which works it way along the argument sub-expressions evaluating each one and returning a list made from the results of these evaluations. `eval-sub-exprs` is presented in section **??**. This is used for the version that runs inside the tower. For the shadow version, the argument evaluation is done inside the code fragments returned by `make-control-arg-splitters`, which arranges this argument evaluation when called with its `do-eval` argument non-nil.

```
(defmacro platypus-def-lispy-prim (fun-name prim-name fun-args
                                            &body
                                            fun-body)
  "Define a lispy platypus function, called FUN-NAME,
and with an implementation based on the lisp function
called PRIM-NAME, with args FUN-ARGS and body FUN-BODY.
The arguments for the call are evaluated by the wrapper
provided by this macro, and given to the function named
by PRIM-NAME."
  (let ((shadow-name (intern (concatenate 'string
                                          "frame-"
                                          (symbol-name prim-name)))))
    `(progn
       (compile
        ',shadow-name
        `(lambda (expr level cont)
           (let ((the-expr expr))
             (apply #',',prim-name
                    (eval-sub-exprs the-expr level)))))
       (platypus-defun1 ',fun-name
                        ;; ',fun-args ',fun-body
                        '(int-expr int-level int-cont)
                        '(let* (,@(make-control-arg-splitter
                                   'int-level
                                   'int-expr
                                   fun-args
                                   t))
                           ,@fun-body)
                        :define-as-operator t
                        :primitive ',shadow-name))))
```

Defining Lisp-like primitives based on a given expression

These primitives, like those defined by `platypus-def-lispy-prim`, have their arguments evaluated before running the given code body, but whereas those call a Lisp function that exists anyway, this macro `platypus-def-lispy-expr-prim` also create the Lisp function from the expression provided, and compiles it.

```
(defmacro platypus-def-lispy-expr-prim (fun-name fun-args
                                                ;; &body
                                                fun-body)
  "Define a lispy platypus function, called FUN-NAME,
and with an implementation based on the lisp expression
given as the function body, below, with args FUN-ARGS
and body FUN-BODY. The arguments for the call are
evaluated by the wrapper produced by this macro, and
given to the function supplied as FUN-BODY."
  (let ((shadow-name
         (intern
          (concatenate 'string
                       "frame-"
                       (symbol-name fun-name)))))
    `(progn
       (compile
        ',shadow-name
        `(lambda (expr level cont)
           (let ((the-expr expr))
             (apply #',',(lambda ,fun-args ,fun-body)
                    (eval-sub-exprs the-expr level)))))
       (platypus-defun1 ',fun-name
                        '(int-expr int-level int-cont)
                        '(let*
                             (,@(make-control-arg-splitter
                                 'int-level
                                 'int-expr
                                 fun-args
                                 t))
                           ,@fun-body)
                        :define-as-operator t
                        :primitive ',shadow-name))))
```

Defining static closures

As described in section **??**, closures are used to represent procedures available for calling. These are not true closures, as they do not have all of the fields filled in—they are completed in the copy made when instantiating the closure. When a closure is instantiated (by `funcall-shadow`, for example), it is copied and completed. The `closure-original` field of the copy points to the original from which it was copied, and this is used to determine whether the closure is shadowed.

```
(defmacro def-unclosure (name args
                                ;;&body
                                body)
  "Set NAME to a new closure with ARGS for its
arguments and BODY for its expression and the rest made
to distinguished nonsense values.  This may then be
copied into towers having the rest of the slots filled
in as appropriate for that tower. It is used for
defining things that are useful in many towers. BODY
may have a docstring, which is thrown away."
  #| (when (and (stringp (car body)) (not (endp (cdr body))))
        (setq body (cdr body))) |#
  `(let ((expr (lisp-to-platypus-expression
                ',args ',body)))
     (setf ,name  (make-closure
                   :evaluator nil       ; :no-evaluator
                   :language :no-language
                   :type-evaluators
                   :no-type-evaluators
                   :procedure-expression expr
                   :continuation-expression expr
                   :values :no-values
                   :lexical-environment :no-lexical-environment
                   :fluid-environment :no-fluid-environment
                   :original nil
                   :level nil
                   :number (incf closure-counter)
                   ))



     (format t "def-unclosure: ~S now set to ~S~%"
             ',name ,name)
     (setf (closure-original ,name)
           ,name)
     ;; This will work even if ,name is
     ;; standard-evaluator, so long as we do the
     ;; def-unclosure for standard-evaluator before any
     ;; other def-unclosures
     (setf (closure-evaluator ,name)
           standard-evaluator)
     ,name))
```

Placing definitions in towers

Procedure and operator definitions are placed in the appropriate part—language or environment—of a tower. The tower they go in is the `current-definition-tower`, which is the top of a stack of such towers. To change the tower into which defini-

tions are placed, the procedures `begin-define-in-tower` and `end-define-in-tower` are provided.

```
(defun tell-current-definition-tower ()
  "Say which the current definition tower is."
  (format t "Definitions now go into tower ~S~%"
          (current-definition-tower)))

(defun begin-define-in-tower (tower)
  "Make TOWER the current tower for definitions."
  (setq *platypus-definition-towers*
        (cons tower *platypus-definition-towers*))
  (tell-current-definition-tower))

(defun end-define-in-tower (&optional tower)
  "Revert to the previous tower for definitions. TOWER
is used to check the nesting of definitions, unless it
is nil."
  (unless (or (null tower)
              (eq tower (car *platypus-definition-towers*)))
          (error "Closing towers for definition in the wrong order."))
  (setq *platypus-definition-towers*
        (cdr *platypus-definition-towers*))
  (tell-current-definition-tower))
```

In defining the helper functions for stack-style operators (as used in PostScript and FORTH) the macros `with-popped-args`, `push-results` and `with-popped-args-push-results` are useful. They take values from the end of a level's value list and bind them to Lisp names, and push result of Lisp expressions onto the value list. Their definitions involve many obscure macros, but, with some of the internal macros expanded out, look like this:

```
(defmacro with-values (some-vl &body some-body)
  "Using the values list SOME-VL, run SOME-BODY."
  `(let* ((.values-list. ,some-vl)
          (.values-last. (value-list-last .values-list.))
          (.values-max. (value-list-max .values-list.))
          (.values-data. (value-list-data .values-list.))
          )
     (declare (type fixnum .values-last. .values-max.)
              (type values-list .values-list.)
              (type simple-vector .values-data.))
     (let ((the-with-values-result
            (progn
              ,@some-body)))
       ;; assume only this has changed:
       (setf (value-list-last .values-list.)
             .values-last.)
       the-with-values-result)))
```

`with-values` makes a particular value list into the current value list, as used by the macros tt the-value-list-last, tt the-value-list-max, tt the-value-list-data, and tt the-nth-value.

```
(defmacro with-values-of-level (some-level &body some-body)
  "Using the values list of SOME-LEVEL, run SOME-BODY."
  `(with-values (level-values ,some-level)
                ,@some-body))
```

`with-values-of-level` builds on `with-values`, for the commonest use of it.

The expander for the macro `with-the-popped-args` needs an auxiliary procedure which returns a `let` binding list:

```
(defun make-popping-arg-list (args)
  "Make a let binding list for ARGS,
with the current value list."
  (let ((i (length args)))
    (map 'list #'(lambda (arg)
                   (decf i)
                   `(,arg (the-nth-value ,i)))
         args)))
```

The bindings described in that list are executed in the provided code body by the following macro:

```
(defmacro with-the-popped-args (args &body body)
  "Using the values list set by with-values-of,
pop the args needed to fill ARGS, and run BODY."
  `(let ,(make-popping-arg-list args)
     (setf (value-list-last (the-value-list))
           (decf (the-value-list-last) ,(length args)))
     ,@body))
```

This macro packages the above on into its most useful form:

```
(defmacro with-popped-args (level args &body body)
  "Using LEVEL to supply the arguments,
pop the args needed to fill ARGS, and run BODY."
  `(with-values-of-level ,level
                         (with-the-popped-args ,args
                                               ,@body)))
```

The following definitions are the complement of the preceding ones—pushing results back onto the stack. The results are specified as a list of arbitrary Lisp expressions. The following procedure is used by the macro expander after it:

```
(defun make-result-placer-pushers (forms n)
  "Return a list of code to put FORMS into the
current value list; for efficiency, we are told
that there are N forms."
  (let ((i n))
    (map 'list #'(lambda (form)
                   (decf i)
                   `(setf (the-nth-value ,i)
                          ,form))
         forms)))
```

This macro runs through its arguments, which are forms (expressions), making code to evaluate each form and push it onto the stack:

```
(defmacro push-the-results (&rest results)
  "Using the values list set by with-values-of, push &REST RESULTS."
  (let ((added-value (length results)))
    `(progn
       (when (>= (+ ,added-value (the-value-list-last))
                 (the-value-list-max)) ; is this enough?
             (extend-value-list .values-list. ,added-value))
       (setf (value-list-last (the-value-list))
             (incf (the-value-list-last) ,added-value))
       ,@(make-result-placer-pushers results added-value))))
```

This packages it into its most useful form:

```
(defmacro push-results (level &rest results)
  "Into LEVEL push &REST RESULTS."
  `(with-values-of-level ,level
                         (push-the-results
                          ,@results)))
```

The following macro combines the two stack facilities above, and may be used as the outermost part of each stack-based operator shadow body:

```
(defmacro with-popped-args-push-results (level
                                         args
                                         results
                                        &body body)
  "Using LEVEL to supply the arguments, pop the args
needed to fill ARGS, and run BODY, finally pushing
RESULTS onto the stack of LEVEL."
  `(with-values-of-level ,level
                         (with-the-popped-args
                          ,args
                          ,@body
                          (push-the-results ,@results))))
```

11.7 The C implementation of Platypus

The C implementation of Platypus is a complete system implementation, not relying on another language's support system and library for memory management and other facilities. Because of this, it is fairly large and complex, and contains much code not directly relevant to the research.

C-Platypus keeps the entire tower system within a *storage heap*. The meta-evaluator is not visible from the tower, and is a C program, which is not contained within the heap. All values in the heap are tagged with their type; to allow as much flexibility as possible, the tags are whole machine words (making each C-Platypus word two machine words), and point to the type descriptors concerned. Each object in the heap has a header, consisting of two C-Platypus words: the length and the type. (The type of the type points to the type descriptor for the whole object, and the value of the type is the object itself—a redundant reference that turned out very useful for debugging.) This type system makes all types first-class [**?**]. There is no distinction between predefined system types and user-defined types.

The garbage-collector is a stop-and-copy one. It is complicated by the need to update the shadowing tables in the meta-evaluator. As an interesting reflection curiosity, the heap is also an object that contains itself.

The shadowing tables use a perfect hashing scheme from addresses within the heap to addresses in the meta-evaluator. The hash code is the low few bits of the address in the heap, and the size of the hash table is a power of two. In the initialization of the heap with the shadowed objects, and when the garbage collector moves them, when a shadowed object is about to be allocated, if its address has the same hash code as one that has already been marked in the shadow map, words of memory are thrown away until one with a free hash code is found. This typically wasted 22 words at any one time, and seems a small overhead to pay for such a fast hashing scheme. However, it is not so suitable for a system using many more mapping tables (environments) as the wastage would go up, as would the amount of special treatment of objects done by the garbage collector.

The meta-evaluator is similar to the one written in Lisp for Platypus89, but is written in as a loop, instead of using the compiler's tail-recursion removal. It does not use a level-type-evaluators environment mechanism for evaluating things of different types, but has it built in to a `switch` statement.

The shadow operators are implemented in a similar way to those in Platypus89, but the C code that defines them must be pre-processed before compilation, to insert some of the standard helper code for taking apart levels and closures, and for evaluating arguments for operators that simply want all their arguments evaluated.

11.8 Summary of the meta-evaluator

Platypus is a trial implementation of a reflective tower-based evaluation system in which data representations inside and outside the tower are the same. It has been through two broadly similar implementation generations, one in C and one in Common Lisp.

Dynamic typing is used throughout the system, both inside and outside the tower. All objects in the tower are kept in a storage heap; in the C implementation this is scavenged by a stop-and-copy garbage collector, which must pay attention to updating variables of the meta-evaluator as it moves the things to which they point.

While several meta-evaluator variables point into the heap, it is not possible for a program in the tower to find from an object in the heap what it corresponds to in the meta-evaluator (although a meta-evaluator that makes this information available could be written).

The meta-interpreter must contain:

- a meta-evaluator that mimics (shadows) the standard evaluator as well as generating levels on demand;
- shadow operator definitions shadowing some of the operators in the tower's base language;
- and an argument evaluator for shadowing operators to use.

These are all fairly similar to their equivalents within the tower.

The meta-evaluator may be reduced to a very concise form built around one function, which is appropriately very similar to the corresponding function for the concise form of the standard internal evaluator for a tower. This function consists of four procedure calls and two environment lookups.

The code for all the shadow operator definitions has much in common; in particular, all operators that must evaluate all their arguments independently share an argument evaluator mechanism. Such operators are built around primitive procedures in the implementation language, by a macro-preprocessor, that takes functions in the implementation language, and produces wrapper functions for use as the shadow operators.

Chapter 12

Results

12.1 Overview of results

In researching this thesis, I developed three implementations of Platypus, at each stage having a different level of experience with reflective programming. The three systems have the same underlying principles, but different implementation technology. Each contributed some different things to my understanding of practical and theoretical procedural reflection, whilst all agreed on the major points.

Performance testing of Platypus took the form of running a set of simple benchmarks—a mixture of the Griss tests and Gabriel tests [**?**] in Lisp and a recursive picture "Circle Limit" in PostScript—on a variety of configurations of the system. The same benchmarks were also run on conventional Lisp and PostScript interpreters to provide a reference point.

The tests were run for each the two versions of the evaluator and meta-evaluator—that is, the versions without and with the `boojum` and `snark` functions. For each of these versions, the tests were run

- with the evaluation system in its initial state (in which the operators used by the application are always shadowed directly);
- with the evaluator changed, but to an identical one, thus forcing another level of interpretation to be added;
- with parts of the evaluator and some of the operators changed to make the whole level use association-list based deep binding instead of Platypus's usual hash-table based shallow binding.

The expected results were that the performance of configuration in which the evaluator and operators used by the application were shadowed directly would be within an order of magnitude of the performance of the conventional Lisp and PostScript systems, and the versions in which the evaluator and operators used by the evaluator and operators were shadowed would be very much slower.

12.2 The design ideas

Before developing the first implementation, I spent some time in devising a way of representing a language interpreter in a manipulable form. My first attempt used re-write rules to describe a language, but it soon became clear that, as well as not being very expressive for describing most languages, they were also hard to manipulate for reasoning about and computing with languages.

I experimented briefly with a message-passing representation of closures in Lisp. I soon realized that, for manipulating languages, this was not much better than the re-write rules, although, with each operator being an object, it was some improvement.

Although the different possible representations for languages were the same in the computational power, they were very different in their expressiveness. Both of these forms lacked expressiveness in this problem domain. Expressiveness is hard to quantify, although for many pairs of expressions of the same idea, one is often clearly more expressive than the other. Yet this is a subjective evaluation, and it is difficult even to describe the parameters we use in deciding expressiveness. Conciseness is, perhaps, an important factor: a good expression of an idea is usually more concise than a poor one. This could be quantified objectively by the number of terms used in the expression, as a poor match between an idea and the language used to express the idea usually results in extra terms that are present only to make up for the poor match. (Consider, for example, using FORTRAN instead of Lisp for consed-list manipulation, or a command shell language such as sh for Fourier Transforms.)

Interpretive closures

My third way of structuring a language for representation divided it into operators and proccesor, each being represented as a procedure. The closure operation for procedures was extended to close over the *language* (environment of operators) and the evaluator, and thus this representation automatically provided interpretive towers.

This was the representation I took into the first implementation (written in Cambridge Lisp), which showed it to be basically suitable. I used the same representation in C-Platypus, and extended it slightly in Platypus89, while still keeping the same structure for the actual representation of languages—the changes were concerned with tower representation and interpretation.

Meta-towers

The idea of many-dimensioned towers, or meta-towers, was important in designing the system, and they are provided in Platypus89 as a compile-time option, making a slight change in the performance.

The slight loss in performance brought about by adding meta-towers appears because the operation `call-meta-evaluator` (used in the real, compiled, meta-evaluator) can no longer be simply a `funcall` in the substrate language,

but must now switch on whether the meta-evaluator of the tower given as an argument to `call-meta-evaluator` is shadowed by the real one. For simplicity and efficiency, I used a different implementation of shadowing from that used in the normal towers; when the meta-evaluator has not been reflected into, it simply *is* the compiled one, and this is called with a Lisp `funcall`. Otherwise, a Platypus closure is there, and a level is launched to evaluate it. The selection here is done by a Lisp `typecase` form, although it would have been more consistent with the rest of Platypus for it to be done by an environment—another shadow map, in effect. The reason why the real meta-evaluator can be built into the tower at this point is that this is the direct, and singular grounding point of the tower. It would have been possible, although inelegant, to connect the meta-evaluator of an ordinary tower to that tower by putting it in place of the rolled-up boring section of the tower, and having the unrolling mechanism find the thing from which to unroll copies from elsewhere. The asymmetry of the direct connection seems more excusable in the light of the idea that, whereas reified data structures continue indefinitely outward, the real meta-evaluator must start from somewhere. This is not so much where the buck does not stop, as where it starts!

The development of the meta-tower system was closely linked with the theoretical development of the type system.

The type system

Platypus is a dynamically typed system, providing dynamic typing to the languages it supports. It would be much harder to implement dynamically typed languages on only a statically typed base, as static typing simply introduces a pre-execution check, after which evaluation of the program may procede with dynamic types (although all the values' types will be predictable). Implementing dynamic types on statically typed system requires the definition of a (static) type including a (dynamic) type field and a value field that can contain values of any type used in the system (and so must be of a union type—and thus not strictly statically typed). Languages with static typing may have their type-checking done when programs are read and converted into Platypus' expression representation.

In all the substrate languages (Cambridge Lisp, C, and Common Lisp), the type system provided was not powerful enough in itself to support Platypus' requirements directly. Common Lisp was the best here, lacking only the facility for the programmer to add new atomic (non-structured) types, which would have been useful for local variable indices; instead, string-characters (see sections **??** and **??**) were used for this, as being a distinct type of integer that was already available.

The first two implementations had double type systems, using both the type system of the implementation language, and a separate type system for programs running in the tower. In the first implementation, this was very confusing, as the tower type system and the meta-evaluator type system were both dynamic, and some code in the meta-evaluator used one system, and some used the other.

In C-Platypus, this was not so confusing, as C's type system is static, and the static types were useful for ensuring that data stored in the tower could not be confused with non-reifiable internal data of the meta-evaluator.

In the first two implementations, the type system of the implementation language was concealed, sometimes with some effort, from programs running in the tower.

In Platypus89, the type system of the substrate language is used within the tower, which made the implementation very much quicker and easier both to construct and to understand. Common Lisp's type system is sufficiently similar to that required by Platypus that this has proved to be a good strategy.

The uniform representation

Early on in designing Platypus, I decided that the representation of reified information handled by a program in the tower was to be the same as that used for the same information when reflected. (In some systems, this is not the case. For example, typical SmallTalk implementations have to construct reified stack frames with some effort when asked for a stack frame. In Platypus, reifiers neither modify values nor construct new ones, except to realize levels, as described in section **??**—an activity which, from the application program's viewpoint, is not happening: it is backstage.) The success of this scheme depends on being able to evaluate efficiently using data representations that are suitable for use as reified data, as long as we generate all stack frames in heap. One representation is suitable for both uses. If we use conventional stacks, to avoid excessive garbage generation, the simplicity of our reifiers is lost, and a more active reifier is needed. Deciding which stack frames must be kept after returning from them, and reifying them then, is probably an appropriate solution, but may well be difficult to implement. This is discussed in [**?**]. In some cases, we know that we can use the substrate system's stack frames instead of allocating them in the heap, as explained in section **??**.

Shadowing

Shadowing is an essential part of tower interpretation, and so appears in all the Platypus implementations. There are two approaches to implementing shadowing:

- storing the shadow of a closure a field in the closure;
- not storing the shadow of a closure a field in the closure, but instead recognizing shadowed closures from outside.

The second of these is better, since the shadowing information belongs on the outside of the tower, and so should not be part of any object within the tower.

I used the first of these in the Cambridge Lisp implementation of Platypus, and for C-Platypus switched to the second, to make sure that the tower remained hidden. (C-Platypus does not reveal higher dimensions of towers.) Having full

control of the storage heap, it was possible to make a perfect hashing system from addresses of closures to their shadows, thus bringing the overhead of the mapping system down to only a little more than that of structure access, involving, at the machine code level, just 4 instructions: a `bitwise and`, an `indexed load`, a `comparison`, and another `indexed load`.

In Platypus89, I used a similar mapping system, although using Common Lisp's `gethash` (which is rather less efficient) instead of the perfect hashing system described above.

It may be possible to make a further short-cut for speed, using numbers stored in each closure, and doing a faster form of hashing based on the number, perhaps like the address hashing used in C-Platypus. This avoids hardwiring shadows into shadowed closures, while still providing very fast access from a closure to its shadow.

12.3 The implementations

Platypus-0

Platypus-0, written in Cambridge Lisp, was extremely slow; at 1.2 seconds to interpret `(cons 1 2)` (on an Orion-1, a machine of similar processing power to an early Vax), it showed that the ideas were viable, while leaving open-ended the question of whether an efficient implementation would be possible.

Platypus-0 provided a more sophisticated type system than the substrate Lisp, using conses of `(type . value)` to represent single values. This made the coding very confusing, as it was hard to keep track of whether a particular value was a `(type . value)` pair, or a simple Lisp value (perhaps with the type it had once had, removed). The very simple dynamic typing provided by Cambridge Lisp was not really powerful enough to be helpful here.

C-Platypus

C-Platypus was written largely with speed in mind. Using an unusual hashing system to map between the tower and the meta-evaluator, it set out to show that it was possible to implement a tower efficiently with an absolutely pure reified heap without any pointers out from the heap to the the meta-evaluator.

This implementation was never sufficiently reliable to perform extensive timing tests, but it seemed to be capable of evaluating simple Lisp and FASL (a stack-based language oriented toward efficient construction of data structures, used for loading Lisp programs) code without noticeable inefficiency.

C-Platypus built some very complex data structures when starting. These were built entirely by hand-coded C routines, which were one of the major reasons for moving back to Lisp for continuing the work. The initialization code was very sensitive to changes, and difficult to maintain or re-organize. In conjunction with the use of a statically compiled implementation language, and the lack of higher dimensions of the tower, this made it impossible to extend the

meta-evaluator once the tower had started to run. This is very much against the spirit of true reflective systems! The difficulty of initializing the system was one of the reasons for its being abandoned and superseded by a system built on a more powerful and flexible substrate language.

One very definite result of this stage of the work is that it is important to allow the building of the meta-evaluator incrementally, rather than providing all the pieces and putting them together at once. Retrospectively, in embarking on the C implementation of a tower, I should first have planned tools to extract the initialization code from comments or macros in the rest of the source code.

The implementation of C-Platypus included two languages, a Lisp-like one for general use (like the base language of Platypus89), and a FORTH-like language for use in program loading (in Lisp terminology, a FASL reader—a simple language for constructing data structures faster than can be done from a Lisp-like language, as its syntax and semantics are even simpler than those of Lisp).

C-Platypus made no provision for higher dimensions of the tower. Partly due to this, and partly due to the greater ease of development in Lisp, particularly concerning structuring and initializing the system, I proceeded to the third implementation, Platypus89. This left C-Platypus insufficiently reliable for timing results, so these are omitted from the result table given here.

Platypus89

Platypus89 was very much simpler to implement than the earlier systems, as well as being much more powerful. Common Lisp provides the combination of a reasonable type system (not quite present in C, and not present in Cambridge Lisp) with incremental compilation and the evaluation facilities of the language being made available at compilation time.

The technological contribution of Common Lisp made a great difference here, but perhaps not as much as the further understanding of reflective interpretation that I had developed while implementing C-Platypus, and while pondering it afterwards. Two conceptual advances were most significant here:

- the mapping of types at successive levels of meta-towers;
- an understanding of higher-dimensioned towers.

Although structurally the former is but part of the latter, it is the former that gave rise to the latter.

The implementation of Platypus89 includes a starter set of two languages: a dialect of Lisp, and a subset of PostScript. Both of these have all their operators defined directly in the base language of the system, although there is nothing to stop a language from using a mixture of shadowed and non-shadowed operators—indeed, for many languages this would be the case—it is desirable, for efficiency, for a language's input parser to use shadowed operators wherever it can; for example, many languages will have a two-way conditional, `if... then... else...`, for which the base language's `if` operator may be used; whereas FORTRAN's subroutine bodies with `GOTO`s, including three-way

arithmetic `GOTO`s, and calculated `GOTO`s; or traditional Lisp's `prog... go... return...` forms, are obscure and high-level enough—away from the natural architecture of the system—not to match anything in the base language, and these would be implemented by non-shadowed operators, that are always interpreted by the meta-evaluator.

Performance of Platypus89

To compare the performance of Platypus89 against other interpreters, I used as the main benchmarks:

- the *Griss tests* for Lisp—a set of simple benchmarks testing the performance of various parts of Lisp systems; I translated these from PSL (Portable Standard Lisp) to Platypus89's base language dialect of Lisp, and wrote a few macros for Common Lisp to enable it to run the tests from the same file.
- *circle limit*, based on a picture by M. C. Escher [**?**], for PostScript, the only change required to the test file being renaming some tokens to avoid dependence on case (as Platypus89 uses a Lisp readtable and the Common Lisp reader system to read PostScript, it is case-insensitive, unlike real PostScript).

Lisp performance

The Griss tests consist of many separate tests, each of which does one thing many times. A table of the results for this is given below. The number of iterations for each test was chosen to bring the total time for the fastest version (usually the Common Lisp interpretive evaluator—the interpretive implementation of the substrate language) to around 10 seconds for each test, the timing resolution of the system being 10mS. The computer used for the tests was a Solbourne SPARC system (Sun-4 compatible) and the Common Lisp interpreter is that of Harlequin's LispWorks, which is also the substrate system on which Platypus89 was compiled and run for the tests.

The times are in seconds, and the column "cl/pl" is the ratio of speeds between Common Lisp (LispWorks) and Platypus (in the version not using the `snark` function), and the column "cl/snark" is the ratio of speeds between Common Lisp and the `snark`-based meta-evaluator.

It can be seen from this that, for these tests, Platypus89 is around one-half of the speed of a plain Lisp interpreter, or better—which is well within the one-tenth (see section 2) that I chose as my acceptance level.

I had expected the version using `snark` to be slower than the other, because of the extra procedure calls; at least one (`evaluate-anything` calling `snark`) at each evaluation. This did not prove to be the case; the most likely explanation is that making each of the most central routines smaller and simpler allows the

substrate compiler to emit more efficient code, with a higher proportion of the live values being in CPU registers at any one time, and less traffic between CPU registers and the spill area (the part of the stack frame in which local values not currently in CPU registers are stored).

The complexity of `snark` and the meta-evaluator routines build around it is, as it turns out, closely matched to the number of registers available on the SPARC computers used to run it for the benchmarks.

PostScript performance

Since Platypus' PostScript implementation does not contain any actual graphics routines, but instead outputs distilled PostScript[1], it is harder to compare this with a conventional PostScript interpreter. To give a more realistic comparison, I compared it with `still.ps`, a program which redefines the graphics operators of PostScript to output distilled PostScript instead of drawing the picture itself. The PostScript system used for comparison is Harlequin's ScriptWorks, and is run on the same kind of computer as Platypus89 was for these tests—again, a Solbourne SPARC system. The results of this benchmark are shown below:

This gives a performance of 42% of that of the ScriptWorks interpreter; as with the Lisp, this is well within the 10% aimed for at the start of the project.

The commonly used PostScript benchmarks are not particularly relevant here, as they are chosen primarily to test the graphics system attached to the PostScript language, rather than the language itself. The program used here was chosen for its complex control flow; it uses a simple picture drawn with a graphic tools, with hand-written program for repeating that picture in a tessellating pattern.

Performance with extra levels of interpretation

Performance testing of the reflective facilities of Platypus89 is, of necessity, largely self-referential. Here are the results of adding one level of interpretation in various versions of the system:

[1] Distilled PostScript is the result of running a PostScript program with the low-level drawing operators redefined to output the PostScript code needed to call them, with the appropriate arguments. All co-ordinate system transformations have been done by the time they are called, and loops in the input program do not appear in the output—they have been unrolled by this processing. Thus, a distilled PostScript program consists purely of a sequence of drawing operators—no transformations, and no flow control beyond the implicit sequential execution.

The times are in seconds, and the column "pl/int ratio" gives the ratio of speeds between Platypus shadowing the application's evaluator and operators directly, and it interpreting them. Likewise, "snark/boojum" ratio gives the ratio of speeds for `snark` evaluating the benchmarks directly, and `snark` evaluating `boojum` evaluating the benchmarks, and so is equivalent to the pl/int ratio of the non-`snark`-based system.

Some of these figures seem anomalous, in that adding a level of interpretation improves the performance. This is not an impossibility; since a shadowed procedure and its shadow are two different extensions (implementations) of the

same intension (specification) it is possible, for example, for a shadow to be less efficient than the interpreted procedure it shadows. In particular, the meta-evaluator has been optimized (by helping the substrate compiler by giving it more type information) especially heavily. However, the results are strange in places. They cannot be explained away simply as timing inconsistencies, since timing with a resolution of 10mS over a total of 10S gives an accuracy of 0.1%—and some of the speed ratios are on the mysterious side by twice this much.

A considerable loss in performance might be expected from adding levels of interpretation between the application and the meta-evaluator. However, since the meta-evaluator consists of several small procedures, of which not all need be changed for many practical uses of reflection, it need not all be interpreted, and so the overhead is lower than it would be for reflective architectures of coarser granularity of reflection. As the table shows, in practice the loss of efficiency is just a few percent. Experiments with interpretation of the entire evaluator showed a slowdown factor of around 40, so the fine-grained reflective changes yield a considerable advantage when reflection into the interpreter is brought into play, a the cost of extra procedure calls at all times, when compared with an evaluator using a single procedure with a `typecase` control structure to do this.

As may be expected, the `boojum`/`snark`-based system suffers less loss of performance on reflection, as less of the system is given another level of interpretation by each change. This is the positive side of the expected trade-off, but as explained on page **??**, the negative side of the expected trade-off turned out to be another positive in the realized trade-off, and so, at least in this situation, the `snark`-based evaluator is the faster of the two.

Room for improvement?

An evaluator built around `boojum` and `snark` is highly parameterized, and thus inherently difficult for a compiler to optimize as well as it can an ad-hoc evaluator.

Changes to the levels of optimization allowed to the compiler, particularly in critical areas such as the evaluator itself, affected the performance by several percent. These changes include switching on or off various checks made by the compiled code, and adding type declarations that promise to the compiler that the expressions declared will be of specific types.

Although in general I expected code in which the same functionality was packed into fewer procedures (because of using fewer function calls), in practice smaller procedures allowed more effective use of CPU registers, and this version was slightly faster.

12.4 Time and space used

Profiling—where does the time go?

A statistical profile of the `snark` version of the evaluator, running the Griss tests, revealed that the following functions were on the top of the stack 1% of the time or more:

The profile is taken by examining the stack on a regular timed interrupt. The third column indicates how many times the function was found anywhere on the stack (counting multiple times for multiple appearances) and the fourth shows this as a percentage—thus, something always present twice on the stack will show as being on the stack 200% of the time. The next two columns show the count and percentage for the procedure being on the top of the stack. Also noticeable in the profile was the system procedure `SYSTEM::DUMMY-STRUCTURE-ACCESSOR`, which implements access to `defstruct`ed data. It was called 166919089 times, but did not occupy the stack even 1% of the time, being called in such a way that it does not show in its own right (as a machine-code subroutine, rather than as a conventional compiled procedure). `gethash` (disguised as `lookup` in the listings in section **??**) occupies 11% of the time, which is much what I had expected. `snark` takes 35% of the time (including the time spent in structure accessors, because of the calling structure of the substrate system—they do not appear on the stack in their own right). `SYSTEM::ARG-IS-SAFE` is part of the implementation of Common Lisp's `apply`—it checks that the last argument (the list of further arguments to the function) is not a circular list. (This has since been made more efficient, which would affect both the timings and the profiling; but the newer version of the substrate system inlines many functions that we are interested in the appearance of on the profile, and so is not used here.)

The functions named —frame-*x*— are shadow procedures for the operators

x. These are all Lisp-like operators, which evaluate their arguments. The time for the evaluation of these arguments is included in the time for which their procedures are on the stack, since the argument evaluator is called from within the —frame-*x*— procedure. (See section **??** for details of how the —frame-*x*— procedures are defined.)

Code size and distribution

The Platypus89 system is implemented in around 5000 lines of Lisp, not including the large comments that make up the running text in sections that are presented in this thesis. Since the core of the system (as presented in section **??**) is so simple, this leaves a lot to be accounted for. Where does all the bulk go?

A large piece of the system—about one third of the total—is the shadow procedures for the base language, and another one-sixth sets up the implementation of the in-tower equivalents of these—the shadowed procedures. Thus, about half of the system is the implementation of the base language, rather than of the generic language framework.

One-sixth of the code defines the data structures used in the tower, and accessor functions and macros for them. Another one-sixth defines syntactic extensions to Lisp, and defining forms for setting up procedures, operators and shadowing. About half of the remaining one sixth is the actual core of the Platypus89 evaluator, and the rest is a variety of miscellaneous support functions.

I consider this to be reasonably concise for an implementation of a language framework and reasonable subsets of two real programming languages, although it is in part parasitic on its substrate language, Common Lisp, of which, for example, the garbage collector is needed.

Further routines needed to make a standalone system

Platypus89, as it stands, relies on the substrate Lisp system for several runtime facilities, as well as for compiling the real meta-evaluator. In particular, it needs the storage allocator, the garbage collector, and the input/output system.

One thing not present in Platypus89 is a complete type definition system, such as `defstruct` in Common Lisp. This would have to be provided—but it can, of course, be provided by an application program and reflected in.

Platypus89 also depends on the substrate system for storage management, that is, allocation and garbage collection. C-Platypus provided these itself, which was complicated by the meta-evaluator being effectively in a separate address space from the evaluator, so the garbage collector (which was of the stop-and-copy variety) had to update the meta-evaluator's pointers into the heap as a separate action from normal block movements.

12.5 Changes needed to support non Lisp-like languages

When I first wrote Platypus89, its Lisp-like base language was the only language which it ran. On adding PostScript, I changed some parts of the common meta-evaluator to make this easier. The most significant of these changes was the reversing of the order of the variables in each apparent frame of the stack, so that the first argument (from Lisp's point of view) is also the first argument from PostScript's point of view. This change was not connected at all with reflection or evaluation, but purely with mixed-language operation.

12.6 General results

Most procedural languages map readily onto reflective systems. The worst mismatch between conventional languages and a reflective interpreter is that it is natural to design reflective languages such that all calls are reflective—or rather, such that all procedures take the tower state as their argument, and return a new state—which does not map well to conventional procedure calls. With hindsight, I can see that this problem could have been tackled with towers and meta-towers.

Parameterized evaluators and meta-evaluators prove to be remarkably simple, particularly when each has a recurring part that may be isolated into a procedure in its own right. These procedures—representable as 4 and 12 lines of Lisp respectively—may be seen as refined representations of what evaluation and meta-evaluation are. Having been found, they seem intuitively very natural procedural realizations of these functions.

Debugging on a reflective system

A reflective system provides access to more information than a conventional system, and the amount of information displayed when tracing execution, getting backtraces and inspecting values turned out to be to voluminous for many purposes. However, debugging in terms of a specific language or situation can be done with more selective display of information.

The information provided is also in terms of Platypus' own model of execution, and may be at a lower level (in conventional terms, not reflective!) than an application programmer might expect. The raw data may need to be processed back into terms of the original language—which could be regarded as a form of decompilation, although not all of the data provided is program code. This decompilation will also reduce the bulk of data to be examined; for example, the insides of languages might not be shown when working in the context of a single language.

Decompilation of reified data should be quite general, and capable of producing sensible debugging printout into each language from any language wherever

the equivalent facilities are available in both. This makes it possible, for example, for someone working in Lisp to debug a PostScript program without having to know PostScript syntax. The PostScript can be displayed as Lisp, and the interpreted definitions of the PostScript operators may be used as explanations of the PostScript language.

What debugging techniques were useful?

A major problem with debugging any of the Platypus implementations was the amount, or complexity, of the data being handled. A full backtrace of the stack, with all procedure arguments being printed, was too voluminous to be manageable, although there were a few occasions when it proved necessary. The bulk was partly because of the depth of the call stack, with many calls to each of the central evaluator routines active at any one time, and partly because of the size of the printed representation of each of the objects. The most useful part of a level to print out was the top entry of its call stack—a closure—and the most useful parts of a closure were its expression (printed as its procedure expression, with a marker pointing at the sub-expression that is currently the current expression) and its *closure number*, and the closure number of its evaluator. The closure number field was originally introduced for debugging, but subsequently also used for other things; it is a number that is issued from a counter every time a closure is allocated, and so identifies closures uniquely in their printed representation, making it visible to the programmer whether a closure is `eq` to some other closure.

At some stages of the work, it was useful to make the meta-evaluator print when it was doing a level-shift, and `funcall-helper` print when it was called. More general tracing than this was too voluminous to be helpful in most cases.

Backtraces of the meta-evaluator's (Common Lisp) stack were rarely useful, as the central meta-evaluator components called each other to such a depth of indirect recursion that it was difficult to work out, and to remember, what each invocation on the stack was there for.

A breakpoint operator, which used Common Lisp's `break` function, provided a `read-eval-print` loop in the substrate language, which was useful for investigating the tower contents interactively at various points in the tower's application's execution—this was often more convenient than bulky non-interactive printouts of the tower contents.

Occasionally it was useful to set a global variable in the substrate language to the value passed through a particular point in the meta-evaluator, so the last value that had appeared in that place could be inspected interactively at a breakpoint or after a crash.

An anecdotal result on resilience

On encountering a bug in `gethash` on the substrate Lisp system, which prevented the meta-evaluator from finding the shadow for an operator, the meta-evaluator went into deep recursion to interpret it from the shadowed definition

(as distinct from the shadowing definition.) I spotted this while it was running, broke into the execution, patched around the `gethash` problem, and resumed execution from the break. The tower now shrunk back, shedding the unwanted levels of interpretation, and carried on as if nothing had been wrong.

12.7 Summary of results

Platypus has proved to be a practical interpretive system for Lisp-like languages, and promises to be able to run at speeds comparable to other, non-reflective, interpreted Lisp systems—it is already well within the factor of ten that I chose initially as a suitable limit for regarding this new interpretation technique as practical. There is certainly promise of being able to improve the performance, and I would expect to reach speeds similar to those of non-reflective, single-language interpreters. The usefulness of dynamic typing is clearly evident, and the appropriateness of implementing reflective interpretation by shadowing has been demonstrated.

Working with mixtures of two type systems, one for the substrate language and one for the tower, was particularly confusing, and this should either be avoided, or handled with careful planning, in future work in this area.

The amount of garbage generated is a potential problem, as all stack frames are, in principle, built on the heap. The problem is reduced by using the stack frames of the substrate system to store some of the data that might have otherwise needed more frames to be built for very short-term use.

Initializing the system is complicated by the number of cross-references and self-references that must be set up. It would be useful to have some program tools for generating (or gathering) the initialization code automatically from the rest of the source code.

It is possible to produce very compact versions of both the evaluator and the meta-evaluator, these being based on a pair of functions, `boojum` and `snark`, that appear to encapsulate the essence of evaluation and meta-evaluation respectively.

And I gave my heart my heart to seek and search out by wisdom
concerning all the things that are done under heaven: this sore travail
hath God given to the sons of man to be exercised with.

Ecclesiastes 1:13

Chapter 13

History and future; related work; observations

Reflective systems have grown in part from a desire to make a tidy but practical model for the meaning of interpretation, and in part from the need for powerful language and program development and debugging tools. Systems such as *SmallTalk* [**?**] have demonstrated the usefulness and flexibility of reflective interpretation, as well as its practicality, while systems such as 3-Lisp [**?**] have shown their power and expressiveness.

As the results of this thesis, and other work on reflective interpreters, have shown, implementation of reflective interpreters is now established as feasible and reasonably efficient.

13.1 Other recent developments in interpretation

Reflection is one of a group of computational techniques that have been explored recently. Others include *partial evaluation* (*mixing*) [**?**] and abstract interpretation [**?**]. All these techniques are to some extent applicable to a variety of language styles (algorithmic, functional, logical).

13.2 Possible further work

Tower implementation

Many variations are possible on the theme of Platypus-like reflective language implementations. Some of these concern implementation details, in particular the mechanism by which an object in one tower is connected to any shadow that it may have in the next meta-level. The possibilities here include

- *A shadow map with a perfect hash*, as used in C-Platypus. This is efficient, and hides the shadowing mechanism, but places strange requirements on the storage manager.
- *A shadow map with a conventional hash table*, as used in Platypus89. This is slow, but, like the system used in C-Platypus, hides the shadowing mechanism, which is important because it avoids making shadowed objects any different from other objects.
- *Pointers from shadowed objects to their shadows* which has the disadvantage of making the shadowing part of the shadowed object, rather than part of the shadowing system, but is efficient. This may be done in two ways:
  - *All objects have a shadow slot* built into their representation, which takes more space for the whole system. Objects without a shadow have a distinguished null value in this slot.
  - *Only shadowed objects have a shadow slot*, which means that they must be distinguished (perhaps by tag bits) from other objects of the same type.

  Building the shadowing into each shadowed object works for systems with flat second levels, and for those with towering second levels that do not share representations between levels, but makes it impossible to represent more than one level of shadowing directly, and would also make it impossible to have multiple meta-evaluators as suggested in section **??**.

  Another implementation technique is to replace the boring section of the tower directy with its shadow (as used in Platypus for meta-tower reflection), and have the procedures that move along the tower recognize this and hide it by returning instead copies of the repeating element of the boring section. This is not very different from the present system, but should allow slightly more efficient evaluation.

Meta-towers

Platypus89 implements one dimension of meta-towers, but does not go beyond the $\omega^2 + 1$ tower. This would be a simple experiment, but should probably be done in conjunction with making Platypus89 more efficient. Work with such a system might well await the development of a better user/programmer interface to reflection, perhaps with graphical output to draw towers.

Better integration with the substrate language

Full implementation of meta-towers would fall out naturally if towers could be called as if they seemed to be ordinary Lisp functions to the substrate Lisp system. It might be possible to represent them as Lisp closures for the substrate system, so that they would become callable.

It would be possible for a tower reflective system based on a substrate language such as Lisp (with incremental compilation) to make new primitives by compiling functions defined within the tower by calling a primitive operator that calls the native system compiler to make the shadow definition, and installing in the language and the shadow map to make it accessible as a real primitive. (The usual rules for when it is run as primitive and when it is interpreted will apply.)

Further applications of reflection: reflection with mixing

It is in principle possible to use the architecture described here to build an interpreter mixing system using some of the ideas from [**?**]. With our simple and regular program and language architecture, we can make this much simpler than the partial evaluator presented by [**?**]. This results not in a full mixer function, but in a compilation system; for each node of the expression tree, we expand the node out using the operator definition closure for that node—an inlining of the operators. There are two notable restrictions on this operation: the procedure being compiled must not change its expression reflectively; and all the operator definitions must be in the same language as each other, since the resulting closure will be in the language used to define the operators, and each closure is allowed only one language. (This second restriction may be relaxed at the cost of having to merge several languages, with automatic renaming of operators where necessary.)

If operators are provided to escape entirely from the meta-tower system down to the real substrate system (in Platypus, this is Common Lisp), it should be possible to generate real compiled code from any executable procedure (that is, one using only grounded definitions) that does not reflect into its expression or interpreter, by repeated substitution of operators as described above, until operators shadowed by the real substrate are reached. At this point, code in the real substrate language may be substituted (possibly from the shadow definitions, but it may be necessary, or at least better, to have other code for this), and the results passed to the substrate's native compiler. The result of this compilation may then be put into the appropriate shadow maps, if it was generated from a closure used as a type-evaluator or as an operator.

Using these techniques, a program-tuning system could be written, that runs a program (possibly its caller) on an interpreter that does time- or call-counting-profiling, then arranges for new primitives to be installed, if possible, to make that program run faster.

Register-based implementation

It is possible to compile reflective programs to an abstract machine code, or to real machine code using a model of execution suited to reification. A register model that might be suitable; it could have a group of working registers similar to those of the SECD machine [**?**], and reification would read from those registers, and reflection write to them. A possible set of registers is:

- `code`: a series of instructions. This corresponds to the `Control` register of the SECD machine, and the `procedure-expression` of Platypus' closures.
- `code_index`: the current index into the `code`. With the `code` register, it is equivalent to the `continuation-expression` of the current closure in Platypus.
- `evaluator`: a procedure to call to implement the current closure. Shadowing can be implemented by letting each shadowed closure be its own shadow. For such closures, the evaluator's closure is the closure's code itself, and that code is the same as the underlying machine code. This is equivalent to the `evaluator` of Platypus' closures.
- `type-evaluators`: an environment binding type names to the closures implementing the evaluation of those types.
- `values`: a series of values, used for procedure arguments and results, and for local workspace. This is equivalent to the `Stack` register of the SECD machine, and to the `values` of Platypus' closures, and, as in those closures, if a closure represents an interpreter, the closure it interprets is passed to it as its first argument.
- `language`: a map from instructions to instruction definitions. This is used to interpret each instruction in the code. For procedures that are their own shadows, the language is ignored, since self-processing procedures are implicitly written in the underlying machine language. This is equivalent to the `language` of Platypus' closures.
- `environment`: the environment in which non-local variables are to be found. This is equivalent to the `Environment` register of the SECD machine, and the `environment` of Platypus' closures.
- `dump`: the chain of saved register sets at this level of interpretation. This is equivalent to the `Dump` register of the SECD machine, and to the list of saved closures in Platypus.

This model allows reflection to take place as in Platypus, but can potentially run faster. It may be possible to compile the standard evaluator down to real machine code, using real machine registers for the abstract registers listed above. Many modern computers have enough CPU registers for this to be possible, and I propose to follow this line of research further.

Another use of compilation with reflection is for producing more efficient versions of programs that were developed using reflective interpretation.

Object-oriented programming

I have deliberately avoided the object-oriented style in developing Platypus, particularly for Platypus89. This was to emphasize the point that reflection and objects are completely separable—Platypus works in terms of *values* rather

than *objects*. This point having been made, it would be interesting to use an object-based style in implementing such towers. On a Lisp substrate, such as Common Lisp with CLOS (the Common Lisp Object System) this would provide a simple way of integrating the tower with the substrate Lisp, as towers could simply be a kind of self-evaluating object.

Efficient use of heap

Platypus generates all stack frames on the heap, which results in a high turnover of heap storage. This problem is to some extent in all reflective interpreters, as any function may have its stack frame returned as a result. It has been solved in practice in some SmallTalk implementations [**?**], which try to use real stack frames whenever possible, and convert them to heap objects as needed. Platypus could be modified to handle stack frame generation more efficiently.

A more thorough theoretical framework

The complexity theory describing meta-tower evaluation could be developed further, and could be extended to include the complexity of the shadowing mechanisms, thus describing the evaluation time for whole meta-towers.

Lexical and syntactics aspects of languages

As presented here, Platypus requires programs to be presented in the form of parse trees, represented as Lisp expressions (although it does include a modification of the Lisp parser, to read PostScript). A natural extension would be to include a facility for defining the lexical and syntactic aspects of a language to a flexible parsing mechanism, which would be able to read programs in the conventional syntax for their language, into the internal parse-tree form. The techniques for this are already well-developed [**?**] [**?**] but are not usually fitted into a Lisp-type framework.

Ideally, the parser control system should allow the listing of programs from the internal representations into the usual textual format, for debugging. In some cases it may be possible to print out in one language a program that was originally read in another, which, in a mixed-language system, allows programmers to read routines listed in a backtrace, for example, in the language with which they are most familiar. The ability to do this depends on the rôles of particular operators being recognized—which, as longs as the procedures preparing the parse trees try to use base language operators where possible, may be quite straightforward.

A systematic framework for syntactic analysis could be provided, perhaps using a mechanism like the "source code transformations" used by some compilers, including some Lisp compilers. These might also be used to effect partial compilation, and be linked closely to operator definition bodies for interpreted operators.

13.3 Higher-level tools

Reflection has been used in conjunction with inspection tools, such as the SmallTalk browser system. This is suitable for flat reflective systems, but tools working at a more abstract level may prove necessary for working with full towers and meta-towers. In particular, the amount of information available can be overwhelming, and some means of focussing interaction on particular parts of a meta-towering system may be necessary, and, with that, means for navigating around the meta-tower.

13.4 Further applications

Reflection has so far been used largely within the field of programming language research, and has not been applied significantly to other areas of Computer Science. Fields in which it could prove useful include:

- The organization of large software systems (including distributed systems), and fault analysis and circumnavigation
- Object-oriented simulation
- Explanation in deduction systems

13.5 Summary of history and future

Reflection is a young field of Computer Science and formal logic (and an older field of philosophy) and there is much yet to be explored in it. Enough is now understood to make an intuitive grasp possible, and, although always touching on the meta-physical, it is possible to explain it formally, although formalisms in common use today do not express reflective concepts very well.

In three main areas of research on the topic, two present considerable scope for further research, and the other presents scope for development and enhancement of ideas that are now established:

- *Understanding* and *describing* reflection is a large and very open field. Some forms of reflection have now been described and analyzed extensively, particularly those concerning procedural languages, but there is much more to be done, particularly on declarative languages.

  The mathematical description of infinite reflective towers, using transfinite numbers, is an intriguing field which could be developed much further, particularly with reference to the type system needed to represent such towers.

- *Implementing* reflective systems has been researched, to the extent that it is possible to build fully tower-reflective interpreters of similar speed to their nearest non-reflective equivalents [**?**]. This could be developed further, and applied more widely.

- *Application* of reflection to real logical, computational and scientific problems has scarcely begun. To date, it has been used within the area of language research, but little further.

The thing that hath been, it is that which shall be; and that which is done is that which shall be done; and there is no new thing under the sun.
Is there any thing whereof it may be said, See, this is new? it hath been already of old time, which was before us.
There is no remembrance of former things; neither shall there be any remembrance of things that are to come with those that shall come after.

Ecclesiastes 1:9-11

Chapter 14

Reflections

14.1 What did I find?

To interpret or evaluate something at one level, we find the appropriate attribute of the thing and appropriate attribute of the level, look the first up in the second, and *apply* the result of the lookup to the thing and the level.

But what does apply mean?

To interpret something in the frame of reference within which its meaning is understood, do the selection and lookup as above, but instead of applying the result, we lookup it up in the appropriate part of the higher level, and apply that. But what, in turn, does *this* "apply" mean?

Thus leans the tower of meaning ever outward. The search for how meaning is given to the level under scrutiny is infinite. However, to give meaning to a level constructed, implemented and understood by the level concerned is one easy step. This is because meaning, at this level of giving meaning, is in the context only of the provider, and so can be given entirely by the provider, and is valid to that provider.

So now we have two arguments; one to demonstrate that for a level to receive its meaning into itself, it must first receive an infinite number of other levels, and so cannot receive its grounded meaning and yet terminate; and the other to demonstrate that to give meaning to a level does not make an infinitely distant connexion, but an immediately proximal one.

How is it that both of these can be true?

One way to see a solution to this problem is to see that the meta-evaluator is an ordinary, non-reflective program, running on an ordinary computing system, whereas the system within the tower is running on an illusory reflective computer, provided with the illusion of an infinite number of levels of interpretation.

The meta-evaluator can realize as many levels as the programs in the tower require (within the constraint of memory space). The apparent infinitude of the tower system is simply generated by a perfectly ordinary piece of compiled Lisp code.

14.2 Human reflection

It is natural to try to apply some of the ideas of procedural reflection to our own human thought processes. For example, learning a new manual skill may be seen as installing a new ability into our store of knowledge and reactions. We reify our thought processes, and attempt to reify our intuitions, when producing a rationâle for a decision. This area of reflection touches on very general philosphical topics [**?**] [**?**] [**?**].

As well as the reification and reflection of 'natural' mental processes, we perform rapid shifting of information up and down between our internal reasoning activities and external media such as paper, computers, counters, abaci, coins.... We write and read information on external media not only to transfer it between people and for long-term archival, but also to assist our short-term memories in areas in which the brain, with conventional training, is generally weak—performing a long division calculation on paper is an obvious example. The amount of information to remember is not significant on the scale of the brain's overall memory capacity (whatever that may be), and the length of time for which it must be remembered is only seconds or minutes, but for most people the workspace storage for this must be external rather than internal.

However, we can devise and learn methods for performing mental arithmetic, perhaps transforming an algorithm for externally assisted calculation into something easier for a particular individual to work with.

How can we describe such external storage in terms of the reflective devices described in this thesis? We reify things from the conceptual form in which we manipulate them, and reflect them into the state of the storage machine 'on its behalf'—or do we interact with the storage in a less mentalistic manner? We read what is on the storage medium, perhaps reifying it in the process, and reflect it back in to our brain's internal storage. Here, the human agent is the interpreter that implements the external device's semantic storage abilities; but it is also the user of the external storage, so it could be seen as a co-towering relationship.

14.3 Reflective changes through other artifices

Finding it difficult to influence mood and mental activity directly by thinking about them, people have accumulated a collection of ways of influencing themselves; for example, trying to become more alert and wakeful simply by deciding to do so often proves ineffective; whereas implementing the intention by drinking coffee is more reliable. This is, in one sense, an instance of reflective control

of our own mental machinery.

14.4 Tower reflective interpretation and other models of computation

Tower-reflective interpretation is just one of many possible models of computation. It does not make computable anything that was not already computable (that is to say, a reflective interpreter cannot do anything beyond what a Turing machine can do) but, intuitively speaking, it seems to bring in fresh possibilities for describing action, language and meaning. What does it introduce that is new, and does it make difficult anything that other systems make easier?

Tower-reflective interpretation takes an unconventional approach to defining the meanings of programs. Whereas other systems offer a purportedly absolute definition of meaning using a substrate that must be taken as representing the framework (world-view) in which the program is taken to have meaning, reflective interpretation aims to bring the framework into the structural field on which the program operates, thus making the definition relative, and at the same time enriching the structural field. Or does it remove the absolute quality of the definition? Is such an absolute quality ever really present within any computational system? Gödel's theorem states that no mathematical system can ever possibly describe itself. Perhaps one of the best ways of comparing the power of descriptive systems is to examine their strengths at the very boundaries (event-horizons?) of self-description. But what is a strength here, and how can we measure it? And how do we examine something that approaches self-description? (On a more philosophical level, we may ask whether this can only be answered absolutely by a wholly self-descriptive agent—and if so, do we qualify ourselves, and by what means and in what frame of reference do we allow ourselves that qualification?)

14.5 Summary of reflections

A reasonably efficient reflective evaluation system can emulate an infinite metatower of evaluation, in remarkably few lines of Lisp. The most central of these lines may be regarded as a refined form of generalized or parameterized evaluator.

Perhaps somewhat fancifully, parallels may be drawn with non-computational procedural activities, such as deliberation, and learning, in people.

The reflective approach to language definition avoids the conventional route of definition in terms of something outside the the system, after the acknowledgement that no system—neither computational nor mathematical—nor for that matter those based on any other linguistic notation—can describe itself completely. An outside reference—visible to the same observer—must always be present, and we allow such a reference to be arbitrary, rather than from some

denotational framework from mathematics, for which in turn the same problem of an outside reference also occurs.

Who hath ascended into heaven, or descended? who hath gathered the wind in his fists? who hath bound the waters in a garment? who hath established all the ends of the earth? what is his name, and what is his son's name, if thou canst tell?

Proverbs 30:4

Chapter 15

Recapitulation

This chapter is composed from the summaries that appear at the end of each chapter.

15.1 Survey

The mechanisms of program interpretation may be analyzed into several areas, some of which are currently active research areas. These include program transformation, partial evaluation, reflection, and mixed-language programming. This thesis concentrates on the latter two.

Reflection—the causal link between actions of a program and its state, text, behaviour and environment—combines ideas from interpretation theory, logic, mathematical philosophy, linguistics, compilation, abstraction and other fields of Computer Science. It also contributes new techniques which may be used in these, and other, fields.

Mixed-language working is already common practice, but has not been formalized. Its existing use argues strongly for its usefulness, and the limitations on its present use, and its current haphazardness argue for further development of the ideas underlying it.

In common between these fields is the systematic definition of programming language interpreters, and the idea of provision of meaning for a value in terms of the context surrounding it (including its interpreter).

15.2 Introduction

Reflective techniques are based on two facilities: *reification*, by which a program may examine its own code, state and interpreter; and *reflection*, by which it may modify any of these. The connection that reflection makes between a programs actions and its behaviour is causal in nature: modifications that a program makes to its interpreter may cause changes in the way that the program is interpreted.

Some programming languages, not normally regarded as reflective, provide a limited range of reflective operations, such as access to the parameter list of a procedure call. However, in a fully reflective program interpretation system, all the features of any programming language can be implemented through reflective programming in the program, thus removing the distinguished status from the interpreter of a language, and making it equivalent to any other program in the system.

There are two kinds of reflection: *simple reflection* and *tower reflection*. Simple reflection provides a program with access to its own code and state and interpreter. Tower reflection also provides it with access to its means of interpretation—that is, the mechanism by which a program is related to its interpreter, and thence to its interpreter's interpreter, and so forth. Tower reflection is more general, and, not having an arbitrary stop after the first level, is a more regular conceptual structure. Thus it is a more powerful tool for reasoning about intensional reflection and about interpretation.

Although tower reflection deals with infinite structures, it is possible to implement it with finite constructions. This thesis explores the infinite towers and their finite implementations, and investigates whether an interpretive programming system built this way can be made reasonably efficient, compared with conventional, non-reflective interpreters.

In this thesis, we develop a reflective tower implementation, called Platypus, and use it to demonstrate many of the points discussed.

15.3 Closures

At each level in our infinite tower, there are one or more procedures, of which at any one time one will be active, and some (possibly none) will be on a list of saved procedure evaluations to be returned to.

In common with many other language systems, we represent procedures by *closures*, each containing the code of the procedure and any context that must be carried with that code to interpret it.

To make explicit the interpreter of a procedure, we *close over* the interpreter when constructing the closure of the procedure, and thence the rest of the tower of which that evaluator is the lower end, thus making it contain the whole of the context in which the procedure is interpreted.

A closure constructed this way we call an *interpretive closure*, since it encloses all the information needed for interpreting a procedure.

We use interpretive closures as the building block for constructing reflective interpretive towers. Each level of a tower contains a closure (actually, a stack (or list) of closures). The interpreter of an interpretive closure is also an interpretive closure, as are the operator definitions. Closures are also used to represent procedures available for calling. When a procedure is called, its closure is instantiated by copying in onto the top of the stack, and filling in fields that come from other parts of the level and the tower (such as the dynamic environment within which it was called.)

15.4 Links between levels

The tower of interpreters is made up from links between adjacent levels of interpretation. Reification and reflection are complementary operations, and they use complementary links between tower levels. Since both links are set up by parts of the calling mechanism, they always occur in pairs. These links make a bidirectional chain throughout the tower.

Since a tower is an infinite structure, any computation involving all levels of it cannot terminate. An application that uses reflection and does terminate must therefore make reference to only a finite part of the tower.

Therefore, the infinite tower may be represented finitely, and it is possible to reason about how many steps of interpretation are needed at one level to implement each step of interpretation at a lower level.

To make the finite representation of an infinite string of identical levels, we make the highest part of the tower into a circle, in which the same level occurs again and again. Whenever an interpreter tries to reify the level that makes up this circle, a hidden meta-evaluator makes a copy of the level in the circle, and passes that *copy* out as the reification of the level in the circle. The application can then modify the level it has been given, without upsetting the level that is still in the circle.

Thus, the infinite part of the tower is stored compactly as a circle, which is unrolled on demand to produce an infinite supply of identical levels. To the application, this is indistinguishable from there being a real infinite chain of levels at the top of the tower, instead of the circle and the unrolling mechanism.

The idea of the reflective tower and the means of reflection may also be applied to towers of towers—that is, meta-towers. Meta-towers might at first seem complicated to reason about, but an understanding of them brings a readier understanding of ordinary towers.

The design of the meta-evaluator, and particularly the mechanisms for detecting the need to unroll a new level from the circle and for unrolling the levels, are a major new development of this thesis.

15.5 Languages in closures

As well as being a reflective system, our system is a mixed-language one, making reified languages into part of the evaluation data that is available for manipulation by application programs running on the system.

To make the language a variable part of the context of a closure, we divide the interpreter into two parts, the *evaluator*, which is language-independent, and the *language*.

To do this, we need an abstraction for languages. The abstraction must be general enough to handle most languages reasonably well, and to handle all languages to some extent. Such an abstraction can be devised only in conjunction with an abstraction for the programs in the languages. For the programs, we choose to use parse trees, with each node being identified by its *operator* such

as `if` or `+`, and for the languages, we use environments binding operator names to operator definitions.

This abstraction makes it easy to add new operators to a language, and also keeps separate the general evaluator, thus making it possible to redefine the evaluator independently from the language.

By making the `language` (the environment binding operator names to operators) of each tower level an explicit part of that level, we extend tower reflection from being a tool for reasoning about interpretation of programs to also being one for reasoning about languages and their interpretation.

15.6 The standard evaluator

The evaluator is the kingpin of a level. It links the parts of the level to each other by using them to evaluate the level, and links adjacent tower levels by making it possible to shift data from one level to another. Its form is tied to the form of the tower level type.

Each evaluable infinite tower (in the scope of this thesis) eventually reaches a repetitive stage (termed the *boring* stage by [**?**]), the procedure running in each of these identical levels being known as the *standard evaluator* level.

The standard evaluator is a fairly small and skeletal procedure, needing, in its most refined form, only six operators in its definition, most of those being for structure handling. The rest of the evaluator is defined separately, partly in individual operator definitions, and partly in some general evaluator procedures that may be called by operator definitions. These general procedures are used for evaluating arguments to operators. To do this, they invoke the evaluator and language mechanism to do the evaluation in the appropriate context.

The concise form of the standard evaluator may be seen as a distillation of the essential matter of a programming language interpreter (independently of any particular language). This may be refined further to a procedure which implements several of the main parts of the evaluator. This procedure consists of three procedure calls and one environment lookup.

15.7 Types, abstraction, and representation

The ideas behind the towers' type system are important in understanding tower reflection. Types are an essential part of the way we represent values, and the mapping from one tower level to the next is a representation of a value in one system by a value in another. The basis for representing values in a computer is Gödelization, in which digits in numbers denote words in a language.

The type system we use must allow the representation of procedures and of procedural evaluation, as well as the representation of the application's problem domain. It must also be possible to represent the infinite towers and meta-towers used in reflective evaluation.

The system must provide operations on types concerned with reflection (that is, types for objects representing parts of the tower) as well as for the types of objects normally handled by an interpreter. We divide types into two kinds: simple and compound.

A few types are of particular importance in a reflective tower. Closures are the central type. Other important types include expressions, environments and value lists.

As information is moved between levels, its representation may be changed, although in Platypus it is not changed. The meaning of the same information may be different at different levels even when the representation is the same.

Although the meaning and representation of information does not normally change between levels of a normal tower, it may well have to change in going between the tower and the meta-evaluator that implements the tower.

15.8 Building languages with reflection

Our mixed-language interpretation is designed to allow many languages to be built on top of it. Languages which can be converted readily to a procedural form are most suitable for this: procedural and functional languages are easiest, declarative and rule-based languages are harder.

We assume handling of such types as numbers to be made available underneath the implementation of reflection. Reflection does not help to describe these, anyway, so nothing would be gained were it possible to include them in the reflective system.

Most conventional language features map readily onto a reflective mixed-language architecture. Occasionally there is a mismatch, such as it being natural to try to make all calls reflective (which builds a tower level for each procedure call).

Using jumpy reflectors (that assign to parts of the state, without saving the old values on a stack) to change specific parts of a tower allows very natural implementation of many common language features such as jumps, calls and assignments.

Reflective features may be used to group together parts of a system, such as all the operators of a language, for interpretation in a particular way.

As well as any languages implemented on top of the reflective system, there is a base language which provides reflective facilities and some simple flow control and calling operators. This is sufficient for running the rest of the system, so long as all parts of the system are connected with integrity to the base language.

Reflection allows new features to be added to conventional languages, including extreme examples such as a non-local exit that goes right out of several levels of interpretation.

Procedure calls are to some extent built into the evaluator, but other features are not so much so. Our procedure calling is naturally call-by-name, but call-by-value may be implemented easily on top of this; such a facility is provided in a form that is useful to many language implementations.

15.9 The base language

A language for use with the standard interpreter in the boring section of the tower must be powerful enough to support both the standard interpreter and the procedures that will run on it, which will typically be operators for other languages.

The implementation of the base language has two parts: the operators themselves, and their shadows, which are run at the next meta-level in the tower. (The last meta-tower is run in the substrate language on which the whole reflective system is built, and it is there that all operator definitions are eventually evaluated.)

The language should provide operations typically needed by interpreters, and those needed for reification and reflection. It is also desirable that the base language be reasonably expressive.

As well as the fundamental reifiers and reflectors, it is convenient to provide some jumpy reflectors that assign only part of the state; these are not only more efficient, but also more expressive of many common language features that they may be used to model.

In practice, we provide many more operators in the base language than are strictly necessary. ([**?**] explains how to work out which operators are necessary.)

Reflective operators

Operators for reflection may be added to an existing language. With our model for mixed-language interpretation, the same operators will work for any language.

Reflective operators (reifiers and reflectors) are of two kinds, *jumpy* which move data between program-as-agent and program-as-subject without automatically creating new levels of interpretation, and *pushy* which create new levels either providing data from the program-as-subject or using it to create a new (or modified) program-as-subject. Jumpy operators are more primitive than pushy operators, in that (on a conventional architecture) they may be used to implement pushy operators, whereas, within one level of interpretation, pushy operators may not be used as the primitive on which jumpy operators may be built (other than by considerable wasted work).

One form of reflective operator is the *grand reflective operator* which reifies or reflects the entire state of the system. However, it is more efficient, as well as often more convenient, to reflect into just the part of the state required, and so reflectors that set only specific parts of the state are also worth providing in a practical system.

Reification of programs is homogenous between languages. The same reifiers (and reflectors) may be used in any language, and the values returned have closed into them all the linguistic information needed to understand the value in any way that might be required.

15.10 A model for the meta-evaluator

The boring part of each tower is not really evaluated, but its evaluation is mimicked by the *meta-evaluator* of that tower. The meta-evaluator has two rôles: it implements finitely the infinite tower, and it stands in for any number of levels.

To do this, it has to be able to absorb level shifts, to stop them going any further along the tower. It does this by realizing new levels, (and abandoning old ones), on demand, when it must extend the non-boring part of the tower, and shadowing things itself when still on the boring section.

The code of the meta-evaluator can be similar to that of the standard evaluator, with the addition of some level-shifting code that would not, within the tower, be allowed to exist within a single level, because it is capable of generating (realizing) levels[1].

The meta-evaluator is alongside the tower (from the tower's point of view) and both alongside and above the tower (from the meta-evaluator's point of view).

The meta-evaluator runs beside the lowest tower level that it can, that is, one level above the highest one that is not mimicked by the meta-evaluator. It follows this boundary by climbing up to new levels as it realizes them, and climbing back off them when they are no longer needed.

There are two approaches to how the meta-evaluator should view data within the tower, in terms of how each type of data is represented, and whether each type appears as the same type inside and outside the tower, or as distinct types. In this thesis, we hold the data in the same form in both, and hence, for example, stack frames do not have to be re-encoded when reified or reflected.

The meta-evaluator must have a map from *shadowed* operators in the tower to *shadowing* operators in the meta-evaluator. In a system with only one dimension to the tower, this map must be visible to the meta-evaluator but not to the tower.

If the meta-evaluator implements the storage system of the tower, it must also have a map from distinguished objects within the tower to variables in the meta-evaluator. If the meta-evaluator and the tower share a storage system, such a map is not needed.

15.11 An implementation of the meta-evaluator

Platypus is a trial implementation of a reflective tower-based evaluation system in which data representations inside and outside the tower are the same. It has been through two broadly similar implementation generations, one in C and one in Common Lisp.

Dynamic typing is used throughout the system, both inside and outside the tower. All objects in the tower are kept in a storage heap; in the C implementation this is scavenged by a stop-and-copy garbage collector, which must pay

[1] Within the tower, each procedure may be in only one level at a time.

attention to updating variables of the meta-evaluator as it moves the things to which they point.

While several meta-evaluator variables point into the heap, it is not possible for a program in the tower to find from an object in the heap what it corresponds to in the meta-evaluator (although a meta-evaluator that makes this information available could be written).

The meta-interpreter must contain:

- a meta-evaluator that mimics (shadows) the standard evaluator as well as generating levels on demand;
- shadow operator definitions shadowing some of the operators in the tower's base language;
- and an argument evaluator for shadowing operators to use.

These are all fairly similar to their equivalents within the tower.

The meta-evaluator may be reduced to a very concise form built around one function, which is appropriately very similar to the corresponding function for the concise form of the standard internal evaluator for a tower. This function consists of four procedure calls and two environment lookups.

The code for all the shadow operator definitions has much in common; in particular, all operators that must evaluate all their arguments independently share an argument evaluator mechanism. Such operators are built around primitive procedures in the implementation language, by a macro-preprocessor, that takes functions in the implementation language, and produces wrapper functions for use as the shadow operators.

15.12 Results

Platypus has proved to be a practical interpretive system for Lisp-like languages, and promises to be able to run at speeds comparable to other, non-reflective, interpreted Lisp systems—it is already well within the factor of ten that I chose initially as a suitable limit for regarding this new interpretation technique as practical. There is certainly promise of being able to improve the performance, and I would expect to reach speeds similar to those of non-reflective, single-language interpreters. The usefulness of dynamic typing is clearly evident, and the appropriateness of implementing reflective interpretation by shadowing has been demonstrated.

Working with mixtures of two type systems, one for the substrate language and one for the tower, was particularly confusing, and this should either be avoided, or handled with careful planning, in future work in this area.

The amount of garbage generated is a potential problem, as all stack frames are, in principle, built on the heap. The problem is reduced by using the stack frames of the substrate system to store some of the data that might have otherwise needed more frames to be built for very short-term use.

Initializing the system is complicated by the number of cross-references and self-references that must be set up. It would be useful to have some program tools for generating (or gathering) the initialization code automatically from the rest of the source code.

It is possible to produce very compact versions of both the evaluator and the meta-evaluator, these being based on a pair of functions, `boojum` and `snark`, that appear to encapsulate the essence of evaluation and meta-evaluation respectively.

15.13 History and future

Reflection is a young field of Computer Science and formal logic (and an older field of philosophy) and there is much yet to be explored in it. Enough is now understood to make an intuitive grasp possible, and, although always touching on the meta-physical, it is possible to explain it formally, although formalisms in common use today do not express reflective concepts very well.

In three main areas of research on the topic, two present considerable scope for further research, and the other presents scope for development and enhancement of ideas that are now established:

- *Understanding* and *describing* reflection is a large and very open field. Some forms of reflection have now been described and analyzed extensively, particularly those concerning procedural languages, but there is much more to be done, particularly on declarative languages.

  The mathematical description of infinite reflective towers, using transfinite numbers, is an intriguing field which could be developed much further, particularly with reference to the type system needed to represent such towers.

- *Implementing* reflective systems has been researched, to the extent that it is possible to build fully tower-reflective interpreters of similar speed to their nearest non-reflective equivalents [**?**]. This could be developed further, and applied more widely.

- *Application* of reflection to real logical, computational and scientific problems has scarcely begun. To date, it has been used within the area of language research, but little further.

15.14 Reflections

A reasonably efficient reflective evaluation system can emulate an infinite meta-tower of evaluation, in remarkably few lines of Lisp. The most central of these lines may be regarded as a refined form of generalized or parameterized evaluator.

Perhaps somewhat fancifully, parallels may be drawn with non-computational procedural activities, such as deliberation, and learning, in people.

The reflective approach to language definition avoids the conventional route of definition in terms of something outside the the system, after the acknowledgement that no system—neither computational nor mathematical—nor for that matter those based on any other linguistic notation—can describe itself completely. An outside reference—visible to the same observer—must always be present, and we allow such a reference to be arbitrary, rather than from some denotational framework from mathematics, for which in turn the same problem of an outside reference also occurs.

15.15 Summary

Reflection and reification let programs access themselves and their interpreters as data. This access is causal in nature: modifications a program makes to its interpreter changes how the program runs.

Simple reflection lets a program access its code and state and interpreter. *Tower reflection* also lets it access its means of interpretation—that is, the mechanism by which a program is related to its interpreter, and thence to its interpreter's interpreter, etcetera.

We *close over* the interpreter when constructing the closure of the procedure, and thence the tower of interpreters starting there, making it contain the whole context in which the procedure is interpreted.

By making the *language* of each tower level an explicit part of that level—a new contribution to this field — we extend reflection from being a tool for reasoning about interpretation of programs to being one for reasoning about language interpretation.

The *meta-evaluator* implementing a tower has two rôles: it implements finitely the infinite tower, and stands in for any number of levels.

To do this, it must absorb level shifts, by realizing new levels, and abandoning old ones, on demand.

We use a *shadow map* from *shadowed* operators in the tower to *shadowing* operators in the meta-evaluator.

The meta-evaluator can itself be a program executed by a tower of interpreters, starting a *meta-tower*—an original development.

Platypus is an implementation of a reflective tower in which data representations inside and outside the tower are identical. It has two similar implementations, in C and in Lisp.

Platypus has proved to be a practical interpreter for Lisp-like languages, and promises to be able to run at speeds comparable to non-reflective interpreted systems. The appropriateness of implementing reflective interpretation by shadowing has been demonstrated.

Scheme, now, *feels* like Algol-60 (the world's sweetest version of Fortran), and I'd say that feel is more important than look.

Reuben_Bert_Mayo@cup.portal.com

Bibliography

[Abelson and Sussman] H. Abelson and G. Sussman with J. Sussman: *Structure and Interpretation of Computer Programs*, MIT Press, 1985

[Agha] G. Agha: *Actors—a model of concurrent computation in distributed systems*, The MIT Press, 1986.

[ANSI C] ANSI X3J11: *Draft proposed American National Standard for Information Systems—Programming Language C*, ANSI document X3J11/88-158 (December 1988)

[Barendregt] H. Barendregt: *The Lambda Calculus: Its syntax and semantics*, Studies in Logic and the Foundations of Mathematics, Volume 103, North-Holland (1981)

[Batali] J. Batali: *Computational Introspection*, MIT AI Memo No. 701 (February 1983)

[Ben-Ari] M. Ben-Ari: *Principles of Concurrent Programming*, Prentice-Hall, 1982

[Cardelli and Wagner] L. Cardelli and P. Wagner: *On Understanding Types, Data Abstraction and Polymorphism*, Technical Report CS-85-14, Brown University, Department of Computer Science, 1985

[Carroll] L. Carroll:

[Clocksin and Mellish] W. Clocksin and C. Mellish: *Programming in Prolog*, Springer-Verlag (1981)

[Danvy] O. Danvy: *Across the bridge between Reflection and Partial Evaluation*, Proceedings of the WorkShop on Partial Evaluation and

Mixed Computation, Dines Bjørner, Andrei P. Ershov and Neil D. Jones (eds), North-Holland, Gl. Avernes, Denmark (October 1987)

[Danvy and Malmkjær 88] O. Danvy and K. Malmkjær: *Intensions and extensions in a reflective tower*, Proceedings of the 1988 ACM conference on Lisp and Functional Programming, Snowbird, Utah (July 1988)

[Danvy and Malmkjær 8?] O. Danvy and K. Malmkjær: *An approach for formalizing computational reflection*, unpublished, Köbenhavns Universitet, 1988

[Doyle] J. Doyle: *A Model for Deliberation, Action, and Introspection*, MIT AI Laboratory Technical Report 581.

[Escher] M. Escher: *The Graphic Work of M. C. Escher*, Meredith Press, 1967

[Futamura] Y. Futamura: *Partial evaluation of computation process – an approach to a compiler-compiler*, Systems, Computers and Control Vol 2 No 5 pp45-50 (1971)

[Gabriel] R. Gabriel: *Performance and evaluation of Lisp systems*, The MIT Press, 1985.

[Goldberg and Robson] A. Goldberg, D. Robson: *SmallTalk-80: The Language and its implementation*, Addison-Wesley (1983)

[Gödel] K. Gödel: *Über Formal Unentscheidbare Sätze def Principia Mathematic und Verwandter Systeme, I*, Monatshefte für Mathematik und Physik, 38, 1931

[Hardy] S. Hardy: *The Poplog programming system*, University of Sussex Cognitive Science Research Paper No. 3, 1982

[Harpaz] Y. Harpaz: *Migrating Common Lisp*, Harlequin Ltd, Chameleon report TR-87-46, 1987

[Haynes *et al* 84] C. Haynes, D. Friedman, M. Wand: *Continuations and Coroutines*, CACM POPL 84?

[Hopkins] D. Hopkins: *A PostScript interpreter written in PostScript*, Usenet article in `comp.lang.postscript` (`don@BRILLIG.UMD.EDU`), 27 Aug 89.

[Halstead] R. Halstead, *Multilisp: A Language for Concurrent Symbolic Computation*, ACM TOPLAS **7**, October 1985, pp501–538.

[Johnson] S. Johnson: *Yacc: Yet Another Compiler-Compiler*, Unix manual Volume 2B (July 1978)

[Jackson] J. Jackson, Harlequin Ltd: *private communication*

[Jones, Sestoft and Søndergaard] A. Jones, Sestoft and Søndergaard: *MIX: a self-applicable partial evaluator for experiments in compiler generation*, DIKU, University of Copenhagen (June 1987)

[Kranz *et al*] D. Kranz, R. Halstead & E. Mohr, *Mul-T: A High-Performance Parallel Lisp*, in Proceedings of SIGPLAN '89 Conference on Programming Language Design and Implementation, published by ACM Press, New York, pp81–90, 1989

[Köhlbecker 86] E. Köhlbecker: *Syntactic extensions in the Programming Language Lisp*, PhD dissertation, Indiana University (August 1986)

[Köhlbecker *et al* 86] E. Köhlbecker, D. Friedman, M. Felleisen, B. Duba: *Hygienic Macro Expansion*, Proceedings of the 1986 ACM Conference on Lisp and Functional Programming (1986)

[Köhlbecker and Wand] E. Köhlbecker and M. Wand: *Macro-by-example: Deriving Syntactic Transformations from their Specifications*, Conference Record of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (Munich, January, 1987)

[Landin] P. Landin: *The Mechanical Evaluation of Expressions*, Computer Journal, Volume 6, pp308-320, 1963

[Lang and Pearlmutter] K. Lang and B. Pearlmutter: *Oaklisp: an object-oriented dialect of Scheme*, Lisp and Symbolic Computation, Volume 1, Number 1, 1988

[Lesk and Schmidt] M. Lesk and E. Schmidt: *Lex—A Lexical Analyzer Generator*, Unix manual Volume 2B, July 1975

[Loeliger] R. Loeliger: *Threaded Interpretive Languages*, BYTE Publications, 1981.

[Magritte] R. Magritte: *The Two Mysteries*, 1966

[Miller] J. Miller, *MultiScheme: A Parallel Processing System*, PhD. thesis, Massachussetts Institute of Technology, 1987.

[Moss] J. Moss: *Managing Stack Frames in SmallTalk*, Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, St. Paul, Minnesota

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

[Occam] Inmos Ltd: *Occam programming manual*, Prentice-Hall, 1984

[Osborne] R. Osborne: *Speculative Computation in MultiLisp*, published in the Proceedings of the 1990 ACCM Conference on Lisp and Functional Programming, ACM Press, ACM, New York, pp198–208.

[Padget] J. Padget: *The ecology of Lisp*, PhD thesis, Bath University (1984)

[Padget and ffitch] J. Padget and J. ffitch: *Closurize and Concentrate*, School of Mathematical Sciences, University of Bath

[Perdue and Waters] C. Perdue and R. Waters: *Generators and Gatherers*, in *Common Lisp the Language, second edition*, edited by G. Steele, Digital Press, 1990

[PostScript] Adobe Systems Incorporated: *PostScript Language Reference Manual*, Addison-Wesley, 1985.

[Rees and Clinger] Rees and W. Clinger (eds): *Revised Revised Revised Report on the algorithmic language Scheme*, Sigplan notices Vol 21 No 12 pp37-39 (December 1986)

[Ryle] G. Ryle: *The Concept of Mind*, Peregrine Books, 1949.

[Shivers] O. Shivers: *Partial evaluation in Scheme*, SIGPLAN '88 conference Programming Language Design and Implementation, published by ACM Press, New York, 1988

[Smith and des Rivières 84a] B. Smith and J. des Rivières: *Reflection and semantics in Lisp*, Conference recordings of 14th Annual ACM Symposium on Principles of Programming Languages, pp23-35, Salt Lake City, Utah (January 1984)

[Smith and des Rivières 84b] B. Smith and J. des Rivières: *The implementation of procedurally reflective languages*, Conference record of the 1984 ACM Symposium on Lisp and Functional Programming pp341-347, Austin, Texas (August 1984)

[Smith] B. Smith: *Reflection and semantics in a procedural language*, PhD thesis MIT/LCS/TR-272, MIT, Cambridge, Massachussetts (January 1982). Summarized in Batali, Computational Introspection, MIT AI-Memo 701.

[Steele 84] G. Steele: *Common Lisp the Language*, Digital Press (1984)

[Steele 90] G. Steele: *Common Lisp the Language, second edition* Digital Press, 1990,

[Turing] A. Turing: *On computable numbers, with an application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society, Ser. 2, 42 (1936-37) pp230-265. Corrections, ibid (1937), pp544-546. Reprinted in *The Undecidable*, edited by Martin Davies, Raven Press, Hewlett, NY, 1965, pp115-154

[Wand and Friedman] M. Wand and D. Friedman: *The Mystery of the Tower revealed: a non-reflective description of the reflective tower*, Vol 1 No 1 pp11-38 of the International Journal of Lisp and Symbolic Computation

[Winograd] T. Winograd: *Understanding Natural Language*, Academic Press, New York, 1972

[Warren] D. Warren: *An Improved Prolog Implementation which Optimizes Tail Recursion*, Research Paper 156, Department of Artificial Intelligence, University of Edinburgh, 1980

[Watson and Tillotson] A. Watson and M. Tillotson: *Efficient Decompilation from Machine Code*, Harlequin Ltd, Chameleon report TR-89-69, 1989

As a tailpiece, the Escher picture used for the PostScript timing tests is presented here: