Computability over Abstract Data Types

Patrick Byers B.A.

Submitted for the degree of Doctor of Philosophy

Department of Mathematics
University of Surrey
September 1990

© Patrick Byers 1990
This thesis extends the study of the notion of termination equivalence of abstract structures first proposed by Kfoury. The connection with abstract data types (ADTs) is made by demonstrating that many kinds of equivalence between ADT implementations are in fact instances of termination equivalence between their underlying algebras. The results in the thesis extend the original work in two directions.

The first is to consider how the termination equivalence of structures is dependent upon the choice of programming formalism. The termination equivalences for all of the common classes of programs and for some new classes of non-computable schemes are studied, and their relative strengths are established.

The other direction is a study of a congruence property of equivalences relative to the join or addition datatype building operation. We decide which of the termination equivalences are congruences for all structures and for all computable structures, and for those equivalences which are not, we characterise those congruences closest to them (both stronger and weaker).

These programmes of work involved the use of constructions and properties of structures relating to program termination which are of interest in themselves. These are examined and are used to prove some general results about the relative strengths of termination equivalences.
## Chapter 2 – Preliminary Definitions and Results

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Many-sorted languages and structures</td>
<td>16</td>
</tr>
<tr>
<td>2.1.1 Languages</td>
<td>16</td>
</tr>
<tr>
<td>2.1.2 Structures $\mathcal{M}$ of a language $L$</td>
<td>17</td>
</tr>
<tr>
<td>2.1.3 Extensions</td>
<td>18</td>
</tr>
<tr>
<td>2.1.4 Generated substructures</td>
<td>19</td>
</tr>
<tr>
<td>2.1.5 Classes of structures</td>
<td>19</td>
</tr>
<tr>
<td>2.1.6 Homomorphism and isomorphism</td>
<td>20</td>
</tr>
<tr>
<td>2.2 Operations on structures</td>
<td>21</td>
</tr>
<tr>
<td>2.2.1 The join operation</td>
<td>21</td>
</tr>
<tr>
<td>2.2.2 The disjoint union operation</td>
<td>22</td>
</tr>
<tr>
<td>2.3 Models of computability and definability</td>
<td>26</td>
</tr>
<tr>
<td>2.3.1 Machine based models</td>
<td>26</td>
</tr>
<tr>
<td>2.3.2 Schema based models</td>
<td>29</td>
</tr>
<tr>
<td>2.3.3 Effective definitional schemes</td>
<td>30</td>
</tr>
<tr>
<td>2.3.4 Remarks on terminology</td>
<td>32</td>
</tr>
<tr>
<td>2.3.5 Equivalence of models of computation</td>
<td>33</td>
</tr>
<tr>
<td>2.4 Generalising notions of computability</td>
<td>35</td>
</tr>
<tr>
<td>2.4.1 Generalised schemes</td>
<td>35</td>
</tr>
<tr>
<td>2.4.2 Operations on schemes</td>
<td>37</td>
</tr>
<tr>
<td>2.5 The unwind property</td>
<td>39</td>
</tr>
<tr>
<td>2.6 Program properties as statements in infinitary logic</td>
<td>43</td>
</tr>
<tr>
<td>2.7 Computable structures</td>
<td>45</td>
</tr>
<tr>
<td>2.8 The pebble game</td>
<td>53</td>
</tr>
</tbody>
</table>
### Chapter 3 - Algorithmic Equivalences

3.1 Observations and equivalences 58

3.2 Aspects of program behaviour and termination 61
   3.2.1 Structures with the same domain 62
   3.2.2 Minimal structures 62
   3.2.3 Other structures 63
   3.2.4 Expanding structures 69

3.3 Identifiability 72

3.4 Distinguishability 77

3.5 Specification equivalence 84

### Chapter 4 - Types and Type-Based Equivalences

4.1 Types 89

4.2 Other type equivalences 99

4.3 The unwind property again 102

4.4 Saturation 111
   4.4.1 Model-theoretic saturation 111
   4.4.2 Completeness 117
   4.4.3 Relative saturation 120

4.5 Computable structures again 125
CHAPTER 5 – SEPARATION AND CONGRUENCE RESULTS

5.1 Separation
5.1.1 Adding iteration
5.1.2 Adding control capability
5.1.3 Adding storage space
5.1.4 Adding operation symbols
5.1.5 Introducing non-recursiveness
5.1.6 Elementary equivalence
5.1.7 Specification equivalence
5.1.8 Isomorphism

5.2 Inner congruences

5.3 Outer congruences

5.4 Congruences for effective structures

DISCUSSION AND CONCLUSIONS

BIBLIOGRAPHY

INDEX OF DEFINITIONS
ACKNOWLEDGEMENTS

I would like to thank my supervisor, David Pitt, for his constant interest and encouragement in the course of this work, and for many stimulating discussions and suggestions; I am especially grateful for the thoroughness and efficiency with which he read drafts of my work. I would also like to thank Prof. John Tucker for his invaluable suggestions and comments.

I am grateful to Steve Schuman, Jason Cozens and Paul Krause and other members of the department for helpful discussions at various stages of my work and also to Mark Ellis for technical support of all computing activities; they all made life at Surrey enjoyable and interesting.

On the personal side, Liz, Tom and my family have given me constant help and support during the last three years – I cannot thank them enough.

Finally, I would like to acknowledge the financial support of the Science and Engineering Research Council (SERC) of the United Kingdom.

DECLARATION

This thesis was composed by myself. The work reported herein, unless otherwise stated, is my own.

Paddy Byers
CHAPTER ONE  INTRODUCTION

1.1 Motivation

In the thesis I will study means of comparing algebraic structures by their behaviour in computations. The purpose of this section is to motivate this work by looking at abstract data types and program modules in a broader context.

1.1.1. Axiomatic specifications

Axiomatic methods for module specification are based on the view that a program module M is an entity with which a program interacts using a fixed set L of operations or procedures. An axiomatic specification for M is a pair (L, E), where E is a set of formal requirements which all implementations of M must satisfy.

Whilst axiomatic specifications of this sort were first proposed in a study of computer arithmetic in [Wijngaarden, 1966], it was only later that their more general purpose for user-defined types was appreciated; this principally came about from work on program specification and correctness [Floyd, 1967], [Hoare, 1969 and 72], [Hoare and Wirth, 1973] and that on modularisation [Parnas, 1972a and 72b]. Parnas' papers proposed an axiomatic language for specifying program modules, and generated research into specification methods best suited to this purpose.

In a subsequent discussion, it is suggested in [Liskov and Zilles, 1975] that any axiomatic specification method should satisfy the following criteria:

(1) Formality: A specification method should be formal, that is, specifications should be written in a notation which is formally sound. This criterion is mandatory if the specifications are to be used in conjunction with proofs of program correctness.
(2) **Constructibility**: It must be possible to construct specifications without undue difficulty. We assume that the writer of the specification understands both the specification technique and the concept to be specified.

(3) **Comprehensibility**: A person trained in the notation should be able to read a specification and then, with a minimum of difficulty, reconstruct the concept which the specification is intended to describe.

(4) **Minimality**: It should be possible using the specification method to construct specifications which describe the interesting properties of the concept and nothing more.

(5) **Wide range of applicability**: Associated with each specification technique there is a class of concepts which the technique can express in a natural and straightforward fashion, leading to specifications satisfying criteria (2) and (3).

(6) **Extensibility**: It is desirable that a minimal change in a concept results in a similar small change in its specification. This criterion especially impacts the constructibility of specifications.

Algebraic specifications were seen as satisfying these criteria for a wide class of problems, and arose in [Zilles, 1975], [Guttag, 1975], [Guttag and Horning, 1978] and [Goguen et al., 1975]; they have subsequently been studied extensively, and are believed to be a versatile tool for abstractly describing data and other program structures. Many different techniques, all differing in the specific kinds of axioms and models they allow have been proposed and studied, and their relative (theoretical) strengths and weaknesses have been studied. Particularly important examples of these relationships are found in [Bergstra and Tucker, 1983a and 83b], [Makowsky, 1985], [Mahr and Makowsky, 1983] and [Bergstra et al., 1981]. In particular, one theoretical issue is settled with the following theorem from [Bergstra and Tucker, 1983a] (Theorem 5.1):

**Proposition 1.1.1.** Every computable and every semicomputable structure is the initial model of a finite equational theory.
So as far as specifying individual computable algebras are concerned, algebraic methods with initial semantics are adequate. However, equational theories are not able to specify all of the classes of structures that users are interested in. The most striking example of this is the case of disjunction.

**Proposition 1.1.2.** Let $L$ be a language with three constant symbols, $a$, $b$ and $c$, say. Let $\phi$ be the sentence $\phi \equiv (a = b \lor a = c)$.

Then there is no equational theory over $L$ (or any extension of $L$) whose models are the same as those of $\phi$.

In view of these deficiencies, there are many theoreticians and practitioners who argue that algebraic methods do not suit their needs. Some reasons are the following.

(1) Some properties, whilst they might be expressible in algebraic terms, are *more naturally* expressed in some other logical framework, such as first order or modal logic. This relates to the requirement that specifications should be easily constructible. Specifications in first order logic are probably most familiar in VDM or Z [Spivey, 1988a and 88b], [Jones, 1980 and 86], whereas those in temporal logic are commonly found in work on reactive systems, e.g. [Pnueli, 1986], [Barringer et al., 1986]. The structures which are implemented are computable, and so are specifiable using equational logic, but it is not always clear how this should be done.

(2) It is unfeasible to determine every property desired of the implementation in question, such as in [Turski and Maibaum, 1987]. The example they give is that of natural number arithmetic – it is clearly not necessary to know whether or not Fermat’s last theorem is a theorem in order to be able to construct a useful theory of numbers. It is, however, true that it is not necessary to know this in order to be able to write an algebraic theory for arithmetic.

(3) Other properties, such as non-determinism, are properties of a physical system. Although, once again, any implementations can be expressed as algebraic structures, in order to reason about them in their environment it is necessary to express their specifications in a logical...
language that is able to handle non-determinism (such as that in the form of disjunction). Applications of this kind include reasoning about processes in non-deterministic environments, such as protocols transmitting through unreliable media. See, for example, [Hennessy and Milner, 1985].

(4) Non-determinism or permissiveness, whilst not being present in an eventual implementation, is an essential tool in the development of specifications and implementations – for example, by delaying design decisions, overspecification can be avoided and specifications and proofs can be reused. Examples of these circumstances are given in the context of process algebras in [Hoare, 1985], but the same can be said of other areas of specification. After all, abstraction is a tool that has been used by mathematicians for simplifying problems; why should we deny a computer scientist the same facility? This relates to the requirement that a specification method should allow specifications to be constructed minimally.

We will not dwell on the theoretical and philosophical issues raised by these points. We will however argue that as practitioners are searching for logical systems in which they can express their specifications naturally and minimally, it is necessary to consider the theoretical questions raised by the use of these systems.

1.1.2 Examples

As an example, suppose one wished to specify and implement a random number generator. An implementation will, we will suppose, be completely deterministic in the sense that if you knew enough about the state and the nature of the implementation, you would be able to predict the next number the generator would produce. However, in the absence of such knowledge, to sequence of numbers might appear to be random.

What specification might one write for such a system? You might say nothing more than that each number is drawn from some set S of interest, such as the rationals or some finite set.
More sophisticated specifications might impose some constraint on the sequence of numbers produced so that it appears to be random in some way. One might choose to insist that the sequence satisfies one or more of the (pseudo-random analogues of the) randomness tests used by statisticians; the most frequently used tests are:

1. the reachability test: every finite sequence of numbers from S eventually occurs in the infinite sequence of random numbers;
2. the frequency test: the expected number of occurrences of each number occurs eventually;
3. the flip test: the number of consecutive occurrences of each number is the expected number eventually;
4. the median test: (assuming there is an ordering on the set S), that the median value is eventually what would be expected.

The combination of these or of other constraints which might be imposed will depend upon the properties required of the implementation. In cases where a random number generator is used as an arbiter, in communications protocols for example, the reachability test (1) above will guarantee the "eventually ..." types of properties one relies on a random number generator to provide. Whichever way the specification is precisely formulated ought not, however, alter the fact that the specification says nothing about the internal state of the implementation, and is purely a statement about the sequence of numbers produced. That is, instead of characterising the state first and then reasoning about the programmed behaviour, we are starting with a particular requirement on the nature of the behaviour, and working towards the state and algorithms which might give us that behaviour. An ability to reason at the level of behaviour, independent of the representation, is essential to there being a suitable modularisation for any system using a random number generator.

Another example is rather more practical. In a recent paper, [Wichman, 1989], presents a formal specification (in VDM) intended as a basis for an ISO standard on floating point
operations. The specification does not state exactly what the outcome of every calculation should be, simply that calculated numbers should fall within a certain range. There will therefore be many different implementations of floating point satisfying this specification. This lack of completeness is quite deliberate – necessary, in fact, if one is going to specify exactly the required properties and nothing more.

1.1.3 Termination equivalence

In the thesis, we will not be concerned with how a module is specified, but associate to each specification a class $\mathcal{K}$ of structures of a language $L$, say. For an algebraic specification with initial model semantics, $\mathcal{K}$ will the the isomorphism class of the initial model, but for other specifications, $\mathcal{K}$ might be any class of $L$-structures closed under isomorphism.

There does not appear to have been any systematic study concerned with understanding the impact differences between module implementations can have on the system as a whole. The purpose of the work in this thesis then is to initiate such a study by suggesting approaches that might be taken; this leads us directly to look at the notion of observable or behavioural equivalence of many-sorted structures, particularly in relation to the outcome of computations which can arise from using such a structure.

So, the questions we address are:

1. How can we compare different implementations?
2. How might differences between structures be reflected in the behaviour of programs?
3. How might differences between structures change the behaviour of composite structures (or systems) built from them?
4. To what extent does the choice of programming formalism affect how we view the equivalence of structures?

Central to the study is the notion of termination equivalence of two structures – we will say that two structures are termination equivalent when precisely the same programs terminate.
everywhere (i.e. are total) on them. We see how several other aspects of the behaviour of a system are related to the termination of programs in different programming formalisms.

To see how this might work, return to the random number generator example; we will look at how behavioural properties of an implementation can be reflected in the properties of programs.

We will concentrate to start with on the reachability property (1) above.

Suppose our specification S consisted of the following:

1. a theory $T_D$ specifying the domain D from which our numbers are drawn;
2. the requirement that there exists some initial state $P_0$, say;
3. the requirement that for every finite sequence $\Delta$ of elements drawn from D, by repeatedly drawing numbers we will eventually come across the sequence $\Delta$ of numbers appearing consecutively.

An implementation of this specification will be a two-sorted structure $\mathcal{M}$, with

1. one number sort domain $D^{\mathcal{M}}$ from which the numbers are drawn;
2. one state sort domain $P^{\mathcal{M}}$ of possible program states for the system, about which we know nothing;
3. an element $P_0^{\mathcal{M}} \in P^{\mathcal{M}}$ which is the initial state;
4. a 'next number' function $\text{Next}: P^{\mathcal{M}} \rightarrow P^{\mathcal{M}} \times D^{\mathcal{M}}$ which, from a particular state, gives us another number and a new state;
5. $D^{\mathcal{M}}$ satisfying the theory given for the set D;
6. by some canny implementation, the $\text{Next}$ function giving sequences satisfying the pseudo statistical property stated.

Now consider the following program $Q_n$ with n input variables $d$ of sort D:

Starting at $P_0$, repeated apply $\text{Next}$ until the sequence $d$ of input variables is output consecutively; when it is, terminate.
Then the behavioural specification is that $Q_n$ is total for each $n$. This tells us that, for example, if $\mathcal{M}$ satisfies the behavioural specification and $\mathcal{N}$ is a termination equivalent model of $S$, then $\mathcal{N}$ is also an allowable random number generator.

For the other tests, we can write a suite of test programs whose termination properties characterise whether or not the generator passes that test. So, termination equivalence of implementations will ensure that passes of those tests will be carried from one implementation to another.

What else would termination equivalence tell us?

Suppose that $D^\mathcal{M}$ is finite of size $k$. For $n \in \mathbb{N}$, let $P_n$ be the program which repeated applies Next until it encounters a sequence of $n$ different domain elements, and terminates when it finds them. Now, $P_n$ will terminate on a random number generator precisely when the domain $D$ has no more than $n$ elements. So if $D^\mathcal{M}$ has size $k$, we will have $P_n$ terminating for all $n \leq k$, and diverging for all others; therefore the same will be true on any termination equivalent structure, and hence any equivalent structure will have its $D$ domain size $k$. Similarly, we can see that if $D$ is infinite on one structure, it will be infinite on any termination equivalent structure.

The other example we gave was that of the floating point operations; we can also look at how termination properties are reflected in that specification.

The following program appears in numerical analysis texts and is used to determine the 'machine accuracy' of a floating point implementation:

```plaintext
x := 1; count := 0;
do x := x/2; count := count + 1;
until x = 0;
output count
```
That is, repeatedly divide a non-zero number by 2 until it vanishes, and count the number of steps taken. The termination of this program is not decided by Wichmann's specification; that is, there exist valid implementations of floating point for which this does not terminate, and some for which it does.

1.1.4 Observable equivalence

The idea of comparing systems by their responses to tests or observations made of them is probably most familiar to the computer scientist in the realm of process algebras, such as observable and testing equivalence in CCS ([Milner, 1989] and [de Nicola and Hennessy, 1984] resp.) and bisimulation [Park, 1981]. The differences between these equivalences revolve around the extent to which internal events can be observed. In many ways, our equivalences are similar in that they do not make any reference to the internal state during a computation, only to the outcome of that computation in terms of its termination or otherwise.

The concept of termination equivalence of algebraic structures first appeared in [Kfoury, 1973] which was a conference paper, but this does not appear to have been followed up; the concluding result of that paper, Thm. 2.7, is false, but other elementary results are used in the thesis.

Other authors have looked at notions of observable equivalence of structures, usually in the context of purely algebraic specifications. One of these is [Sanella and Tarlecki, 1987], in which equivalence of structures is defined in terms of the satisfiability of types in a manner similar to that used in §4, but no specific instances of equivalences of interest are given, nor are they related to the computational properties of structures. For example, they assume that any equivalences of interest will have the horizontal compositionality property (§5.2), whilst termination equivalence does not have that property. In both [Reichel, 1981] and [Goguen, 1989], the sorts in a language are partitioned into data sorts and program sorts. The equivalences they propose are remarkably similar, and insist that the data sort domains are isomorphic, whilst the program sorts are related in a less trivial way. Once again, the
equivalences seem to be motivated in an ad hoc manner from examples, and no attempt is made to analyse how equivalence relates to other aspects of structure equivalence. In [Hayes, 1989], a notion of observable equivalence is proposed in which there are two type of sorts — *observable* sorts and *hidden* sorts. Observations relate to observing the equality of ground terms in observable sorts.
1.2 Computability over abstract structures

The plan to study termination equivalence means that the body of literature on computability over abstract structures is of particular interest to us. The study of programming features such as iterations, recursions, gotos, arrays, stacks and queues and their relationships seems to have been started independently and at about the same time by Paterson in [Paterson, 1968], [Luckham et al., 1970] and [Paterson and Hewitt, 1970] and by Friedman in [Friedman, 1971]. (Kfoury notes in [Kfoury, 1985b] that whilst Friedman's report was given at the logic colloquium in August 1969, Paterson and Hewitt's original report was dated November 1970.)

The work centres around the idea of a program scheme and its interpretation on a structure, and that of the equivalence of program schemes on all structures or uniformly on a class of structures. The importance of a general syntactic notion of a program scheme that can be applied to abstract structures was first discussed in [Luckham and Park, 1964] and [Engeler, 1968]; the latter paper is the origin of algorithmic and dynamic logic.

Many different programming formalisms were proposed, and their relative power studied. The most significant result arising from this study is that, in contrast to the case of computing over the natural numbers, different formalisms such as iterative programs and recursive programs actually have different programming powers. The research went on to study the circumstances under which formalisms that are different when all structures are considered have the same power if attention is restricted to a strict subclass of structures. The relationships between all of the different formalisms and the circumstances under which they can be regarded as the same are now largely understood; the results in this area we will need are stated in §2.

This study of programming formalisms became known as schematology. Important work in schematology is discussed in [Shepherdson, 1985], [Constable and Gries, 1972], [Kfoury, 1974], [Garland and Luckham, 1973], [Chandra, 1973], and [Chandra and Manna, 1972 and 73].
In parallel to the work on definability over the natural numbers, there has been a good deal of research into generalising these notions to abstract structures. Methods such as inductive definability, equational definability and fixed-point methods have all been generalised; the relationship between these methods and the machine-theoretic methods are also understood. These results and many others of an extensive classification programme are discussed in [Moldestad and Tucker, 1979] and [Tucker and Zucker, 1989].

Work on dynamic logic has succeeded that work on schematology. This work follows an important observation is made in [Kfoury and Park, 1975], which proves that dynamic logics have expressive power distinct to first order logic, and a great deal of work has been done in comparing the dynamic logics arising from different programming formalisms. Many of these results are based around a study of what has become known as the *unwind property*; structures with this property can be used to compare dynamic logics, and the relevance of this work to our study is explained more fully in §4.3, once the necessary definitions have been made. Work by Tiuryn [Tiuryn, 1981 and 85], [Tiuryn and Urzyczyn, 1988] and Engeler [Engeler 1971 and 75] and others has settled many of the questions of equivalence of dynamic logics. Of particular importance to our work is the study of the unwind property, notably in [Kfoury, 1983, 85a and 85c], [Urzyczyn, 1979, 81, 83 and 87] and [Kfoury and Urzyczyn, 1985 and 87].

Another formulation of abstract computability appears as 'term-enrichment' in [Bauer and Wirsing, 1988], [Orejas, 1985]. The exact power of computability by this means is not clear, and no attempt is made to discover how it relates to other older notions of computability or definability. It is clear that its power lies (not strictly) between that of fap and fapS.
1.3 Thesis outline

The material is divided into chapters, with sections and subsections, as it is in this introduction; we will refer to chapter 2, for example, as §2; to chapter 1, section 2, subsection 3 as §1.2.3.

In the next chapter, §2, I present definitions and results which are needed but are not specific to the thesis; this means that I will introduce all of the notations and conventions, and a more formal review of existing work in areas of interest, such as computability. I have included proofs of some necessary but uninteresting results for which I have not found detailed proofs elsewhere. There are sections containing results we will need in each different area, such as computability, the unwind property, computable structures, and so on. I also introduce some definitions of generalised definability which are distinct from all of the formalisms for computability looked at so far; these are needed later when we get on to look at congruences.

In §3, definitions and basic results which are specific to the thesis are presented. These are chiefly those of the algorithmic equivalences. Different equivalences are motivated by different concerns; we discuss how they might be relevant to other aspects of a structure's behaviour and look at some of their basic properties. Some work in relating different equivalences is also presented.

The following chapter, §4, is an assortment of interesting results all motivated by trying to understand the nature of termination equivalence. Another view of equivalences is given – namely equivalences based on what I have called types by analogy with types in model theory. I show how all of the equivalences of interest are expressible as type-based equivalences, and look at how the unwind property relates to these types. In §4.4 we make the analogy with model-theoretic types clearer, and show how several of the known results in model theory carry across to the generalised case. The model-theoretic concept of saturation is generalised to this case, and a few techniques for exploring termination equivalences are developed. In §4.5 we look at how computable structures can be characterised as those with certain termination properties.
The final chapter, §5 is intended to be a fairly complete analysis of the relationships between termination equivalences for the most common programming formalisms, and the congruences generated by them for the usual structure-building operation join. In §5.1 all but a few of the issues of relative strengths of equivalences are settled. The techniques used here and in the rest of the chapter are drawn largely from §4. In §5.2 and §5.3 the congruences generated by each of the equivalences are characterised. The final section deals with the differences between the equivalences and their congruences in the case where only computable structures are considered.

There is a section giving conclusions and a discussion of some open problems.
In this chapter we formally review known results we will be using in the thesis.

2.1 Many sorted languages and structures

We adopt the notations of [Tucker & Zucker, 1989] throughout.

2.1.1 Languages

A many sorted language (or signature) \( L \) may be defined as a pair \( L = (\text{Sort}, \text{Func}) \), where

1. \( \text{Sort} = \text{Sort}(L) \) is the finite set of sorts of \( L \): the \textit{algebraic sorts} labelled \( s_1, ..., s_r \) (for some \( r \geq 0 \)) say, and the \textit{boolean} sort \( B \). There may be an additional sort \( N \), the \textit{natural number} sort; \( N \) and \( B \) are the \textit{non-algebraic} sorts.

2. \( \text{Func} = \text{Func}(L) \) is a (usually finite) set of pairs \((f, \tau)\), where \( f \) is a \textit{function symbol} and \( \tau \) is the \textit{sort type} of \( f \), i.e. a tuple of the form \((m; i_1, ..., i_m, i)\) with \( m \geq 0 \), \( i_j \in \text{Sort} \) for \( j = 1, ..., m \) and \( i \in \text{Sort} \). The symbol \( f \) is called an \textit{m-ary function symbol}, with \textit{argument sorts} \((i_1, ..., i_m)\) and \textit{value sort} \( i \).

In particular:

(a) if \( i = B \) then \( f \) is a boolean-valued function symbol or \textit{relation symbol} of type \((m; i_1, ..., i_m, B)\), and we might emphasize this by writing \( \phi \) instead of \( f \);

(b) if \( m = 0 \), so that \( \tau = (0; i) \), then \( f \) is a nullary function symbol or \textit{individual constant symbol} of sort \( i \).
(3) We assume that \( \text{Func} \) includes symbols for certain standard functions associated with sorts \( N \) (where present) and \( B \):

(a) *arithmetical* function symbols for \( N \), representing the operations zero and successor and the equality relation on the natural numbers;

(b) *logical* function or relation symbols for \( B \): the constant symbols true and false, and symbols for a complete set of propositional connectives, say not and and;

(c) *equality* symbols \( =_i \) for equality on each sort \( i \in \text{Sort} \);

we will write \( = \) for the boolean equality relation \( =_B \).

All function and relation symbols other than those in (a)-(c) above will be called *algebraic*.

As a loose terminology, we will write "f in \( \text{Func} \)" to mean \( (f, \tau) \in \text{Func} \) for some \( \tau \). We will often say f is a \( \tau \)-ary function symbol or that f has *argument type* \( (m; i_1, \ldots, i_m) \).

2.1.2 Structures \( \mathcal{M} \) of language L

Given a language \( L = (\text{Sort}, \text{Func}) \) where \( \text{Sort} \) has algebraic sorts \( s_1, \ldots, s_r \) and \( \text{Func} = \{(f_1, \tau_1), \ldots, (f_s, \tau_s)\} \), a *structure \( \mathcal{M} \) of language \( L \), or \( L \)-structure \( \mathcal{M} \) has the form

\[
\mathcal{M} = ((M_i)_{i \in \text{Sort}}, (f_{ij})_{1 \leq j \leq s})
\]

where \( M_1, \ldots, M_r \) are non-empty sets called the *algebraic domains* (or *carriers*) and \( M_B = B = \{\text{tt, ff}\} \) is the domain of *truth values*; and for \( j = 1, \ldots, s \), if \( \tau_j = (m; i_1, \ldots, i_m, i) \) then

\[
f_{ij}^\mathcal{M} : M_{i_1} \times \ldots \times M_{i_m} \to M_i.
\]

If \( f_j \) is a relation symbol, we can alternatively regard \( f_{ij}^\mathcal{M} \) as a subset of \( M_{i_1} \times \ldots \times M_{i_m} \), viz.

\[
\{(x_1, \ldots, x_m) \in M_{i_1} \times \ldots \times M_{i_m} \mid f_{ij}^\mathcal{M}(x_1, \ldots, x_m) = \text{tt} \}.
\]

We insist that every equality relation symbol \( =_i \) is interpreted as identity; i.e.

for every \( x_1, x_2 \in M_i \), \( (x_1, x_2) \in =_i^\mathcal{M} \iff x_1 = x_2 \).
Finally, the standard function symbols have their standard interpretations on the domain \( \mathbb{B} \). If \( N \) is present as a sort in \( L \), then the domain \( M_N = \mathbb{N} = \{0, 1, \ldots\} \) is the domain of natural numbers. The function symbols zero and successor and the equality relation have their standard interpretations on this domain.

A structure \( M = ((N_i)_{i \in \text{Sort}}, (f^M_j)_{1 \leq j \leq s}) \) of \( L \) is a substructure of a structure \( \mathfrak{M} = ((M_i)_{i \in \text{Sort}}, (f^\mathfrak{M}_j)_{1 \leq j \leq s}) \) if:

1. for each \( i \in \text{Sort}, N_i \subseteq M_i; \)
2. for each \( j \in \{1, \ldots, s\}, f^\mathfrak{M}_j = f^M_j \restriction_{N_{i_1} \times \ldots \times N_{i_m}} \) if \( \tau_j = (m; i_1, \ldots, i_m, i) \).

i.e. the interpretation of \( f_j \) on \( M \) restricted to the appropriate domain on \( N \).

We will write \( \mathcal{N} < \mathfrak{M} \) to mean \( \mathcal{N} \) is a substructure of \( \mathfrak{M} \).

### 2.1.3 Extensions

Let \( L \) and \( L' \) be languages. We say that \( L' \) is an extension of \( L \) if

1. \( \text{Sort}(L) \subseteq \text{Sort}(L') \)
2. \( \text{Func}(L) \subseteq \text{Func}(L') \)

Now suppose that \( L' \) is an extension of \( L \) and let \( \mathfrak{M} \) be a structure of \( L' \). We define the reduct of \( \mathfrak{M} \) to \( L \) to be the structure

\[
\mathfrak{M}|_L = ((M_i)_{i \in \text{Sort}(L)}, (f^\mathfrak{M}_j)_{(f_j, \tau_j) \in \text{Func}(L)}).
\]

Given a structure \( \mathcal{N} \) of \( L \) and a structure \( \mathfrak{M} \) of \( L' \), we say \( \mathfrak{M} \) is an expansion of \( \mathcal{N} \) if \( \mathcal{N} \) is the reduct of \( \mathfrak{M} \) to \( L \). Of course, there can in general be many different expansions of any given structure to a larger language.

A particular extension of languages we will use is the following. Given a structure \( \mathfrak{M} \) of a language \( L \), and a set \( m = \{m_1, \ldots, m_k, \ldots\} \) in \( \mathfrak{M} \), the language \( L_m \) is \( L \) extended with new constant symbols \( c_1, \ldots, c_k, \ldots \). The structure \( \mathfrak{M}_m \) of \( L_m \) is the expansion of \( \mathfrak{M} \), with for each \( i \in \{1, \ldots, k, \ldots\} \), \( c^\mathfrak{M}_i = m_i. \)
2.1.4 Generated substructures

A many-sorted language $L = (\text{Sort}, \text{Func})$ is said to be void in algebraic sort $i$ if there is no ground term of $L$ with sort $i$; $L$ is said to be void if it is void in one of its algebraic sorts. Otherwise we say $L$ is non-void. There will be occasions when we have to consider whether or not a language is void – this is one of them.

Suppose that $X = (X_i)_{i \in \text{Sort}}$ is a family of subsets of the domains $(M_i)_{i \in \text{Sort}}$ of a structure $\mathcal{M}$; we will define a structure $\langle X \rangle$ (the structure generated by $X$). We first define the family $\langle (X_i)_{i \in \text{Sort}} \rangle$ of subsets of the domains $(M_i)_{i \in \text{Sort}}$ inductively by

1. For each $i \in \text{Sort}$, $X_i \subseteq \langle X_i \rangle$;
2. If $(f, \tau) \in \text{Func}$ with $\tau = (m; i_1, \ldots, i_m, i)$ and $x_1, \ldots, x_m$ are elements of $\langle X_i \rangle_{i_1}, \ldots, \langle X_i \rangle_{i_m}$ respectively then $f_M(x_1, \ldots, x_m) \in \langle X_i \rangle$;
3. That's all.

If each $\langle X_i \rangle$ is non-empty (as will be the case if $L$ is non-void, for example), we define $\langle X \rangle$ to be the substructure of $\mathcal{M}$ whose sort domains are the subsets $\langle (X_i)_{i \in \text{Sort}} \rangle$.

In the case where some of the $\langle X_i \rangle$ are empty we cannot form an $L$-structure with those domains because we insist on each sort domain being non-empty. In this case we say $\langle X \rangle$ is the $L'$-substructure $\langle X \rangle$ of the reduct $\mathcal{M}_{L'}$ where $L'$ is the language obtained from $L$ by removing every sort $i$ for which $\langle X_i \rangle$ is empty and every algebraic function symbol for which $i$ is an argument sort or the value sort.

Given an argument type $\tau = (m; i_1, \ldots, i_m)$, a $\tau$-ary tuple $x \in \mathcal{M}$ is a tuple $x = (x_1, \ldots, x_m)$ of elements of sort domains $M_{i_1}, \ldots, M_{i_m}$ respectively. We will write $\langle x \rangle$ to mean the structure $\langle (X_i)_{i \in \text{Sort}} \rangle$ where $X_i = \{ x_i \mid i = i \}$.

2.1.5 Classes of structures

We will often consider operations such as computation over a class $\mathcal{K}$ of $L$-structures. In these cases, $\mathcal{K}$ will be a class of structures sharing some property, such as all being equivalent in
some way, or all being models of some logical theory over \( L \). Of particular interest will be operations or computations which are uniform over \( K \) (such as translation between different classes of programs, as in §2.3.5). We will assume any class \( K \) is closed under isomorphism.

### 2.1.6 Homomorphism and isomorphism

Given two \( L \)-structures \( \mathcal{M} \) and \( \mathcal{N} \), we define an \( L \)-homomorphism \( \theta: \mathcal{M} \to \mathcal{N} \) between them to be a family \( \{ \theta_i \mid i \in \text{Sort}(L) \} \) of maps \( \theta_i: M_i \to N_i \) between domains of \( \mathcal{M} \) and \( \mathcal{N} \) such that for each function symbol \( f \) of \( L \) with type \( (m; i_1, \ldots, i_m, i) \), and for all \( x_1 \in M_{i_1}, \ldots, x_m \in M_{i_m} \),

\[
\theta_i(f^{\mathcal{M}}(x_1, \ldots, x_m)) = f^{\mathcal{N}}(\theta_{i_1}(x_1), \ldots, \theta_{i_m}(x_m)).
\]

In particular, for any constant \( c \) of sort \( i \), \( \theta_i(c^{\mathcal{M}}) = c^{\mathcal{N}} \). The functions \( \theta_N \) (where present) and \( \theta_B \) are thus required to be the identity functions and hence bijections on the domains \( \mathbb{N} \) and \( \mathbb{B} \) respectively.

We denote by \( \text{im}\theta \) or by \( \theta(\mathcal{M}) \) the structure of \( L \) that is the image of \( \mathcal{M} \) under \( \theta \), that is, the substructure of \( \mathcal{N} \) whose algebraic domains are \( \{ \theta_i(M_i) \mid i \in \text{Sort} \} \).

An \( L \)-homomorphism \( \theta \) is an isomorphic embedding if each of the \( \theta_i \) is injective; if, in addition, the \( \theta_i \) are surjective, we say that \( \theta \) is an isomorphism. If, for two \( L \)-structures \( \mathcal{M} \) and \( \mathcal{N} \), there exists an \( L \)-isomorphism \( \theta: \mathcal{M} \to \mathcal{N} \), we say that \( \mathcal{M} \) and \( \mathcal{N} \) are isomorphic and write \( \mathcal{M} \cong \mathcal{N} \).

Now suppose \( L' \) is an extension of \( L \), \( \mathcal{M} \) is a structure of \( L \) and \( \mathcal{N} \) is a structure of \( L' \). If \( \theta: \mathcal{M} \to \mathcal{N}_{|L} \) is a homomorphism (resp. isomorphic embedding) then we will (by abuse of notation) write that \( \theta: \mathcal{M} \to \mathcal{N} \) is a homomorphism (resp. isomorphic embedding).

The class of all \( L \)-structures isomorphic to an \( L \)-structure \( \mathcal{M} \) is written \( \text{ISO}(\mathcal{M}) \).
2.2 Operations on structures

There are several methods by which one can combine structures to form new structures. We will be particularly interested in two such methods, which we now define.

2.2.1 The join operation

This operation is intended to represent the way structures are put together to form composite systems and appears commonly as datatype addition or horizontal composition (as in e.g. [Sanella and Tarlecki, 1987]) and the combine operation (as in CLEAR, [Burstall and Goguen, 1977]). The idea behind the join operation is a very simple one: given two structures, we regard their algebraic sorts as being disjoint, and simply put the two structures together side by side to get a structure of a language whose sorts are those of one structure together with those of the other.

The join operation can be used to combine any two structures of any two many-sorted languages provided that the sets of algebraic sort names in the languages are disjoint. We define the join in two stages: we first have to describe the join of two languages, and then describe how the join of two structures is obtained.

Let \( L_1 = (\text{Sort}_1, \text{Func}_1) \) and \( L_2 = (\text{Sort}_2, \text{Func}_2) \) be two languages. The join of \( L_1 \) and \( L_2 \), written \( L_1 + L_2 \), has sort set \( \text{Sort}_{1+2} \) and function set \( \text{Func}_{1+2} \), where:

1. \( \text{Sort}_{1+2} = \text{Sort}_1 \cup \text{Sort}_2 \). Thus, if, for example, \( \text{Sort}_1 = \{r_1, \ldots, r_s, B\} \) and \( \text{Sort}_2 = \{s_1, \ldots, s_t, N, B\} \), then \( \text{Sort}_{1+2} = \{r_1, \ldots, r_s, s_1, \ldots, s_t, N, B\} \).

2. \( \text{Func}_{1+2} = \text{Func}_1 \cup \text{Func}_2 \). That is, the operation symbols in the new language are just those of the old languages operating on their original sorts.

Now let \( M_1 = ((M_1 \downarrow i)_{i \in \text{Sort}_1}, (f_{1_j}M_1^1)_{1 \leq j \leq s_1}) \) and \( M_2 = ((M_2 \downarrow i)_{i \in \text{Sort}_2}, (f_{2_j}M_2^2)_{1 \leq j \leq s_2}) \).
be structures of languages $L_1$ and $L_2$ respectively. We define the join of $\mathcal{M}_1$ and $\mathcal{M}_2$, written $\mathcal{M}_1 + \mathcal{M}_2$, to be a $L_1 + L_2$-structure

$$\mathcal{M}_1 + \mathcal{M}_2 = \langle (M_1)_i \in \text{Sort}_1, (M_2)_i \in \text{Sort}_2 \rangle, \langle (f_{ij})_{1 \leq i \leq s_1}, (f_{ij})_{1 \leq j \leq s_2} \rangle$$

We state the following facts without proof.

1. For any structures $\mathcal{M}_1$ and $\mathcal{M}_2$, the structures $\mathcal{M}_1 + \mathcal{M}_2$ and $\mathcal{M}_2 + \mathcal{M}_1$ are identical (we say join is commutative);
2. For any structures $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$, the structures $\mathcal{M}_1 + (\mathcal{M}_2 + \mathcal{M}_3)$ and $(\mathcal{M}_1 + \mathcal{M}_2) + \mathcal{M}_3$ are identical (join is associative).

Using these we can define the join of any finite set of structures.

The join operation is intended to be used where the algebraic sorts and operations on two structures are thought of as distinct. So, for example, given a gimp $(G, *)$ we might take the join of the two structures $((G, B), (e, *, \cdot, true, false, and, not))$ and $((N, B), (zero, succ, true, false, and, not))$ to get the structure $((G, N, B), (e, *, \cdot, true, false, and, not))$, a group with counting; the algebraic sort and the non-algebraic sort are kept apart. The other operation we introduce, disjoint union, does not keep the elements from constituent structures apart, however.

### 2.2.2 The disjoint union operation

The idea behind this operation is quite different to that of join; it is simple, but is motivated by different considerations. We will be using disjoint union to build structures for our examples, and it is not intended to have any meaning in the context of datatype-building. Given a set of structures of the same language, we form a new structure whose domain for a given sort is the (disjoint) union of the domains for that sort of each of the constituent structures. The operations are interpreted in a way on these larger domains in a way which makes sense.
In contrast to the join operation, the disjoint union operation can only be used to combine structures of the same language. Moreover, the language must not contain any function symbols with algebraic value sort but no algebraic argument sorts (such as constant symbols). However, we can disjoint union any set of structures subject to this constraint, so we are not limited to just finite sets of structures.

The only technical difficulty in defining the disjoint union is that arising when there are function symbols with arities greater than 1. We need to have a way of interpreting these symbols when their arguments are drawn from different structures involved in the disjoint union. We cannot just allow them to be interpreted arbitrarily as we will need to reason about the behaviour of programs on the composite structure. Instead, we introduce new elements into the sort domains onto which those instances of the functions are interpreted.

Let $L = (\text{Sort}, \text{Func})$ be a many sorted language with no operation symbols whose only algebraic sort is the value sort and let $\{M_k\}_{k \in K}$ be an indexed family of $L$-structures, where for each $k$,

$$M_k = ((M_{k,i})_{i \in \text{Sort}}, (f^M_{i,j})_{1 \leq j \leq s}).$$

For each $f$ in $\text{Func}$ with arity greater than 1 and algebraic value sort $i$, we introduce a new domain element $\perp_f$ of sort $i$. Then the disjoint union $\mathcal{N} = \bigcup_{k \in K} M_k$ is the structure

$$\mathcal{N} = ((\mathcal{N}_i)_{i \in \text{Sort}}, (f^\mathcal{N}_{i,j})_{1 \leq j \leq s}),$$

where

1. the domain $\mathcal{N}_i$ of algebraic sort $i$ is the disjoint union

$$\mathcal{N}_i = \bigcup_{k \in K} M_{k,i} \cup \{ \perp_f | f \text{ has arity } > 1 \text{ and value sort } i \}$$

of the domains for the sort $i$ in each of the structures $M_k$ together with the new domain elements;
(2) the function $f_j$ of sort type $\tau = (m; i_1, \ldots, i_m, i)$ for algebraic sorts $i$ is interpreted by

$$f_j^\mathcal{M}(x_1, \ldots, x_m) = \begin{cases} f_j^{M_k}(x_1, \ldots, x_m) & \text{if for some } k, (x_1, \ldots, x_m) \in M_{i_1} \times \cdots \times M_{i_m} \\ f & \text{otherwise.} \end{cases}$$

(3) the function $f_j$ of sort type $\tau = (m; i_1, \ldots, i_m, B)$ is interpreted by

$$f_j^\mathcal{M}(x_1, \ldots, x_m) = \begin{cases} f_j^{M_k}(x_1, \ldots, x_m) & \text{if for some } k, (x_1, \ldots, x_m) \in M_{i_1} \times \cdots \times M_{i_m} \\ f & \text{otherwise.} \end{cases}$$

(4) the function $f_j$ of sort type $\tau = (m; i_1, \ldots, i_m, N)$ is interpreted by

$$f_j^\mathcal{M}(x_1, \ldots, x_m) = \begin{cases} f_j^{M_k}(x_1, \ldots, x_m) & \text{if for some } k, (x_1, \ldots, x_m) \in M_{i_1} \times \cdots \times M_{i_m} \\ 0 & \text{otherwise.} \end{cases}$$

So the operations are extended to the new domain elements by 'strictness'. The reason for our insistence that functions with algebraic value sort must have at least one algebraic argument sort is that it would otherwise be more difficult to define the disjoint union in an intuitive way.

Notice that of a class $\mathcal{K}_1$ is a subclass of a class $\mathcal{K}_2$ of structures then $\bigcup_{k \in \mathcal{K}_1} \mathcal{M}_k$ is a substructure of the disjoint union $\bigcup_{k \in \mathcal{K}_2} \mathcal{M}_k$.

The disjoint union operation will nearly always be used in the case where there are only unary function symbols, but there are a few occasions where it is used in the more general case. Once again we state without proof the commutativity property, that up to isomorphism, the disjoint union of an indexed set of structures is independent of the indexing set, provided that they are bijective with one another. Because of the difficulty over the new domain elements, it is not associative up to isomorphism.
As an example of a disjoint union, consider the reduct $\mathfrak{N}$ of the standard structure $\mathbb{N}$ to the language containing just the successor function. We might be interested in the structure which is the disjoint union of countably many copies of $\mathfrak{N}$; this is a single sorted structure (not counting $B$) with just the successor function (Fig. 2.2.1).

![Figure 2.2.1. A disjoint union](image)

Figure 2.2.1. A disjoint union
2.3 Models of computation

The theory of recursive or computable functions has been generalised in many ways:

(1) to provide techniques for use in other parts of mathematics, particularly logic, set theory, and classical recursive function theory itself, such as in [Kleene, 1959] and [Moschavakis, 1969];
(2) to gain better understanding of advanced recursion theory, e.g. degree theory and hierarchy theory, such as in [Post, 1948] and [Kleene, 1943];
(3) to gain better understanding of the nature of computation.

The generalisations resulting from (1) and (2) have been very successful but have not helped with (3) very much because they usually involve generalising the notion of finite, and involve infinitely long computations; these limitations are discussed in, for example, [Grilliot, 1974] and [Moschavakis, 1969]. The models of computation we will consider, however, arose in studies aimed at (3) (particularly in [Friedman, 1971] and [Paterson and Hewitt, 1970]) and generalise computation in a way that retains the effectivity of procedures. In our work, it is convenient at times to consider different (though equivalent) models of computation over abstract structures, and so we present here the approaches we will need to use. The most important distinction between the models we consider is the way in which recursion (as opposed to iteration) is achieved. In the machine based models, the machine is equipped with a stack, and recursive procedures must be implemented; in the schema models there is no stack, but recursive procedures may be written directly.

2.3.1 Machine based models

The machine based formulation of computation over abstract structures was first considered in the paper [Friedman, 1971]. Since then, several extensions to the machines have been examined (notably the inclusion of a stack by various authors such as in [Moldestat et al., 1980]), and these give rise to different classes of computable functions. In its most basic form, a machine consists of a fixed finite set of registers, each of which at any time may hold a data element. A machine is programmed by a formalised algorithmic procedure (fap) which consists of a finite
set of labelled instructions, each having one of four forms. More powerful machines are created by adding either counting registers, a stack, or both. The original definitions were made in the context of single sorted structures, so the formal definitions which follow are different in that they apply in the many sorted case.

Let \( L = (\text{Sort}, \text{Func}) \) be a many sorted language. We will first describe the class \( \text{fap}_L \) of programs over \( L \), and then describe how a given \( P \in \text{fap}_L \) is to be interpreted on a structure \( \mathcal{M} \) of \( L \).

A fap over \( L \) is a quintuple \( (V, \sigma, I, O, P) \), where

1. \( V \) is a finite list of program variables \( V = \{v_1, ..., v_k\} \); 
2. \( \sigma \) is a sort function \( \sigma : V \rightarrow \text{Sort} \); 
3. \( I = (I_1, ..., I_l) \) is a tuple of input variables drawn from \( V \); 
4. \( O \in V \) is the output variable; 
5. \( P = (P_1, ..., P_t) \) is a finite sequence of program instructions, each of which has one of the following forms:
   a. \( v_i = f(v_{i_1}, ..., v_{i_m}) \), where \( f \) is a function symbol of \( L \); 
   b. if \( v_i \) go to \( P_a \) else go to \( P_b \); 
   c. goto \( P_a \) 
   d. stop.

A fap is type correct if

1. for each instruction of type (a), if the sort type of \( f \) is \( \tau = (m; i_1, ..., i_m, i) \), then for each \( t \in \{1, ..., m\} \), \( \sigma(v_{i_t}) = i_t \) and \( \sigma(v_i) = i \); 
2. for each instruction of type (b), \( \sigma(v_i) = B \).

The set \( \text{fap}_L \) is the set of all type-correct faps over \( L \).

Now let \( \mathcal{M} \) be an \( L \)-structure, and let \( P \in \text{fap}_L \). The type of \( P \) is the tuple \( (I; \sigma(I_1), ..., \sigma(I_l), \sigma(O)) \). We define the partial function

\[ P^\mathcal{M} : M_{\sigma(I_1)} \times ... \times M_{\sigma(I_l)} \rightarrow M_{\sigma(O)} \]
by picking \((x_1, \ldots, x_j) \in M_{\sigma(I_1)} \times \cdots \times M_{\sigma(I_1)}\) and considering the sequence of instructions executed in the program if \(x_1, \ldots, x_j\) are assigned to the variables \(I_1, \ldots, I_j\) and execution started at the first instruction. The sequence of instructions followed is defined in the usual way, and \(P^m(x_1, \ldots, x_j)\) is defined iff the execution arrives at a stop instruction with an assignment having been made to the output variable \(O\); the value of \(P^m(x_1, \ldots, x_j)\) is taken to be the contents of that variable. We say that the sort type of \(P\) is \((I; \sigma(I_1), \ldots, \sigma(I_j), \sigma(O))\). In the usual way, we define \(\text{domain}(P^m)\) to be the set of all tuples \((x_1, \ldots, x_j)\) for which \(P^m(x_1, \ldots, x_j)\) is defined. If \(\sigma(O) = B\), we can also think of \(P^m\) as a subset of \(M_{\sigma(I_1)} \times \cdots \times M_{\sigma(I_j)}\) in the way already described.

We define the set \(\text{FAP}(M)\) to be the set of all partial functions \(\{P^m | P \in \text{fap}_L\}\).

Now we can extend the machine to include counting; formally, we define the set \(\text{fap}_C_L\) of algorithms with counting by allowing the fap to include a finite set of counter variables \(c_1, \ldots, c_j\); the additional instructions are

\[
\begin{align*}
\text{(e)} & \quad c_i := \text{zero}; \\
\text{(f)} & \quad c_{i_1} := \text{succ}(c_{i_2}); \\
\text{(g)} & \quad \text{if } c_{i_1} = c_{i_2} \text{ then go to } P_a \text{ else go to } P_b.
\end{align*}
\]

The interpretation \(P^m\) of a fapC on a structure \(M\) is once more defined in the usual way; accordingly, we define \(\text{FAPC}(M)\) to be the set of all partial functions \(\{P^m | P \in \text{fap}_C_L\}\).

A further mechanism is a stack. By introducing it we obtain \(\text{fap}_S\) and \(\text{fapCS}\). It was observed in [Shepherdson, 1985] that if at least two stacks are allowed then these can be used to mimic the effect of any number of counters. However, with only one stack, it is in general possible to compute more than without a stack, but less than is possible with a stack and counters. We therefore define \(\text{fap}_S\) to be the set of all programs with one stack; this makes our definition agree with [Moldestad et al., 1980] but disagree with [Shepherdson, 1985].
A formalised algorithmic procedure with a stack, or fapS, is a quintuple \((V, \sigma, I, O, P)\), where

1. \(V\) is a finite list of program variables \(V = \{v_1, \ldots, v_k\}\);
2. \(\sigma\) is a sort function \(\sigma: V \rightarrow \text{Sort}\);
3. \(I = (I_1, \ldots, I_l)\) is a tuple of input variables drawn from \(V\);
4. \(O \in V\) is the output variable;
5. \(P\) is a finite sequence \(P = (P_1, \ldots, P_t)\) of program instructions, each of which has one of the following forms:
   a. \(v_i := f(v_{i_1}, \ldots, v_{i_m})\), where \(f\) is an operation symbol of \(L\);
   b. if \(v_i\) go to \(P_a\) else go to \(P_b\);
   c. goto \(P_a\);
   d. push (a); (meaning push the contents of all registers on the stack, together with the marker a)
   e. if stack=\(\emptyset\) then go to \(P_b\) else go to marker; (meaning go to \(P_a\) where a is the marker placed with the register entries at the top of the stack)
   f. restore (j); (meaning replace the contents of all variables except \(v_j\) with those in the entry at the top of the stack)
   g. stop.

The type-correctness requirements are as before; the stacks are assumed to be empty at the start of a computation. Given a fapS \(P\) and an L-structure \(\mathcal{M}\), we extend the definitions already given to define \(P^{\mathcal{M}}\), the sort type of \(P\), \(\text{domain}(P^{\mathcal{M}})\) and FAPS(\(\mathcal{M}\)).

Similarly, we extend fapS with counter variables to obtain fapCS, FAPCS(\(\mathcal{M}\)).

### 2.3.2 Schema based models

The study of computability over abstract structures appears to have started in two independent strands of research; the machine based models already discussed were one of these strands and schema based models were the other. As we have already mentioned, the main difference is in the way recursion is approached; in recursive program schemes recursive calls can be written directly. The reason we consider this model is that it gives us insight into the computing power...
of a machine with (what might seem to be the arbitrary restriction of having only) one stack; it gives us precisely the same power as an ability to write recursive schemes (the relationships between different models are discussed more fully in §2.3.5).

Given a language, associated with some structure $M$, four classes of program schemes are commonly considered: **basic program schemes**, **loop free program schemes**, **iterative program schemes**, and **recursive program schemes**. An iterative program scheme can be drawn as a flow-chart which only mentions relation and function names from $L$; whereas a recursive program scheme can be drawn as a flow-chart which may also mention in its instructions names of program schemes (including itself). A loop-free program scheme is an iterative program scheme whose flow-chart contains no loops but which can be forced to diverge with a special instruction, whereas basic program schemes are not allowed this instruction are are therefore forced to define total functions on every structure. Iterative and recursive program schemes can be with or without counters; these correspond to adding counter variables to a fap or fapS respectively.

We will not give formal definitions of these models because we will never use them. We will, however, formalise how they relate to the models we will be discussing and explain how those relationships affect the work in the remainder of the thesis. We will formally define the notions of basic and loop free scheme when we discuss effective definitional schemes in the next subsection.

### 2.3.3 Effective definitional schemes

Effective definitional schemes (eds) were introduced in [Friedman, 1971]. They are less closely related to actual sequences of computations than the other models, hide all details such as whether counters are allowed and how recursion is actually implemented and are in many ways easier to work with. They are used in this work because they can easily be generalised to functions which are not effectively computable, which is not as easy with the machine based and schema based models.
The other aspect in which they differ from the other models is that it is possible to define the schemes over languages which are not finite, that is infinitely many operation symbols, but each with finite arity and only finitely many sorts. However, the opportunity to do this has not been taken in any of the definitions appearing in the literature; we will not do it either, as we will in the first instance be using eds as an equivalent representation of other models. Once again, we explain the relationships between these and other models in §2.3.5.

Let $L$ be a language, and $\tau = (m; i_1, \ldots, i_m, i)$ be a sort type of $L$. An effective definitional scheme over $L$ of type $\tau$ is a recursively enumerable set $S$ of clauses

$$\{ E_{1s}(x) \land E_{2s}(x) \land \cdots \land E_{ks}(x) \rightarrow t_s(x) \mid s \in S \}$$

where

1. $x = (x_1, \ldots, x_m)$ is a tuple of variables associated to sorts $i_1, \ldots, i_m$ respectively
2. each $E_j(x)$ is a term in $L$ over $x$ of sort $B$;
3. $t(x)$ is a term in $L$ over $x$ of sort $i$.

In the case where $L$ is infinite, we also demand that the set of all operation symbols appearing in any of the clauses is finite. We also demand that the antecedents of the clauses are pairwise exclusive; that is that there must be a boolean term appearing in one whose negation appears in the other. The set of all effective definitional schemes over a language $L$ is denoted $\text{eds}_L$.

Now let $\mathcal{M}$ be a structure of $L$ and let $(x_1, \ldots, x_m)$ be assigned to elements of the domains $M_j$ of the appropriate sorts $j$ in $\mathcal{M}$. Then $S^\mathcal{M}$ is defined at that particular interpretation if there exists a clause $E_{1s}(x) \land E_{2s}(x) \land \cdots \land E_{ks}(x) \rightarrow t(x)$ for which each $E_j^\mathcal{M}(x_1, \ldots, x_m) = t$; $S^\mathcal{M}$ is then defined to be $t^\mathcal{M}(x_1, \ldots, x_m)$.

The set of all partial functions eds-definable on a structure $\mathcal{M}$ is denoted $\text{EDS}(\mathcal{M})$. 
The class loop free\(_L\) of loop free schemes associated to a language \(L\) is the class of finite schemes
\[
\text{loop free}\_L = \{ \{ E_{1s}(x) \land E_{2s}(x) \land \ldots \land E_{ks}(x) \rightarrow t_s(x) \mid s \in S \} \in \text{eds}_L \mid S \text{ is finite} \}.
\]
Then for an \(L\)-structure \(\mathcal{M}\), we define \(\text{LOOP FREE}(\mathcal{M}) = \{ P^{\mathcal{M}} \mid P \in \text{loop free}_L \} \).

A loop free scheme \(\{ E_{1s}(x) \land E_{2s}(x) \land \ldots \land E_{ks}(x) \rightarrow t_s(x) \mid s \in S \}\) is termed basic if it is total; i.e. if
\[
\vdash \forall x. \bigvee_{s \in S} (E_{1s}(x) \land E_{2s}(x) \land \ldots \land E_{ks}(x)).
\]

We define basic\(_L\) and BASIC(\(\mathcal{M}\)) for languages \(L\) and structures \(\mathcal{M}\) in the natural way.

2.3.4 Remarks on terminology

Given a program \(P\) of type \(\tau = (m; i_1, \ldots, i_m, i)\) over a language \(L\) in some model of computation and an \(L\)-structure \(\mathcal{M}\), we say \(P\) is total on \(\mathcal{M}\) if \(P^{\mathcal{M}}\) is a total function on \(M_{i_1} \times \ldots \times M_{i_m}\). Otherwise we say \(P\) is partial (and so mean strictly partial).

If we say 'the (possibly partial) functions \(f\) and \(g\) are equal on \(\mathcal{M}\)' or write \(f^{\mathcal{M}} = g^{\mathcal{M}}\), we will mean that
\[
\begin{align*}
(1) \quad \text{domain}(f^{\mathcal{M}}) &= \text{domain}(g^{\mathcal{M}}) = D \text{ say}; \\
(2) \quad f\mid_D^{\mathcal{M}} &= g\mid_D^{\mathcal{M}}.
\end{align*}
\]
That is, their domains coincide and they are equal on that domain.

If we say 'the computable function \(f\)', we will mean the program \(f\) in some model of computation (where the model is not important or clear from the context).

If we say 'the computable function \(f\) on the structure \(\mathcal{M}\)', we will write \(f^{\mathcal{M}}\) where \(f\) is computable.

If we say 'the function \(f\) on \(\mathcal{M}\) is computable', we mean that there exists a program \(P\) for which \(P^{\mathcal{M}}\) is equal to \(f\). Where the choice of \(P\) is important, this will be made clear.

Let \(K\) be a class of \(L\)-structures for some language \(L\), and let \(P\) and \(S\) be computable over \(L\). Then we say \(P\) and \(S\) are \(K\)-equivalent if, for each \(\mathcal{M} \in K\), \(P^{\mathcal{M}}\) and \(S^{\mathcal{M}}\) are equal.
Some authors, notably Kfoury in [Kfoury, 1983], allow their schemes to involve parameters drawn from the sort domains of a structure, and consider computations performed uniformly over those parameters. There is only one section (§4.4) in which we are interested in parameterised computations, so the mechanisms are introduced there in order that this introduction is not weighed down unnecessarily.

2.3.5 Equivalence of models of computation

As already mentioned, a good deal of work has been done in unifying the various models of computation and assessing their relative computing powers. Here we summarise the results in this area which we will need.

Suppose that $P$ and $Q$ are programs in some models of computation of sort type $\tau$ over a language $L$; we say $P$ translates to $Q$ if for every $L$-structure $\mathcal{M}$, $P^\mathcal{M} = Q^\mathcal{M}$. That is, $Q$ acts as an equivalent representation for $P$ uniformly over all structures.

We say a class $C_1$ of programs translates to a class $C_2$ of programs if there is a map

$$\theta: C_1 \to C_2$$

such that for every $P \in C_1$, $P$ translates to $\theta(P)$.

The idea is that if we translate one program $P$ to another $Q$, the form of $Q$ should be obtained syntactically to that of $P$, but it is not always clear how a simple relationship would be guaranteed by the existence of a translation as defined above. This means that we will need to be careful to make sure that whenever we switch from one model of computation to another we pay attention to the way we rely upon the particular choice of translation.

We will use the following facts. There are translations uniformly over all structures in each of the directions:

1. basic $\rightarrow$ loopfree;
2. loopfree $\rightarrow$ iterative;
(3) fap → iterative;
(4) iterative → fap;
(5) fap → fapC;
(6) fap → fapS;
(7) fapC → fapCS;
(8) fapS → fapCS;
(9) fapS → recursive;
(10) recursive → fapS;
(11) fapCS → eds;
(12) eds → fapCS;

These facts are proved either trivially (1, 2, 3, 4, 5, 6, 7, 8), are in [Chandra, 1973] (9, 10), or in [Friedman, 1971], (11, 12).
2.4 Generalising notions of computability

In addition to using the schemes defined in the previous section, we might be interested in defining functions by schemes which are not in general recursive. Functions such as these will be of interest in §4 and §5. There are several different ways in which we might consider extending the classes of definable functions; we might be interested in first-order definable functions, for example. We will look at ways in which we could generalise effective schemes whilst retaining some of their useful properties.

2.4.1. Generalised schemes

One important property we will use, is that if a program terminates (or a scheme is defined) for a particular point in a structure, it is as a result of evaluating only finitely many atomic relations on its arguments. Furthermore, if another set of arguments satisfy the same finite set of atomic relations, then the outcome of the computation using those arguments (in terms of the sequence of actions, the terms over the arguments stored in each variable, etc.) is the same.

Another property is that computations are *local* and only involve elements in the substructure generated by the finite set of input variables. We are also interested in just those functions which have a finite set of arguments.

Properties which are independent of these, and which we might consider losing in a generalisation of the schemes are the following:

1. there are only countably many computable functions;
2. the sequence of operations undergone in a computation is r.e. in the given functions and relations of a structure;
3. a program scheme, being only finitely presented, can only involve a finite subset of the (possibly infinite) set of operations in the language.

In this section, we generalise the notion of effective definitional scheme to give the widest class of definable functions satisfying the requirements outlined, but without the properties (1) - (3).
Recall that in the definition of effective definitional scheme, a function was defined by a recursively enumerable set of clauses

$$E_1(x) \land E_2(x) \land ... \land E_k(x) \rightarrow t(x)$$

where

1. $x = (x_1, ..., x_m)$ is a tuple of variables associated to sorts $i_1, ..., i_m$ respectively.
2. Each $E_j(x)$ is a term in $L$ over $x$ of sort $B$.
3. $t(x)$ is a term in $L$ over $x$ of sort $i$.

We also demanded that the antecedents of the clauses are pairwise exclusive, that is that there must be a boolean term appearing in one whose negation appears in the other.

We can generalise the definition of scheme simply by dropping the requirement that the set of clauses is r.e., and dropping the requirement for the set of operation symbols to be finite.

Suppose then that $f$ is a definable function with the properties:

1. The result of applying the function if it terminates to arguments $x$ is in the substructure generated by $x$, and is generated symbolically from the variables $x$;
2. Is equal on tuples satisfying the same atomic relations;
3. Is determined in every terminating case by a finite set of atomic relations.

Then $f$ associates a term in $x$ to each of a finite set of atomic formulae or their negations over $x$; so $f$ is nothing more than a set of clauses of the form described.

**Definition 2.4.1.** We will denote this set of function definitions as $G_0(L)$, or simply as $G_0$ where no confusion arises.

The interpretation of such a function on a structure $\mathcal{M}$ is defined in exactly the same way as for eds.
More terminology: If we say 'the definable function $P$ over $L$', we will mean the function definition $P$ from $G_0(L)$.

If we say 'the definable function $P$ on the structure $\mathcal{M}$', we will $P^\mathcal{M}$ where $P$ is definable.

If we say 'the function $f$ on $\mathcal{M}$ is definable, we mean that there exists a $G_0$ scheme $P$ for which $P^\mathcal{M}$ is equal to $f$. Where the choice of $P$ is important, this will be made clear.

Let $K$ be a class of $L$-structures for some language $L$, and let $P$ and $S$ be definable over $L$. Then we say $P$ and $S$ are $K$-equivalent if, for each $\mathcal{M} \in K$, $P^\mathcal{M}$ and $S^\mathcal{M}$ are equal.

2.4.2. Operations on schemes

In the context of programming languages, we understand what is meant by constructs such as sequential composition and conditional expressions. In order to be able to manipulate scheme definitions, we allow ourselves analogous mechanisms to these constructs.

Definition 2.4.2. Let $L$ be a language and let $\tau = (m; i_1, \ldots, i_m, B)$ be the sort type of schemes $P$ and $S$ over $L$, where

$$P = \{ E_{1p}(x) \land \ldots \land E_{kp}(x) \to t_p(x) \mid p \in P \}$$

and

$$S = \{ E_{1s}(x) \land \ldots \land E_{ks}(x) \to t_s(x) \mid s \in S \}.$$

We define the conjunction of $P$ and $S$, $\wedge(P, S)$, to be the $\tau$-ary scheme

$$\wedge(P, S) = \{ (E_{1p}(x) \land \ldots \land E_{kp}(x)) \land (E_{1s}(x) \land \ldots \land E_{ks}(x)) \to t_p(x) \land t_s(x) \mid p \in P, s \in S \}.$$ 

Now let $\tau = (m; i_1, \ldots, i_m, i)$ and $\tau' = (m+1; i_1, \ldots, i_m, i, j)$ be sort types and let

$$P = \{ E_{1p}(x) \land \ldots \land E_{kp}(x) \to t_p(x) \mid p \in P \}$$

and

$$S = \{ E_{1s}(x, y) \land \ldots \land E_{ks}(x, y) \to t_s(x, y) \mid s \in S \}.$$
be schemes of type $\tau$ and $\tau'$ respectively. The we define the \textit{sequential composition} of $P$ and $S$, $P; S$, to be the scheme

$$P; S = \{ C_{ps} \mid p \in P, s \in S \},$$

where for each $p \in P$ and $s \in S$, the clause $C_{ps}$ is

$$C_{ps} = E_{1p}(x) \land \ldots \land E_{kp}(x) \land E_{1s}(x, t_p(x)) \land \ldots \land E_{ks}(x, t_p(x)) \rightarrow ts(x, t_p(x)).$$

Now let $\tau = (m; i_1, \ldots, i_m, i)$ be the type of schemes $P$ and $S$, where

$$P = \{ E_{1p}(x) \land \ldots \land E_{kp}(x) \rightarrow t_p(x) \mid p \in P \}$$

and

$$S = \{ E_{1s}(x) \land \ldots \land E_{ks}(x) \rightarrow t_s(x) \mid s \in S \};$$

Let $\psi(x)$ be an atomic relation from $L$ of sort type $(m; i_1, \ldots, i_m)$. The we defined the conditional scheme $\text{if}(\psi, P, S)$ to be

$$\text{if}(\psi, P, S) = \{ \psi(x) \land E_{1p}(x) \land \ldots \land E_{kp}(x) \rightarrow t_p(x) \mid p \in P \}$$

$$\cup \{ \lnot\psi(x) \land E_{1s}(x) \land \ldots \land E_{ks}(x) \rightarrow t_s(x) \mid s \in S \};$$

We will also be interested in classes of schemes that are closed under these basic constructions.

\textbf{Definition 2.4.3.} Let $L$ be a language and $C$ a class of definable functions over $L$. We say $C$ is \textit{closed under basic constructs} if

1. every loop free scheme over $L$ is in $C$;

2. whenever $P$ and $S$ are in $C$ and $\psi$ is an atomic relation over $L$, the schemes
   \begin{enumerate}
     \item $\land (P, S)$
     \item $P; S$
     \item $\text{if}(\psi, P, S)$
   \end{enumerate}
   are all in $C$ when they exist.
2.5 The unwind property

Whilst everything we have done so far has been in the general context of many sorted languages and structures, a good deal of our work will be in the single sorted case. The first reason for this is that for most of our examples we will only need single sorted examples as illustrations. The second, and more important, reason is that we are often looking at the interaction of logics (usually first order logic) and programs. In this case, exactly how first order logic works in the many sorted case is not sufficiently well understood (by me at least) to be confident about how results in the single sorted case will generalise. The material in this section is such an example and so we only consider structures with one algebraic sort, and no natural number sort. We will be thinking of structures of a conventional first order language (with function and relation symbols) as structures with one algebraic sort and the boolean sort, where the interpretations of the boolean operations are obtained naturally from the interpretations of the relation symbols. The result, first proved in [Kfoury and Park, 1974], was also independently proved by me. It really provided the motivation for the rest of the work in the thesis, so it is presented here, together with its proof.

Proposition 2.5.1. Let \( L \) be a first order language, and let \( T \) be a theory over \( L \). Let \( K \) be the class of all models \( M \) of \( T \) and let \( P \) be a definable function over \( L \). Then the following are equivalent:

1. \( P^M \) is total for each \( M \in K \);
2. there exists a basic \( S \) such that \( P \) and \( S \) are \( K \)-equivalent.

Proof. We prove the implications (2) \( \Rightarrow \) (1) \( \Rightarrow \) (3) \( \Rightarrow \) (2), where \( P \) is the scheme

\[
P = \{ E_{1p}(x) \land \ldots \land E_{kp}(x) \rightarrow t_p(x) \mid p \in P \}
\]

and (3) is

\[
(3) \text{ there exists a finite subset } P' \subset P \text{ such that }
\]

\[
(a) \text{ for each } p \in P, E_{1p}(x) \land \ldots \land E_{kp}(x) \text{ is consistent with } T \text{ iff } p \in P';
\]

\[
(b) \ T \vdash \forall x. \bigvee_{p \in P'} E_{1p}(x) \land \ldots \land E_{kp}(x).
\]
The implication (2) \(\Rightarrow\) (1) follows from the fact that a basic program is total on every structure.

We can see that condition (3) implies

\begin{enumerate}
\item[(a)] \(P\) is \(K\)-equivalent to the eds \(P' = \{ E_{1_p}(x) \land \ldots \land E_{k_p}(x) \rightarrow t_p(x) \mid p \in P' \}\);
\item[(b)] there is a basic scheme extending \(P'\) which is \(K\)-equivalent to \(P'\);
\end{enumerate}

and so implies (2).

Now suppose (1). Let \(P' = \{ p \in P \mid E_{1_p}(x) \land \ldots \land E_{k_p}(x) \text{ is satisfiable on some } \mathcal{M} \in K \}\). We will show \(P'\) is finite. So, suppose not. Consider the theory

\[ T' = \{ \forall x.\neg(E_{1_p}(x) \land \ldots \land E_{k_p}(x)) \mid p \in P' \} \]

We will show \(T \cup T'\) is consistent which will give us a model \(\mathcal{M}\) of \(T \cup T'\) (and hence in \(K\)) on which \(P\) is partial. Using the compactness theorem, it will suffice to show that \(T \cup T'\) is finitely satisfiable, and this will be guaranteed if for each finite subset \(U \subset T\), there is an \(\mathcal{M} \in K\) with \(\mathcal{M} \models U\).

So, pick a finite \(U \subset T\). Since \(P'\) is infinite there exists an \(p \in P'\) with \(\forall x.\neg(E_{1_p}(x) \land \ldots \land E_{k_p}(x)) \notin U\); since \(p \in P'\) we have a model \(\mathcal{M} \in K\) with \(\mathcal{M} \models \exists x. E_{1_p}(x) \land \ldots \land E_{k_p}(x)\). But then by the exclusivity condition we have \(\mathcal{M} \not\models U\).

The condition (3) (a) motivates the following definition:

**Definition 2.5.1.** Let \(L\) be a many sorted language,

\[ P = \{ E_{1_p}(x) \land \ldots \land E_{k_p}(x) \rightarrow t_p(x) \mid p \in P \} \]

be a definable function over \(L\) and \(\mathcal{M}\) an \(L\)-structure. We say \(P\) *unwinds on* \(\mathcal{M}\) if there exists a finite \(P' \subset P\) such that if \(E_{1_g}(x) \land \ldots \land E_{k_g}(x)\) is satisfiable on \(\mathcal{M}\) then \(p \in P'\).

For the machine based models of computation, the definition of unwinding is usually given as

**Definition 2.5.2.** Let \(L\) be a many-sorted language, \(P\) a program over \(L\) and \(\mathcal{M}\) an \(L\)-structure. We say \(P\) *unwinds on* \(\mathcal{M}\) if there exists an \(n \in \mathbb{N}\) such that
if P terminates at a point $m \in M$, it does so after no more than $n$ steps.

We must ensure that if we fix on a translation $\theta: \text{fapCS}_L \rightarrow G_\theta(L)$ then

(*) for every fapCS P over L and every L-structure $M$,

$$P \text{ unwinds on } M \Leftrightarrow \theta(P) \text{ unwinds on } M.$$  

In other words, the translation must behave in a 'similar' way to the original program. We will not go into the details of how we ensure this; there are examples of how to construct schemes from programs in many of the discussions of translatability in the literature, and we direct the reader to those proofs e.g. in [Shepherdson, 1985]. In what follows, we will assume that the unwinding of programs and schemes is respected by the translation in that (*) holds.

Given a class of programs or definable functions $C$ over L, we say $M$ has the **unwind property for $C$** if every $P \in C$ unwinds on $M$.

A closely related property is the truth-table property:

**Definition 2.5.3.** Let L be a many sorted language, $C$ a class of programs over L and $M$ an L-structure. We say $M$ has the **truth-table property for $C$** if every $P \in C$ is equivalent to a loop free scheme on $M$.

If a program unwinds on a structure then it is clearly equivalent to a loop free scheme on that structure; the converse is not true, however:

Consider the standard structure $\mathbb{N}$ of arithmetic over a language including addition, subtraction, zero and successor. Let $P$ be the program implementing predecessor simply using the operations zero and successor. Then $P$ clearly does not unwind on $\mathbb{N}$ but it is equivalent to a loop free scheme on $\mathbb{N}$ (which uses the subtraction operation, for example).

These results generalise: so a structure with the unwind property plainly has the truth-table property, but there exist structures with the truth table property but not the unwind property.
This result is proved in [Kfoury, 1983]. There are several algebraic characterisations of the unwind property, and these are discussed at length in that paper.
2.6 Program properties as statements in infinitary logic

In [Engeler, 1968], a subset of the infinitary logic $L_{\omega_1 \omega}$ called AP (for *algorithmic properties*) is exhibited with the following property:

**Proposition 2.6.1.** Let $L$ be a finite language. Then there exists an effective procedure which associates to each program $P$ over $L$ a formula $\phi$ of AP such that for all structures $\mathcal{M}$ of $L$, $P^{\mathcal{M}}$ is total $\iff \mathcal{M} \models \phi$.

The subset AP is defined using regular formulas and allowing only certain infinitary disjunctions, so it is countable. The motivation for this work came principally from two considerations:

1. to determine exactly which formulas of $L_{\omega_1 \omega}$ correspond to program properties;
2. to give a constructive view (the effective procedure) of the correspondence between the formulae and the programs.

Our motivation for writing infinitary formulae or sentences corresponding to program properties is different however; we will be using them simply to express algorithmic and other properties formally and precisely. In addition, having generalised programs to non-effective definitions, the language AP will no longer be adequate to express properties such as definedness or totality.

In this section then, we will be introducing a subset of the sentences in $L_{\omega_1 \omega}$ (we do not use the word *fragment* as this is defined as a subset with certain closure properties such as in [Makkai, 1977]) which we will use to describe algorithmic and other properties.
Definition 2.6.1. Let $L$ be a many sorted language and $x = (x_1, ..., x_m)$ be a tuple of variables associated to sorts $i_1, ..., i_m$ respectively; the subset $SP(x)$ (for scheme properties) is defined inductively by the following rules:

1. every boolean sorted term $t(x)$ over $x$ is in $SP(x)$;
2. if $\phi$ is in $SP(x)$ then $\neg \phi$ is in $SP(x)$;
3. if $\{\phi_i\}_{i \in I}$ is a countably indexed set of formulas in $SP(x)$, then the infinitary disjunction $\bigvee_i \phi_i$ and infinitary conjunction $\bigwedge_i \phi_i$ are in $SP(x)$;
4. that's all.

Then the language $SP_L$ is the set of all existentially and universally quantified formulae from $SP(x)$ for $x$ a tuple of sorted variables over $L$; i.e.

$$SP_L = \{ \forall x. \phi(x) \mid x \text{ is a sorted tuple of variables over } L \text{ and } \phi(x) \in SP(x) \} \cup \{ \exists x. \phi(x) \mid x \text{ is a sorted tuple of variables over } L \text{ and } \phi(x) \in SP(x) \}.$$

Satisfaction in $SP_L$ is defined to be the same as satisfaction in $L_{\omega_1\omega}$.

Definition 2.6.2. Formulae we will frequently be using from $SP$ will be the following, obtained from a scheme $P = (E_1p(x) \land ... \land E_kp(x) \rightarrow t_p(x) \mid p \in P)$:

1. $total(P) = \forall x. \forall p \in P (E_1p(x) \land ... \land E_kp(x));$
2. $non-total(P) = \exists x. \exists p \in P (E_1p(x) \land ... \land E_kp(x));$
3. $undefined(P) = \forall x. \forall p \in P (\neg (E_1p(x) \land ... \land E_kp(x)));$
4. $defined(P) = \exists x. \exists p \in P (E_1p(x) \land ... \land E_kp(x));$
5. $undefined(P, x) = \exists p \in P (E_1p(x) \land ... \land E_kp(x));$
6. $defined(P, x, y) = \exists p \in P (E_1p(x) \land ... \land E_kp(x)) \land \exists p \in P ((E_1p(x) \land ... \land E_kp(x)) \Rightarrow y = t_p(x)).$

Other notions will be introduced as and when we need them.
2.7 Computable structures

The following definitions come from [Bergstra and Tucker, 1983].

Definition 2.7.1. Let $L$ be a many sorted language and $\mathcal{M}$ be a structure of $L$. Then $\mathcal{M}$ is said to be effectively presented when it is given an effective coordinatisation $(\alpha, \Omega)$ consisting of

1. recursive sets $\Omega_1, ..., \Omega_r \subseteq \mathbb{N}$ for algebraic sorts $i \in \{1, ..., r\}$;
2. surjections $\alpha_1, ..., \alpha_r, \alpha_i: \Omega_i \rightarrow M_i$ for $i \in \{1, ..., r\}$;
3. for each function symbol $f_j$ of type $\tau_j = (m; i_1, ..., i_m, i)$, a recursive function $f'_j: \Omega_{i_1} \times ... \times \Omega_{i_m} \rightarrow \Omega_i$ that tracks $f_j$ in the sense that

$$f_j(\alpha_{i_1}(x_1), ..., \alpha_{i_m}(x_m)) = \alpha_i(f'_j(x_1, ..., x_m))$$

for all $(x_1, ..., x_m) \in \Omega_{i_1} \times ... \times \Omega_{i_m}$.

In other words, the following diagram commutes:

$$\begin{array}{ccc}
M_{i_1} \times ... \times M_{i_m} & \xrightarrow{f_j} & M_i \\
\downarrow \alpha_{i_1} \times ... \times \alpha_{i_m} & & \downarrow \alpha_i \\
\Omega_{i_1} \times ... \times \Omega_{i_m} & \xrightarrow{f'_j} & \Omega_i
\end{array}$$

We sometimes write $\alpha: \Omega \rightarrow \mathcal{M}$ for an effective coordinatisation $(\alpha, \Omega)$.

The structure $\mathcal{M}$ is said to be computable if there exists an effective presentation $\alpha: \Omega \rightarrow \mathcal{M}$ for which the relations $\equiv_i$ defined on $\Omega_i$ by

$$x \equiv_{\alpha_i} y \iff \alpha_i(x) = \alpha_i(y) \text{ in } M_i,$$

for $i \in \{1, ..., r\}$ are recursive.

We will only be interested in the computability of structures of languages with finitely many operation symbols. The reason is that in the case where the language has infinitely many
operation symbols the above definition admits structures as being computable when intuitively we might say that they ought not to be.

As an example, consider the single sorted language with one constant, zero, one unary function symbol, successor, and a countable set \( \{ \theta_1, \theta_2, \ldots \} \) of unary relation symbols. Let \( \Sigma \subset \mathbb{N} \) be a non-r.e. set.

Let \( \mathfrak{N} \) be the \( \mathcal{L} \)-structure obtained as an expansion of the standard structure \( \mathbb{N} \) of the language \{zero, successor\} by interpreting for each \( i \)

\[ \theta_i^{\mathfrak{N}}(n) = \text{true} \iff n \text{ is the } i^{\text{th}} \text{ element of } \Sigma. \]

Clearly each of the \( \theta_i^{\mathfrak{N}} \) are recursive (and so \( \mathfrak{N} \) is computable by the definition) but the infinite collection of them is not \textit{uniformly} computable, so we should not think of \( \mathfrak{N} \) as being a computable structure.

Effective structures include those we might actually envisage implementing as datatypes. One of the benefits of studying non computable structures is that we can see which of the properties we are interested in depend upon the computability of the structures in question and which are independent. Plainly not all structures are computable — each of the sort domains in a computable structure will be countable, for instance. The following result [Bergstra and Tucker, 1983] tells us that we could have defined the notion of computability of a structure in what might at first seem to be a more restrictive way.

**Proposition 2.7.1.** A computable structure \( \mathfrak{M} \) is isomorphic to a recursive structure \( \mathfrak{R} \) of numbers each of whose algebraic domains \( R_i \) is the set \( \mathbb{N} \) of natural numbers, or the set \( \mathbb{N}_m \) of the first \( m \) natural numbers, accordingly as the corresponding domain \( M_i \) of \( \mathfrak{M} \) is infinite, or finite of cardinality \( m \).

The next two facts about computable algebras are stated without proof.
Proposition 2.7.2. Let $\mathfrak{M}$ and $\mathfrak{N}$ be structures of the languages $L_1$ and $L_2$ respectively. Then the join $\mathfrak{M} + \mathfrak{N}$ is computable if and only if each of the structures $\mathfrak{M}$ and $\mathfrak{N}$ are computable.

Proposition 2.7.3. Let $\{\mathfrak{M}_i\}_{i \in I}$ be finite family of computable structures of the language $L$ which contains no constants in the algebraic sorts. Then the disjoint union $\bigcup_{i \in I} \mathfrak{M}_i$ is computable. •

Notice that for an infinite family, the result is not always true.

Proposition 2.7.4. There exists an indexed family $\{\mathfrak{M}_i\}_{i \in I}$ of computable structures for which the disjoint union $\bigcup_{i \in I} \mathfrak{M}_i$ is not computable.

Proof. Let $\Sigma \subseteq \mathbb{N}$ be a non-r.e. set. We will denote the $i$th element of $\Sigma$ by $\sigma_i$. Let $L$ be the language with just one algebraic sort, one unary function symbol, $f$ say, and two unary relation symbols, $\phi$ and $\theta$, say. We will define the structure $\mathfrak{M}_i$ as follows.

As the carrier set $M$, take the natural numbers, $\mathbb{N}$. The function $f$ is interpreted as the usual successor function. The relation $\phi$ is interpreted to be false everywhere except on $\sigma_i$, where it is true. The relation $\theta$ is interpreted true just on $0 \in \mathbb{N}$.

![Figure 2.7.1. The structure $\mathfrak{M}_i$.](image-url)
The structure $\mathcal{M}_1$ is shown in Fig 2.7.1 and is clearly computable.

Now consider the disjoint union $\bigcup_{i \in I} \mathcal{M}_i$. This has one sort still, its carrier bijective with $\mathbb{N} \times \mathbb{N}$ and hence $\mathbb{N}$. We will show that it cannot be computable by assuming that it is, and then giving an effective procedure for deciding if a given $n \in \mathbb{N}$ is an element of $\Sigma$. So, we have recursive function $f$ and relations $\theta'$ and $\phi'$ on $\mathbb{N}$.

Let $P$ be the program (on $\mathbb{N}$) which, given a number $n$, performs the following steps:

1. searches through the numbers using the successor function, until it finds a number $m$ with $\theta'(m) = \text{true}$;
2. repeatedly applies the recursive function $f$ until it finds the $i$ for which $\phi'(f^i(m)) = \text{true}$; (this will always terminate)
3. checks to see if $i = n$ — if it is, then it terminates; otherwise it goes back to (1) and finds the next $m$ with $\theta'(m) = \text{true}$.

Then, given $n$, $P$ will terminate precisely when $n \in \Sigma$, and so $\Sigma$ is r.e. •

We will need to form infinite unions however, and will in some cases need them to be computable. The problem with the structures $\mathcal{M}_1$ in the proof of 2.7.4 is that there is no uniform way of coding them into the natural numbers. This problem leads us to make the following definition.

**Definition 2.7.2.** Let $(\mathcal{R}_n)_{n \in \omega}$ be an indexed family of recursive structures of a finite language $L$ containing no constant symbols. The family $(\mathcal{R}_n)_{n \in \omega}$ is said to be uniformly effectively presented if

1. for each algebraic sort $i$, the set $\Sigma_i = \{ n \mid R_{n_i} \text{ is finite} \}$ is recursive ($R_{n_i}$ is the domain of sort $i$ in the structure $\mathcal{R}_n$);
2. for each algebraic sort $i$, there exists a recursive function $\psi_i: \Sigma_i \to \mathbb{N}$, where $\psi_i: n \mapsto |R_{n_i}|$;
3. for each operation symbol $f$ of $L$ with type $\tau = (m; i_1, \ldots, i_m, i)$, there exists a (possibly partial) recursive function $F: \mathbb{N}^{m+1} \to \mathbb{N}$ such that for each $n$ and
each \((x_1, \ldots, x_m) \in \mathbb{R}_{m_1} \times \cdots \times \mathbb{R}_{m_m}\), \(F(n, x_1, \ldots, x_m)\) is defined and equals \(f^{\mathbb{R}_n(x_1, \ldots, x_m)}\). In other words, \(F\) represents \(f\) uniformly over all the structures \(\mathbb{R}_n\).

**Proposition 2.7.5.** The disjoint union \(\mathcal{R}\) (if it exists) of a uniformly effectively presented family \((\mathcal{R}_n)_{n \in \omega}\) of structures is computable.

**Proof.** We first recall a lemma from elementary recursion theory, which we state without proof:

**Lemma.** (Existence of pairing functions). Let \(S \subseteq \mathbb{N} \times \mathbb{N}\) be recursive and infinite; let \(k \in \mathbb{N}\). There exist recursive functions \(a, b, c,\)

\[
a: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N};
\]

\[
b: \mathbb{N} \rightarrow \mathbb{N};
\]

\[
c: \mathbb{N} \rightarrow \mathbb{N},
\]

such that \(a_k\) is a bijection with \(\mathbb{N} \setminus \{0, \ldots, k-1\}\), and

for each \((x, y) \in S\), \(b(a(x,y)) = x\) and \(c(a(x, y)) = y\). •

**Proof of 2.7.5.** We know the following:

1. for each structure \(\mathcal{R}_n\), the domain \(\mathbb{R}_{m_i}\) of algebraic sort \(i\) is either the first \(t\) numbers for some \(t\), or the whole of \(\mathbb{N}\);
2. in the disjoint union, the domain for sort \(i\) will be the disjoint union of all of these, together with a finite set \(\{\bot_{f_{1_i}}, \ldots, \bot_{f_{k_i}}\}\) of distinguished elements;
3. this domain is bijective with the set \(S_i \cup \{\bot_{f_{1_i}}, \ldots, \bot_{f_{k_i}}\}\), where \(S_i = \{(x, y) \mid x \in \omega, y \in \mathbb{R}_{m_i}\}\); \(S_i\) is recursive from conditions (1) and (2) of the definition uniform effective presentation;
4. for each algebraic sort \(i\) there exists a recursive bijection \((b_i, c_i)\) from \(S_i\) to \(\mathbb{N} \setminus \{0, \ldots, k-1\}\) with inverse \(a_i\) by the lemma;
5. if we interpret \(\bot_{f_j}\) as \(j-1\) for each \(j \in \{1, \ldots, k\}\), we have a bijection between that domain and \(\mathbb{N}\).
Taking these domains for each $i$, all that remains to be checked is that the functions as interpreted on the disjoint union are recursive. So, for an algebraic function $f$ of sort type $\tau = (m; i_1, \ldots, i_m)$ we have that

for each $x_1, \ldots, x_m \in \mathbb{N}^m,$

$$f^\mathcal{R}(x_1, \ldots, x_m) = \begin{cases} a_i(b, F(b, c_{i_1}(x_1), \ldots, c_{i_m}(x_m))) & \text{if } x_1 \geq k_1, \ldots, x_m \geq k_m, \\
 j-1 & \text{and } b_{i_1}(x_1) = \ldots = b_{i_m}(x_m) = b \text{ say otherwise where } f \text{ is } f_{i_j}. \end{cases}$$

The converse of 2.7.3 is also in general not true, as illustrated by the following examples.

**Proposition 2.7.6.** There exist structures $\mathcal{M}$ and $\mathcal{N}$ of a language $L$ for which $\mathcal{M} \cup \mathcal{N}$ is computable, but neither $\mathcal{M}$ nor $\mathcal{N}$ are computable.

**Proof.** Let $L$ be the language with just one non-algebraic sort, and one unary function symbol on that sort, $f$ say. Given a natural number $n$, we will first define the structure $\mathcal{M}_n$ of $L$.

The domain $M_n$ for $\mathcal{M}_n$ is the finite set $\{0, \ldots, n-1\}$ of natural numbers; the function $f$ is interpreted as $f^\mathcal{M}_n(x) = x+1 \pmod{n}$. The structure $\mathcal{M}_n$ is shown in Fig. 2.7.2.

Now we need a set $\Sigma \subset \mathbb{N}$ which is neither r.e. nor co-r.e.. We define $\mathcal{M}$ to be the disjoint union $\bigcup_{n \in \Sigma} \mathcal{M}_n$ and $\mathcal{N}$ to be the disjoint union $\bigcup_{n \in \Sigma} \mathcal{N}_n$. The structures $\mathcal{M}$, $\mathcal{N}$, and $\mathcal{M} \cup \mathcal{N}$ are shown in Fig. 2.7.3.

Firstly, it is easy to see that $\mathcal{M} \cup \mathcal{N}$ is just $\bigcup_{n \in \omega} \mathcal{M}_n$ and so is computable.

Now we show that if $\mathcal{M}$ is computable, then $\Sigma$ is r.e.. We give a program which, given a number $n$, will terminate just when $n \in \Sigma$. Since $\mathcal{M}$ is computable, we have a recursive function $f'$ on $\mathbb{N}$ such that the structure $\langle \mathbb{N}, f'\mathbb{N}\rangle$ is isomorphic to $\mathcal{M}$. 

Chapter 2 – Preliminaries – Computable structures 50
Let P be the program which, given a number n, performs the following steps:

1. starts at 0 and repeated applies f until it finds the i for which f^i(0) = 0;
2. compares i to n, and terminates if it is; otherwise it goes back to (1) and starts at the number 1, and so on.

Then P terminates precisely when n ∈ Σ. A similar argument gives us that Σ is co-r.e. if M is computable. •

Another fact involving computable structures we will be using is the following.

**Proposition 2.7.7.** Let L be a language and M a computable structure of L. Then for any argument sort τ = (m; i_1, ..., i_m), the set of τ-ary atomic formulas and negated atomic formulas satisfiable in M is r.e.

**Proof.** Let φ be a τ-ary atomic relation (or its negation). We give an effective procedure for deciding if φ is satisfiable in M. Given a point (m_1, ..., m_m) ∈ M_{i_1} × ... × M_{i_m}, it is decidable whether or not the atomic relation φ^M_{M_{i_1} × ... × M_{i_m}} = true, since the operations and relations are all recursive on M_{i_1} × ... × M_{i_m}. Therefore, all that is needed is a procedure which enumerates all elements of M_{i_1} × ... × M_{i_m} and tests to see if φ is true, and terminates if that is the case. This procedure will terminate if such set is found, and diverge otherwise. •
Figure 2.7.3. Structures for proof of 2.7.6.

More facts about computable algebras will be developed as and when they are required in the remainder of the thesis.
2.8 The pebble game

In the classical recursion theory of the natural numbers, extensive use is made of the following fact:

there exist primitive recursive functions $p(x,y)$, $u(x)$, and $v(y)$ such that

\[ p: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \text{ is a bijection} \]
and for all $x,y$, $u(p(x, y)) = x$ and $v(p(x, y)) = y$.

That is, $p$ is a primitive recursive pairing function with primitive recursive inverses $u$ and $v$; this enables us to store and retrieve any number of natural numbers in one number; everything can be 'coded up' into as few numbers as we like, and passed around or stored without having to dynamically reallocate storage space.

One feature of the generalisations of recursive function theory to arbitrary structures of arbitrary languages is that these pairing functions do not necessarily exist. At a stroke, many of the powerful techniques of recursive function theory, such as universality, are lost (see, for example, [Moldestat, et al., 1980]), and many new interesting points to study present themselves.

The pebble game is a technique whose applications were previously restricted to complexity theory, but which becomes important to every aspect of the theory of computability over abstract structures. Its main application is in the study of iterative program schemes, or fap without stacks, viz. fap and fapC. In an iterative scheme, there are only finitely many variables which can store elements from the algebraic domains, so there are certain values in a structure which could never be generated by one particular scheme. Further, for some structures there are sequences of domain elements which cannot be enumerated by any iterative scheme. We will use an example of such a sequence in §5.1.3. The pebble game is a useful way to study the nature of these restrictions and ways of overcoming them. We will first define the game and its rules, and then see how it relates to program schemes.
Definition 2.8.1. The pebble game is a one-person game played on a finite dag (directed acyclic graph) D. At any point in the game, some nodes of D will have pebbles on them (one pebble per node), while the remaining nodes will not. A configuration is a subset of the nodes comprising just those nodes that have pebbles on them. A move in the game consists of placing a pebble on a node, or removing a pebble from a node, according to the following rules:

1. if all the immediate predecessors of a node have pebbles on them, then a pebble may be placed on that node,
2. a pebble may be removed from any node.

A legal move is represented by an ordered pair of configurations, the second of which follows from the first according to (1) or (2). Since a leaf has no immediate predecessors, a pebble may be placed on any leaf. We will play the pebble game on trees only, where leaves are labelled, so that we can think of two leaves being either the same or distinct depending on whether or not they have the same label.

A calculation is a sequence of configurations $C_1, C_2, \ldots, C_q$ such that

(a) $C_1 = \emptyset$,
(b) each pair $(C_i, C_{i+1})$ is a legal move;

a calculation is complete when the root of D appears in some configuration $C_i$, i.e. has a pebble on it at some point.

We also define the extended pebble game, which enables us to use, in addition to rules (1) and (2), a third rule:

3. if there are two nodes $n_1$ and $n_2$ whose subtrees are identical (including leaf annotations) and there is a pebble on $n_1$ then a pebble may be placed on $n_2$.

Finally, in both forms of the game, we allow the null move (where no pebbles are added or removed) simply for convenience.
There are two measures associated with a complete calculation on a tree $D$: *time* (number of moves) and *space* (maximum number of nodes in any configuration, i.e. maximum number of pebbles on the tree at any one point). We can talk accordingly about the time and space requirements of $D$.

Now consider the following problem. Let $L$ be a language and $\tau$ an argument type over $L$. Suppose $t$ is a term in $L$ over a tuple $x$ of variables of type $\tau$. Given an interpretation $m$ of $x$ in a structure $M$ of $L$, we wish to compute the element $t^M(m)$. We have a finite set of program variables available to us and we need to work out the order in which to evaluate the subterms of $t$, which to store and which to dispose of. If the number of program variables is small, it might be difficult, or not even possible, to allocate the program variables in such a way as to be able to compute the value of $t$.

The pebble game is relevant to this problem because we can think of the term $t$ as a tree $T$; the subterms of $t$ (including $t$ itself) are the nodes and the labelled variables in $x$ are the leaves with corresponding labels. if for some operation symbol $f$ in $L$, $f(t_1, \ldots, t_m)$ is a subterm of $t$, then its descendants in $T$ are $t_1, \ldots, t_m$.

We think of the program variables as the pebbles; a pebble on node corresponds to that variable storing the value of the corresponding term. The rules of the game reflect the ways in which we can assign values to program variables: rule (1) corresponds to applying a function to a set of program variables, rule (2) freeing a variable, and rule (3) to copying. A complete evaluation of a term (symbolically) corresponds to a complete calculation in the game. We say a term $t$ over $x$ has *pebble complexity* $n$ in $x$ if $n$ is the least space requirement of any complete calculation on the corresponding tree.

We ought to stress that the pebble complexity of a term (which is obtained purely syntactically from the structure of the term) is not necessarily the same as the minimum number of program variables required to evaluate that value in a given structure; there may be other terms whose
interpretation *in that structure* have the same value. However, the following correspondence does exist.

**Proposition 2.8.1.** Let L be a language and x a tuple of type τ over L. Let t(x) be a term over x of pebble complexity n. Then there is a structure \( \mathcal{M} \) of L and interpretation m of x in \( \mathcal{M} \) such that there is no iterative program scheme P of argument type τ with fewer than n variables (including input variables) for which \( P^\mathcal{M}(m) = t^\mathcal{M}(m) \).

**Proof.** Take \( \mathcal{M} \) to be the free structure in L over x.*

We can also talk about the space requirement of a particular element m with respect to a tuple m on a particular structure \( \mathcal{M} \).

**Definition 2.8.2.** Let L be a language and \( \mathcal{M} \) a structure of L. Let \( m \in M_i \) for some sort i, and \( m = (m_{i_1}, \ldots, m_{i_m}) \in M_{i_1} \times \ldots \times M_{i_m} \) for some sorts \( i_1, \ldots, i_m \). Then we say m has *space requirement* n w.r.t. m if m is in the substructure \( \langle m \rangle \) generated by m and n is the least pebble complexity of all terms t with \( t^\mathcal{M}(m) = m \).

This notion of space requirement gives us a genuine lower bound on the number of variable required to evaluate a domain element.

**Proposition 2.8.2.** No iterative program scheme can evaluate m from m in fewer than n variables if n is the space requirement of m w.r.t. m. There exists an iterative scheme with exactly n variables evaluating m from m. •

An illustration that there really is a need to consider these issues is given by the next fact.

**Proposition 2.8.3.** Let L be a language containing an algebraic operation symbol of arity greater than 1. Then for each \( n \in \mathbb{N} \), there exists a term over L of pebble complexity greater than n. •

The first statement of this result appeared in [Friedman, 1971], but his proof doesn't work because he forgot to take account of our copying rule (3). A correct proof appears in e.g.
We conclude this section with more definitions.

**Definition 2.8.3.** Let $L$ be a language and $\mathcal{M}$ a structure of $L$. We say $\mathcal{M}$ is *structural* if for each sort type $\tau = (m; i_1, \ldots, i_m)$, there exists a number $n_{\tau}$ such that

\[
\text{for every } m \in M_{i_1} \times \ldots \times M_{i_m}, \text{ and every } m' \in \langle m \rangle, \text{ the space requirement of } m \text{ w.r.t. } m' \text{ is } \leq n_{\tau}.
\]

We say the map $\theta: \tau \mapsto n_{\tau}$ is a *structural map* for $\mathcal{M}$.

Structural structures have the following desirable property.

**Proposition 2.8.4.** Let $L$ be a language and $\theta$ be a structural map for $L$. Then there is a translation (see §2.3.5) $\theta^*: \text{eds} \rightarrow \text{fapC}$ such that for each eds $P$ and structural $\mathcal{M}$ for which $\theta$ is a structural map,

\[
P_{\mathcal{M}} = \theta^*(P)_{\mathcal{M}}.
\]

**Proof.** In [Shepherdson, 1985], the weaker statement

\[
\text{for any structural } \mathcal{M}, \text{FAPC}(\mathcal{M}) = \text{EDS}(\mathcal{M})
\]

is asserted, but the proof given is in fact a proof of the proposition. •

Many familiar algebraic systems are structural, by which we mean that every model of their theories are structural. In fact, they are often *uniformly structural* which means that there is a single structural map for the theory which does the job for each model. There are numerous examples in [Tucker, 1980].
CHAPTER THREE  ALGORITHMIC EQUIVALENCES

Our plan in this chapter is to propose and start to study the notions of observation and equivalence that will be of interest to us in the thesis.

3.1 Observations and equivalences

We define observable equivalence in terms of a general set $\Gamma$ of *observations*. Two structures will then be equivalent if the same observations may be made of each of them. Strictly speaking, if we are considering structures over a language $L$, we will be interested in a set $\Gamma(L)$ of observations, but the $(L)$ will be dropped where no confusion arises. Observations will in general be sentences over some infinitary logic [Makkai, 1977], ($L_{\omega1\omega}$ will be sufficient for our purposes) and they will usually be formulae in SP (see §2.6). Given a structure $M$ over $L$ and an observation $\gamma \in \Gamma(L)$ we say $M$ satisfies or supports the observation $\gamma$ (written $M \models \gamma$) if the infinitary formula $\gamma$ is satisfied in $M$. The definitions will typically be given in terms of a general class $C$ of definable functions, which will be a subset of $G_0$.

**Definition 3.1.1.** Given structures $M$ and $N$ over a language $L$ and a set $\Gamma$ of observations, we say $M \Gamma$-refines $N$ (written $M \prec_{\Gamma} N$ ) if for every $\gamma \in \Gamma$, $M \models \gamma \Rightarrow N \models \gamma$.

If $M \prec_{\Gamma} N$ and $N \prec_{\Gamma} M$ then we say that $M$ and $N$ are $\Gamma$-equivalent (written $M \equiv_{\Gamma} N$).

Given a class $K$ of structures of a language $L$ and two sets of observations, $\Gamma_1$ and $\Gamma_2$, we say that $\Gamma_1$ and $\Gamma_2$ are $K$-similar if for every pair of structures $M, N \in K$, $M \equiv_{\Gamma_1} N \Leftrightarrow M \equiv_{\Gamma_2} N$.

If $K$ is the set of all $L$-structures we simply say that $\Gamma_1$ and $\Gamma_2$ are similar.
Given an L-structure $\mathcal{M}$ and a set $\Gamma$ of observations over L, we will write $[\mathcal{M}]_{\Gamma}$ to denote the class of all L-structures $\Gamma$-equivalent to $\mathcal{M}$.

The following well understood notions from logic are examples of observable equivalences.

**Definition 3.1.2.** Let $L$ be a many sorted language; let $\Gamma$ be the set of all first order sentences over $L$. The equivalence $\equiv_{\Gamma}$ is *elementary equivalence*; we will write $\mathcal{M} \equiv_{\Gamma} \mathcal{N}$ and say $\mathcal{M}$ and $\mathcal{N}$ are *elementarily equivalent* if two L-structures $\mathcal{M}$ and $\mathcal{N}$ have $\mathcal{M} \equiv_{\Gamma} \mathcal{N}$.

The set of first order sentences satisfied by an L-structure $\mathcal{M}$ is written $\text{Th}_{L}(\mathcal{M})$, or $\text{Th}(\mathcal{M})$ where no confusion arises.

Two L-structures $\mathcal{M}$ and $\mathcal{N}$ are said to be *basically equivalent* (written $\mathcal{M} \equiv_{b} \mathcal{N}$) if precisely the same quantifier-free (finitary) formulae are satisfiable on each structure. That is, $\equiv_{b}$ is the observable equivalence $\equiv_{\Gamma_{\text{basic}}}$, where

$$\Gamma_{\text{basic}} = \{ \exists x. \phi(x) \mid \phi(x) \text{ is a quantifier-free formula over } L \text{ with free variables } x \}.$$

The following facts tell us how we can obtain a particular equivalence from different sets of observations.

**Proposition 3.1.1.** Let $L$ be a many sorted language. Then

1. For any two sets of observations $\Gamma_1$ and $\Gamma_2$ over $L$, $\Gamma_1 \subseteq \Gamma_2$ implies $\Gamma_2$ is stronger than $\Gamma_1$, i.e. for any two L-structures $\mathcal{M}$ and $\mathcal{N}$,

$$\mathcal{M} \equiv_{\Gamma_1} \mathcal{N} \Rightarrow \mathcal{M} \equiv_{\Gamma_2} \mathcal{N};$$

2. For an indexed family of sets of observations $\{\Gamma_i\}_{i \in I}$ over $L$ and L-structures $\mathcal{M}$ and $\mathcal{N}$, let $\Gamma = \bigcup_{i \in I} \Gamma_i$; then $(\forall i \in I. \mathcal{M} \equiv_{\Gamma_i} \mathcal{N}) \iff \mathcal{M} \equiv_{\Gamma} \mathcal{N};$

3. For any set of observations $\Gamma$, the equivalences $\equiv_{\Gamma}$, $\equiv_{\Gamma'}$, and $\equiv_{\Gamma''}$ are similar, where $\Gamma'' = \{ \neg \gamma \mid \gamma \in \Gamma \}$ and $\Gamma'' = \Gamma \cup \Gamma''$. 

Chapter 3 – Algorithmic equivalences – Observations 59
Proof. (1) Suppose $M \models_{\Gamma_2} N$; then there is an observation $\gamma \in \Gamma_2$ with (w.l.o.g.) $M \models \gamma$ and $N \not\models \gamma$. But then $\gamma \in \Gamma_1$, and so $M \models_{\Gamma_1} N$.

(2) $\Rightarrow$: Suppose $M \models_{\Gamma} N$. Then for some $i \in I$, there is a $\gamma \in \Gamma_I$ with (w.l.o.g.) $M \models \gamma$ and $N \not\models \gamma$. But then $M \models_{\Gamma_I} N$.

$\Leftarrow$: Suppose that for some $i \in I$, $M \models_{\Gamma_I} N$. Then there is a $\gamma \in \Gamma_I$ with (w.l.o.g.) $M \models \gamma$ and $N \not\models \gamma$. But then $\gamma \in \Gamma$ and $M \models_{\Gamma} N$.

(3) It suffices to show that $\Gamma$ and $\Gamma'$ are similar by (2). So, let $M$ and $N$ be $L$-structures and suppose $M \models_{\Gamma} N$. Then there is a $\gamma$ in $\Gamma$ with (w.l.o.g.) $M \models \gamma$ and $N \not\models \gamma$; but then $M \models \neg \gamma$ and $N \models \neg \gamma$ and $\neg \gamma \in \Gamma'$. So $M \models_{\Gamma} N$. •
3.2 Aspects of structure behaviour

Given two different structures of a many sorted language \( L \), we are always interested in knowing about the ways in which those structures are the same, and the ways in which they are different. If the structures are intended to be representations of a datatype or a program module, we ought to be interested to see if there are differences in the outcome of computations using the datatype. It is important to stress that we are only concerned with isomorphism invariant properties, and not properties directly concerned with a particular representation; there are occasions when discussing equivalences that we will insist on them being isomorphism invariant. Since observations are logical sentences, and logical satisfaction is defined on an isomorphism class, our observable equivalences will always be isomorphism invariant.

In this section we are going to look at different criteria we might use to compare structures, and see how these relate to program termination. In this way, we hope to justify the claim that by looking at the termination of programs alone, we can go a long way towards knowing all we want to about any particular structure. Later results in §5 will tell us the extent to which program termination on a structure in isolation determines its termination properties in wider contexts.

The discussion will involve the following fundamental concept.

**Definition 3.2.1.** Let \( L \) be a language and \( C \) a class of definable functions over \( L \). The set \( \text{tec} \) (for termination equivalence) of observations is the set \( \text{tec} = \{ \text{total}(P) \mid P \in C \} \). Thus two \( L \)-structures \( \mathcal{M} \) and \( \mathcal{N} \) will be \( \text{te} \)-equivalent when for each definable function \( P \in C \), \( P^{\mathcal{M}} \) is total iff \( P^{\mathcal{N}} \) is total. Typically, we might be interested in the classes corresponding to loop free, fap, fapC, and eds.

It is important to realise that, although we have defined \( \text{te} \)-equivalence in terms of classes of definable functions, we obtain the same equivalence as we would if we had defined the observations directly in terms of another model of computation. Formally, if \( \theta \) is a translation \( C_1 \rightarrow C_2 \), the sets \( \text{te}_{C_1} \) and \( \text{te}_{\theta(C_1)} \) of observations are similar.
3.2.1. Structures with the 'same' domains

A naive way of stating that two structures $\mathcal{M}$ and $\mathcal{N}$ behave in the same way with respect to computable functions over them is to say that every program $P$ 'computes the same function on each of them'. So, let $f$ be a computable function and $f^\mathcal{M}$ and $f^\mathcal{N}$ be its interpretations on the structures $\mathcal{M}$ and $\mathcal{N}$ respectively. We would like to be able to say that the functions $f^\mathcal{M}$ and $f^\mathcal{N}$ are equal, or 'give the same result for the same arguments', but in order to say precisely what we mean by this, it is necessary to have some notion of correspondence between the domains of the structures $\mathcal{M}$ and $\mathcal{N}$. In minimal structures (see §3.2.3) this might be provided by there being a surjective map from the term structure $T_L$ to each of them, or on other structures it might be given by some other canonical relation between the domains. But this is unsuitable for our purpose, as we are looking for criteria that can be expressed as formulae in a logical language and are therefore independent of the names of the elements of the domains, unless they can be named in the language.

3.2.3. Minimal structures

A significant body of the literature in the field of abstract data types is concerned with structures that are minimal structures, in the sense that they have no proper substructures. Since every structure has a substructure which is the set of interpretations of ground terms, it must be the case that in a minimal structure every domain element is the interpretation of a ground term.

It happens that for minimal structures termination equivalences are quite powerful at distinguishing different structures.

**Proposition 3.2.1.** Let $L$ be a non-void finite language, and $\mathcal{M}$ a minimal structure of $L$. Then

1. all minimal structures in the class $[\mathcal{M}]_{e_{\text{loop\_free}}}$ are isomorphic;
2. all structures in the class $[\mathcal{M}]_{e_{\text{eds}}}$ are isomorphic.
Proof. (1). Let \( t_1 \) be a ground term of sort \( B \) over \( L \). Let \( P_{t_1} \) be the loop free scheme

\[
\{ t_1 \to \text{true} \};
\]

let \( \mathfrak{N} \) be a minimal \( L \)-structure with \( \mathfrak{M} \equiv_{\text{loop-free}} \mathfrak{N} \). Since the termination of all of the schemes \( P_{t_1} \) is the same on \( \mathfrak{M} \) as it is on \( \mathfrak{N} \), we have that for every \( B \)-sorted ground term \( t \),

\[
t^\mathfrak{M} \equiv t^\mathfrak{N}.
\]

Now we can define the map \( \theta: \mathfrak{M} \to \mathfrak{N} \) by

for each \( m \in \mathfrak{M} \),

\[
\theta(m) = t^\mathfrak{N} \quad \text{if for some ground term } t, m = t^\mathfrak{M}.
\]

Since equality is a basic relation, \( \theta \) is well-defined and injective. Since \( \mathfrak{N} \) is minimal, \( \theta \) is surjective. The map \( \theta \) is a homomorphism since if \( f \) is an operation symbol from \( L \) of sort type \( \tau = (m; i_1, \ldots, i_m, i) \) and \( t_1, \ldots, t_m \) are ground terms of sorts \( i_1, \ldots, i_m \) respectively,

\[
\theta(f(m, t_1^\mathfrak{M}, \ldots, t_m^\mathfrak{M})) = f(t_1^\mathfrak{N}, \ldots, t_m^\mathfrak{N}) = t^\mathfrak{M} = f(t_1^\mathfrak{N}, \ldots, t_m^\mathfrak{N}) = f(t_1^\theta, \ldots, t_m^\theta); \]

so \( \theta \) is an isomorphism and \( \mathfrak{M} \equiv \mathfrak{N} \).

(2) By (1) and the fact that \( \equiv_{\text{eds}} \)-equivalence is stronger than \( \equiv_{\text{loop-free}} \)-equivalence, it is sufficient to show that if an \( L \)-structure \( \mathfrak{N} \) has \( \mathfrak{M} \equiv_{\text{eds}} \mathfrak{N} \) then \( \mathfrak{N} \) is minimal. Now let \( i \) be an algebraic sort of \( L \) and let \( \{t_1, \ldots, t_k, \ldots\} \) be the (r.e.) set of ground terms (ordered according to some suitable Gödel ordering) of sort \( i \) over \( L \). Let \( P_i \) be the scheme of argument sort \( (1; i) \) and input variable \( x \)

\[
P_i = \{ x=t_1 \} \cup \{ x \neq t_1 \land \ldots \land x \neq t_k \land x=t_{k+1} \mid k \geq 1 \};
\]

each \( P_i \) is an eds because the set of clauses is r.e.. Now every \( P_i \) will be total on an \( L \)-structure iff it is minimal; since \( \mathfrak{M} \) is minimal and \( \mathfrak{M} \equiv_{\text{eds}} \mathfrak{N} \) we have that \( \mathfrak{N} \) is minimal.

3.2.3. Other structures

When structures are not minimal, and there is no obvious correspondence between their domains, we cannot compare two structures by looking at the outcome of one particular
function alone, except to ask whether or not it is total, or perhaps to ask whether or not it is defined somewhere. What we can do, however, is compare different functions or relations on one structure, and then ask whether or not their relationship is the same on another structure.

Motivated by this, in this subsection we will introduce several different notions of equivalence based on these kinds of concerns. They will all turn out to be instances of te-equivalence, so there will be little need to remember them all in reading the remainder of the thesis. However, in order to help in reading this section, here are the equivalences we will be discussing.

(1) nf  (nowhere false) the same B-valued programs are true on the whole of their domain;
(2) sd  (same domain) the same programs have coincident domains;
(3) nd  (nowhere defined) the same programs have empty domains.

Suppose then, for example, we have two programs P and Q, say, and a structure M on which their interpretations P^M and Q^M are equal on the intersection of their domains. Given another structure N, we might like to know if the two functions P^M and Q^M are similarly equal. We can generalise this idea with the following definition.

**Definition 3.2.2.** Let L be a language and C a class of definable functions. The set nf_C (for nowhere false) of observations is the set

\[ n_{f_C} = \{ \neg \exists x. \text{defined}(P, x, \text{false}) \mid P \text{ is a boolean valued function from } C \} \]

That is, two structures will be nf_C-equivalent if for every definable relation P in C, P^M is true on the whole of the domain of P^M \iff P^N is true on the whole of the domain of P^N.

In the case of comparing the functions P and Q in the previous example, the relation of interest is the relation P=Q, which will be true on the whole of its domain on a structure iff P and Q agree on the intersection of their domains.

The next fact tells us that nf_C-equivalence is not really as strong as it might at first seem.
Proposition 3.2.2. For every language $L$ and class $C$ of definable functions over $L$ containing every basic scheme, $\eta_C$ is similar to $t_{\text{loop free}}$.

Proof. Let $\mathcal{M}$ and $\mathcal{N}$ be $L$-structures, and suppose $\mathcal{M} \not\equiv_{\eta_C} \mathcal{N}$. Let $P$ be a boolean valued scheme in $C$ for which (w.l.o.g.) $P^\mathcal{M}$ is true on the whole of its domain, but there exists a tuple $n \in \mathcal{N}$ for which $P^\mathcal{N}(n) = \text{false}$. Thus there must be a clause of $P$

$$E_1(x) \land E_2(x) \land ... \land E_k(x) \rightarrow t(x)$$

for which $E_1^\mathcal{N}(n) \land ... \land E_k^\mathcal{N}(n) \equiv \text{true}$ and $t^\mathcal{N}(n) = \text{false}$, but for which there exists no $m \in \mathcal{M}$ such that both $E_1^\mathcal{M}(m) \land ... \land E_k^\mathcal{M}(m) \equiv \text{true}$ and $t^\mathcal{M}(m) = \text{false}$. Now let $Q$ be the loop free scheme

$$Q = \{ -E_1(x) \rightarrow \text{true} \}$$

$$\cup \{ E_1(x) \land ... \land E_{j-1}(x) \land -E_j(x) \rightarrow \text{true} \mid 1 < j \leq k \}$$

$$\cup \{ E_i(x) \land E_2(x) \land ... \land E_k(x) \land t(x) \equiv \text{true} \rightarrow \text{true} \}.$$ 

which, given $x$, evaluates $E_1(x) \land ... \land E_k(x)$ and terminates if it evaluates to false and $t(x)$ evaluates to true, and diverges otherwise. Then $Q^\mathcal{M}$ is total, but $Q^\mathcal{N}$ is not. So $\mathcal{M} \not\equiv_{\text{loop free}} \mathcal{N}$.

Conversely, suppose there is a loop free scheme $Q$

$$Q = \{ E_{s}(x) \land E_{2s}(x) \land ... \land E_{ks}(x) \rightarrow t_s(x) \mid 1 \leq s \leq r \}$$

such that (w.l.o.g.) $Q^\mathcal{M}$ is total, but $Q^\mathcal{N}$ is not. Writing the formula

$$\phi(x) \equiv \bigwedge_{1 \leq s \leq r} (\neg (E_{1s}(x) \land E_{2s}(x) \land ... \land E_{ks}(x)))$$

in disjunctive normal form

$$\phi(x) \equiv \bigvee_{1 \leq s \leq q} (F_s(x))$$

say,

let $Q_B$ be a basic scheme

$$Q_B = \{ E_{s}(x) \land E_{2s}(x) \land ... \land E_{ks}(x) \rightarrow \text{true} \mid 1 \leq s \leq t \}$$

$$\cup \{ F_s(x) \rightarrow \text{false} \mid 1 \leq s \leq q \}.$$
The functions $Q_B^M$ and $Q_B^N$ are both total, but $Q_B^M$ is true everywhere, whereas $Q_B^N$ is not.

Hence $M \not\equiv_{nfC} N$.

Alternatively, we might wish to say something about the corresponding domains of the functions $P$ and $Q$ in our first example. This motivates the following definition.

**Definition 3.2.3.** Let $L$ be a language and $C$ a class of definable functions over $L$. The set $sd_C$ (for *same domain*) of observations is the set

$$sd_C = \{ \forall x. \text{undefined}(P, x) \leftrightarrow \text{undefined}(Q, x) \mid P, Q \in C \}.$$ 

We can see that in most cases, $sd$ is the same as $te$.

**Proposition 3.2.3.** Suppose that $C$ is a class of programs closed under basic constructs. Then the sets $te_C$ and $sd_C$ are similar.

**Proof.** The fact that $sd$ equivalence implies $te$ equivalence is straightforward, as we can compare the termination of any program with that of a basic program (which will always be total) of the appropriate type.

Now suppose that two $L$-structures $M$ and $N$ are not $sd_C$-equivalent; so there exist schemes $P$ and $Q$ in $C$ for which (w.l.o.g.) $\text{domain}(P^M) = \text{domain}(Q^M)$ but $\text{domain}(P^N) \neq \text{domain}(Q^N)$.

So, w.l.o.g. again, there is a tuple $n \in N$ for which $n \in \text{domain}(P^N)$ but $n \notin \text{domain}(Q^N)$; then there is a clause $E_1(x) \land ... \land E_k(x) \rightarrow t(x)$ of $P$ with $E_1^N(n) \land ... \land E_k^N(n) \equiv \text{true}$.

Let $R$ be the boolean valued loop free scheme (necessarily in $C$)

$$R = \{ \neg E_1(x) \rightarrow \text{false} \}$$

$$\cup \{ E_1(x) \land \ldots \land E_{j-1}(x) \land \neg E_j(x) \rightarrow \text{false} \mid 1 \leq j \leq k \}$$

$$\cup \{ E_1(x) \land E_2(x) \land \ldots \land E_k(x) \land t(x) \rightarrow \text{true} \}.$$ 

Now consider the scheme $T = \text{if}(R, Q, \text{true})$ (§2.4.2) which, given a tuple $x$, first executes $R$; if $R(x) \equiv \text{false}$, then $T$ terminates; otherwise $T$ goes on to evaluate $Q(x)$.
Now \( T^\mathcal{M}(n) \) is clearly undefined. However, for any \( m \in \mathcal{M} \), \( R^\mathcal{M}(m) \equiv \text{true} \) implies that \( Q^\mathcal{M}(m) \) is defined; hence \( T^\mathcal{M} \) is total. So \( \mathcal{M} \not\equiv \text{teC} \mathcal{N} \).

Another aspect program termination behaviour we might like to consider is the following.

**Definition 3.2.4.** Let \( L \) be a language and \( C \) a class of programs over \( L \). The set \( nd_C \) (for *nowhere defined*) of observations is the set

\[
nd_C = \{ \text{undefined}(P) \mid P \in C \}.
\]

The next fact tells us that in most cases, \( nd \)-equivalence is the same as \( re_{\text{loop free}} \)-equivalence.

**Proposition 3.2.4.** Let \( L \) be a language and \( C \) a class of definable functions over \( L \) containing every loop free scheme. The the sets \( nd_C \) and \( re_{\text{loop free}} \) are similar.

**Proof.** We show that for all \( L \)-structures \( \mathcal{M} \) and \( \mathcal{N} \), \( \mathcal{M} \equiv_{nd_C} \mathcal{N} \iff \mathcal{M} \equiv_{re_{\text{loop free}}} \mathcal{N} \).

\( \Rightarrow \): Suppose that \( \mathcal{M} \not\equiv_{re_{\text{loop free}}} \mathcal{N} \). Then there exists a loop free scheme, \( P \) say, for which we can assume w.l.o.g. that \( P^\mathcal{M} \) is total but \( P^\mathcal{N} \) is partial.

Let \( P \) be the scheme

\[
\{ E_{1g}(x) \land ... \land E_{kg}(x) \rightarrow t_s(x) \mid 1 \leq s \leq t \}.
\]

We wish to consider a scheme \( P^c \) which is undefined at an interpretation precisely when \( P \) is defined. Let \( \phi(x) \) be the formula

\[
\phi(x) \equiv \neg (\bigvee_{1 \leq s \leq t} (E_{1s}(x) \land ... \land E_{ks}(x)));
\]

writing \( \phi \) in disjunctive normal form we obtain

\[
\phi(x) \equiv \bigvee_{1 \leq s \leq v} (F_{1s}(x) \land ... \land F_{ls}(x))
\]

for some atomic relations \( F_{ij} \). Now let \( P^c \) be the scheme

\[
P^c = \{ F_{1s}(x) \land ... \land F_{ls}(x) \rightarrow \text{true} \mid 1 \leq s \leq v \};
\]
clearly $P^M$ is defined nowhere but $P^N$ is defined somewhere; so $M \neq_{ndC} N$.

$\Leftarrow$: Suppose that $M \neq_{ndC} N$. Then we can assume w.l.o.g. that there is a scheme $P$ say, in $C$, for which $P^M$ is defined nowhere but $P^N$ is defined somewhere, at $n$, say. As before, we consider the clause $E_1(x) \land \ldots \land E_k(x) \rightarrow t(x)$ for which $E_1^N(n) \land \ldots \land E_k^N(n)$; let $Q$ be the loop free scheme

$$Q = \{ -E_1(x) \rightarrow \text{false} \}$$

$$\cup \{ E_1(x) \land \ldots \land E_{j-1}(x) \land -E_j(x) \rightarrow \text{false} \mid 1 < j \leq k \}.$$ 

Then $Q^M$ is total, whereas $Q^N$ is undefined at $n$. Hence $M \neq_{te\text{loop free}} N$. •

The next result tells us that $te\text{loop free}$-equivalence is the same as a more familiar equivalence.

**Proposition 3.2.5.** The equivalences $te\text{loop free}$-equivalence and basic equivalence are the same.

**Proof.** We show that the sets of observations $nd\text{loop free}$ and $\Gamma_{\text{basic}}$ (§3.1) are similar.

For, let $P = \{ E_{i_s}(x) \land \ldots \land E_{k_s}(x) \rightarrow t_s(x) \mid 1 \leq s \leq r \}$ be a loop free scheme; the observation $\text{undefined}(P)$ is simply the sentence $-\exists x. \lor_{1 \leq s \leq r} (E_{i_s}(x) \land \ldots \land E_{k_s}(x))$. Conversely, since every quantifier-free formula can be written in disjunctive normal form, we can see that every $\Gamma_{\text{basic}}$ observation is logically equivalent to the negation of the $nd\text{loop free}$ observation corresponding to that disjunction. Thus $nd_{\text{loop free}} = \{ -\gamma \mid \gamma \in \Gamma_{\text{basic}} \}$; so the equivalences are similar by 3.1.1. •

Since elementary equivalence is stronger than basic equivalence we have the following consequence.

**Proposition 3.2.6.** Elementary equivalence implies $te\text{loop free}$-equivalence. •

In summary, we looked at the sets
(1) \( nf \) (nowhere false) the same \( B \)-valued programs are true on the whole of their domain;

(2) \( sd \) (same domain) the same programs have coincident domains;

(3) \( nd \) (nowhere defined) the same programs have empty domains of observations and concluded that

(1) for most \( C \), \( nf_C \) is similar to \( nd_C \), \( te_{loop \ free} \) and \( \Gamma_{basic} \);

(2) for most \( C \), \( sd_C \) is similar to \( te_C \).

3.2.4. Expanding structures

For the purposes of applying termination equivalence to real systems, it is possible that we might be interested in different structures which share a common reduct. Take the following example. Suppose that, in our modular programming language, we have a type \( \text{real} \) and given operations on it, such as \( \text{plus}, \text{minus}, \text{multiply}, \) and \( \text{divide} \). Suppose that one of our modules is intended to be an implementation of a square root algorithm, which, for every positive real number in the data type, outputs another real number as an approximation to the square root. In an effective modularisation, we will want to encapsulate that routine, and just have a procedure \( \text{square root} \) which calls the routine for us. We can think of the module as a structure over original \( \text{real} \) language, expanded with the interpretation of a new operator symbol, \( \text{root} \).

We might, of course, try different ways of implementing the square root algorithm; these will give us different expansions of the old structure. Since we will want to program using that module, we might ask how the different expansions affect the outcome of programs using those implementations. The work in this section is directed toward answering this question.

The first case we consider is where the implementation of the algorithm coincides with a definable function.

**Proposition 3.2.6.** Let \( L \) be a language and \( \mathcal{M} \) a structure of \( L \). Suppose that \( L' \) extends \( L \) with operation and relation symbols \( f_1, \ldots, f_p, \ldots \) and that \( \mathcal{N} \) and \( \mathcal{R} \) are expansions of \( \mathcal{M} \) to \( L' \);
suppose also that the interpretations $f^1_M, \ldots, f^n_M$ on $M$ coincide with the interpretations $g^1_M, \ldots, g^n_M$ of $G_0(L)$-definable functions $g_1, \ldots, g_i, \ldots$ on $M$.

Then

1. if $\mathfrak{R} \equiv_{\text{nf}} \mathfrak{N}$ then $f^1_M = g^1_M, \ldots, f^n_M = g^n_M, \ldots$;
2. if $\mathfrak{R} \equiv_{\text{nf}} \mathfrak{N}$ then $\mathfrak{R} \equiv \mathfrak{N}$.

Proof. (1) Suppose $g_i$ is the scheme $g_i = \{ E_{i_1}(x) \wedge E_{i_2}(x) \wedge \ldots \wedge E_{i_k}(x) \rightarrow t_i(x) \mid s \in S \}$; then for each $s \in S$, the loop free scheme $P_{i,s}$ defined by

$$P_{i,s} = \{ E_{i_1}(x) \wedge E_{i_2}(x) \wedge \ldots \wedge E_{i_k}(x) \wedge (f_i(x) = t_i(x)) \rightarrow \text{true} \}$$

is true on the whole of its domain in $\mathfrak{R}$; therefore by $\text{nf}$-equivalence (guaranteed by basic equivalence by 3.2.2 and 3.2.5) it is true on the whole of its domain on $\mathfrak{R}$; so for each $g_i$ and clause $s$, the interpretation of $g_i$ coincides with that of $f_i$ on $\mathfrak{R}$.

(2) The structures $\mathfrak{R}$ and $\mathfrak{N}$ are identical expansions of $M$ to $L'$, so they are isomorphic.

This tells us, then, that if two expansions are distinct, we can distinguish them using loop free programs.

The more general case, where the extra operators are not definable, is not as straightforward, however.

Proposition 3.2.7. There exist languages $L$ and $L'$ with $L'$ extending $L$, with $L$-structure $\mathfrak{R}$ and expansions $\mathfrak{M}$ and $\mathfrak{N}$ of $\mathfrak{R}$ to $L'$ such that $\mathfrak{N} \equiv_{\text{teG}_0} \mathfrak{M}$ but $\mathfrak{N} \not\equiv \mathfrak{M}$.

Proof. We will use an example from §5. In the proof of 5.1.17 there are two countably infinite structures $\mathfrak{M}$ and $\mathfrak{N}$ of a single sorted language $L'$ which are non-isomorphic but are termination equivalent for $G_0$. Now take $L$ to be the language $L'$ without any operation symbols. The reducts $\mathfrak{M}_L$ and $\mathfrak{N}_L$ are isomorphic since they are both just an infinite set with no operations interpreted on it.
The discussions in this section suggest then that termination equivalence seems to determine several aspects of program behaviour that we might be interested in; this is why termination equivalence is of greatest interest to us in the remainder of the thesis. The remainder of this chapter is concerned with a few other algorithmic notions of equivalence and their basic properties.
3.3 Identifiability

The remaining sections of this chapter are concerned with 'miscellaneous' notions of algorithmic equivalence; the work in subsequent chapters will concentrate almost exclusively on \( te \)-equivalence and these other ideas will not feature.

In the discussion in §3.2 we briefly mentioned how we would be able to compare the outcome of functions on a structure if we had a notion of correspondence between the domains of a structure. We observed that this would be the case where elements were the interpretation of a particular term, and were 'named'. In this section, we look at a slightly more general way in which elements can be named.

Consider the following example. Let \( L \) be the language of fields; so we have one algebraic sort, constants 0 and 1, and operations add, multiply, subtract and divide, say. Then the elements of any field include the interpretations of the terms, which correspond the the rationals, together will all the terms built from things like 1/0. But using computable functions, we can identify more elements than just these; in the reals, for example there will be a unique element \( x \) for which the program evaluating the boolean \((x \times x \times x = 2)\) will return true. We have effectively identified the element \(3\sqrt{2}\) from its behaviour in a computation, and we can think of that program (together with the information that there is an element satisfying it) as naming \(3\sqrt{2}\).

It is also clear that in any \( L \)-structure \( nd_{\text{loop free}} \)-equivalent to the reals, there will also have to be an element with this property. We can name all of the algebraic numbers in addition to the rationals. We might then think about comparing structures by the outcome of programs on these namable elements, or by looking at which of the namable elements are present. The following definitions get us started.

Let \( C \) be a class of definable functions over a language \( L \).

Definition 3.3.1. Let \( \mathcal{M} \) be a structure over a language \( L \). Given an argument sort type \( \tau = (m; i_1, \ldots, i_m) \) of \( L \), a \( \tau \)-tuple \( m \in \mathcal{M} \) is uniquely identified if there is a \( \tau \)-ary scheme \( P \) from \( C \) such that \( \text{domain}(P^m) = \{m\} \).
An element $m \in M_i$ for some $i \in \text{Sort}$ is *uniquely identifiable* if $m \in m$ for some uniquely identified $m$.

The uniquely identifiable elements are those we are able to name, without necessarily being able to generate them in a program in any way. We see immediately that it is not necessary to be specific about the class $C$ of definable functions, provided it contains all loop free schemes.

**Proposition 3.3.1.** Suppose $m$ is a tuple uniquely identified by a scheme $P \in C$, where

$$P = \{ E_{s_1}(x) \land \ldots \land E_{s_k}(x) \rightarrow t_s(x) \mid s \in S \}. $$

Then there is a loop free scheme $P'$ whose value sort is $B$ which uniquely identifies $m$.

**Proof.** We know $P^M(m)$ is defined, by clause $s_0$ say of $P$. Then $m$ is uniquely defined by the scheme

$$P' = \{ E_{s_0}(x) \land \ldots \land E_{k_0}(x) \rightarrow \text{true} \}. $$

We also see that the collection of uniquely identifiable elements has convenient closure properties.

**Definition 3.3.2.** Let $\mathcal{M}$ be a structure. For each $i \in \text{Sort}$, the subset $M_i^* \subseteq M_i$ is defined to be the set of all uniquely identifiable elements of sort $i$; that is,

$$M_i^* = \{ m \in M_i \mid m \text{ is uniquely identifiable} \}. $$

**Proposition 3.3.2.** For any $\mathcal{M}$, $(M_i^*)_{i \in \text{Sort}}$ is closed under the operations of $L$.

**Proof.** Firstly, for any constant symbol $c$ of sort $i$ in $L$, $(c)$ is uniquely identified by the scheme $P(x)$, where

$$P(x) = \{ x=c \rightarrow \text{true} \}. $$

Now suppose that $f$ is an operation symbol of $L$ of type $\tau = (m; i_1, \ldots, i_m, i)$ and that the tuple $(m_1, \ldots, m_m) \in M_{i_1}^* \times \ldots \times M_{i_m}^*$. Then for some tuples $m_1, \ldots, m_m$ we have that for each $j \in \{1, \ldots, m\}$, $(m_j, m_j)$ (the tuple whose first element is $m_j$ and remainder are $m_j$) is uniquely identified, by schemes $P_j$ say.
Now let $Q$ be the scheme

$$Q(y, x_1, \ldots, x_m, x_1, \ldots, x_m) = \bigwedge \{ y = f(x_1, \ldots, x_m) \rightarrow \text{true} \}, P_1, \ldots, P_m;$$

the conjunction of $B$-sorted schemes was defined in §2.4.2.

Then $Q$ uniquely identifies the tuple $(f(m_1, \ldots, m_k), m_1, \ldots, m_k, m_1, \ldots, m_k)$ of $\mathcal{M}$ so $f(m_1, \ldots, m_k) \in M_i^*$. 

So the uniquely identifiable elements form a substructure provided there is at least one uniquely identifiable element of each sort; for non void languages this will always be the case. We make the following definition.

**Definition 3.3.3.** Let $\mathcal{M}$ be a structure over a language $L$ such that each set $M_i^*$ is non-empty. We denote by $\mathcal{M}^*$ the substructure of $\mathcal{M}$ whose domains are the subsets $M_i^* \subseteq M_i$.

We might like now to compare structures by these identifiable substructures.

**Definition 3.3.4.** The set $id$ (for identifiability) of observations is the set

$$id = \{ \exists x, y. \text{defined}(P, x, y) \mid P \in \text{loop free} \}.$$ 

We say two structures $\mathcal{M}$ and $\mathcal{N}$ are identifiably equivalent if $\mathcal{M} \equiv_{id} \mathcal{N}$.

**Proposition 3.3.3.** Suppose that for two structures $\mathcal{M}$ and $\mathcal{N}$, we have $\mathcal{M} \equiv_{id} \mathcal{N}$. Suppose also that the substructure $\mathcal{M}^*$ exists. Then

1. $\mathcal{N}^*$ exists;
2. if $m \in M_i^*$ for some $i \in \text{Sort}$ with both $(m, m_1)$ and $(m, m_2)$ uniquely identified by $P_1$ and $P_2$ respectively, then $n_1 = n_2$ whenever $(n_1, n_1) \in \text{domain}(P_1^\mathcal{N})$ and $(n_2, n_2) \in \text{domain}(P_2^\mathcal{N})$.
3. There is a natural isomorphic embedding $f: \mathcal{M}^* \rightarrow \mathcal{N}^*$.
4. $\mathcal{M}^* \equiv \mathcal{N}^*$.
Proof. (1) Let $i \in \text{Sort}$ and $m_i \in M_i^*$. Then there exists an $m \in M$ with $(m_i, m)$ uniquely identified by $P_{m_i}$, say. Since $M$ and $N$ are id equivalent, there exists $n_i \in N_i^*$ and $n \in N$ with $(n_i, n)$ uniquely identified by $P_{m_i}$. Thus $n_i \in N_i$.

(2) Suppose not; then there exist $n_1$ and $n_2 \in N_i^*$ with both $(n_1, n_1)$ and $(n_2, n_2)$ uniquely identified by $P_1$ and $P_2$ resp., but $n_1 \neq n_2$. Consider the scheme $Q$ given by

$$Q(x_1, x_2, x_3, x_4) \equiv \bigwedge (P_1(x_1, x_1), P_2(x_2, x_2), \{x_1 \neq x_2 \rightarrow \text{true}\}).$$

Then $Q$ uniquely identifies nothing on $M$ but identifies $(n_1, n_1, n_2, n_2)$ on $N$. So $n_1 = n_2$ whenever $(n_1, n_1) \in \text{domain}(P_1)$ and $(n_2, n_2) \in \text{domain}(P_2)$.

(3) Let $m \in M^*$. Then there is a tuple $(m, m)$ which is uniquely identified on $M$, by $P$ say. By id-equivalence, there is a corresponding $(n, n)$ uniquely identified by $P$ on $N$. Define $f(m) = n$. By (2), this map is well-defined.

To see that $f$ is an isomorphic embedding, let $\varphi(x)$ be a $\tau$-ary term where $\tau = (k; i_1, \ldots, i_k, B)$, and let $m = (m_1, \ldots, m_k)$ be a tuple of $M^* \subset M$ of type $(k; i_1, \ldots, i_k)$. Suppose that for each $j$, $1 \leq j \leq k$, $(m_j, m_j)$ is uniquely identified on $M$, by $P_j$, say. Without loss of generality, we may assume that each $P_j$ is true wherever it is defined.

Let $Q$ be the scheme given by

$$Q(x_1, \ldots, x_k, x_1, \ldots, x_k) =$$

$$\bigwedge (\{\varphi(x_1, \ldots, x_k) \rightarrow \text{true}\}, P_1(x_1, x_1), \ldots, P_k(x_k, x_k)).$$

Then $Q$ is defined precisely at $(m_1, \ldots, m_k, m_1, \ldots, m_k)$ on $M$, so it uniquely identifies the tuple $(n_1, \ldots, n_k, n_1, \ldots, n_k)$ say on $N$. Then $n_i = f(m_i)$ and $\varphi(n_1, \ldots, n_k) = \text{true}$. Similarly, if $\varphi M(m_1, \ldots, m_k) = \text{false}$, we would define

$$Q(x_1, \ldots, x_k, x_1, \ldots, x_k) =$$

$$\bigwedge (\{-\varphi(x_1, \ldots, x_k) \rightarrow \text{true}\}, P_1(x_1, x_1), \ldots, P_k(x_k, x_k)).$$

Chapter 3 – Algorithmic equivalences – Identifiability 75
to obtain \( n_i = f(m_i) \) and \( \varphi(n_1, \ldots, n_k) \equiv \text{false} \).

(4) Using the same argument we can see that there is a homomorphism \( g: \mathcal{N} \rightarrow \mathcal{M} \) which is a left- and right-inverse for \( f \). Thus \( f \) is a bijection and hence an isomorphism.*

Now we can easily see that \( \text{id} \)-equivalence is guaranteed by \( nd \)-equivalence.

**Proposition 3.3.4.** Suppose \( \mathcal{M} \) and \( \mathcal{N} \) are \( L \)-structures and \( \mathcal{M} \equiv_{ndC} \mathcal{N} \). Then \( \mathcal{M} \equiv_{idC} \mathcal{N} \).

**Proof.** Suppose a \( \tau \)-ary scheme \( P \) uniquely identifies \( m \) on \( \mathcal{M} \) where \( \tau = (k; i_1, \ldots, i_k) \). Since \( P \) is defined somewhere on \( \mathcal{M} \), it is defined somewhere on \( \mathcal{N} \) by \( nd \)-equivalence. To see that it is defined uniquely on \( \mathcal{N} \), consider the \( 2k \)-ary scheme \( Q \) given by

\[
Q(x_1, x_2) \equiv \bigwedge \{ P(x_1), P(x_2), (x_1 \neq x_2 \rightarrow \text{true}) \}.
\]

Then \( Q \) is defined nowhere on \( \mathcal{M} \) and thus nowhere on \( \mathcal{N} \) by \( nd \)-equivalence. So there is a unique \( n \in \mathcal{N} \) with \( n \in \text{domain}(\varphi_\mathcal{N}) \).

To see that the reverse implication does not hold, we can simply find non isomorphic structures \( \mathcal{M} \) and \( \mathcal{N} \) for which there are no identifiable elements other than the sort constants.

**Proposition 3.3.5.** There exist structures \( \mathcal{M} \) and \( \mathcal{N} \) for which \( \mathcal{M} \equiv_{id} \mathcal{N} \) but \( \mathcal{M} \not\equiv_{ndC} \mathcal{N} \).

**Proof.** Let \( L \) be a language with one sort and one constant symbol, \( c \) say. Let \( \mathcal{M} \) be any three element \( L \)-structure and \( \mathcal{N} \) any four element \( L \)-structure. It is clear* that \( \mathcal{M} \not\equiv_{ndC} \mathcal{N} \), since the scheme given by

\[
\{(x_1 \neq c) \land (x_2 \neq c) \land (x_3 \neq c) \land (x_1 \neq x_2) \land (x_2 \neq x_3) \land (x_3 \neq x_1) \rightarrow \text{true} \}
\]

is defined somewhere on \( \mathcal{N} \) but not on \( \mathcal{M} \).

However, the only uniquely identifiable element in each of them is \( c \), this by the unary scheme \( \{ x = c \rightarrow \text{true} \} \). So \( \mathcal{M} \equiv_{id} \mathcal{N} \). •

Chapter 3 – Algorithmic equivalences – Identifiability 76
3.4 Distinguishability

In the last section, we were interested in elements of structures which were unique with a certain property; but the examples in 3.3.4 show that not every element or tuple in a structure has a property or set of properties that are not shared by other elements in that structure. The structures $\mathcal{M}$ and $\mathcal{N}$ in that example were not isomorphic, but they shared much of the same 'character'; had we been able to stick together those elements in each structure which were indistinguishable from one another, we might have had a better chance of comparing them favourably. In this section then, we look at the extent to which we can think of indistinguishable elements as being the same, and what that tells about the relationship between different structures.

It would be convenient if, given an $L$-structure $\mathcal{M}$, we were able to 'extract' the essence of its behaviour by regarding as equivalent those elements which could not be told apart by looking at their behaviour in computations. If $\sim$ were such an equivalence then the quotient $\mathcal{M}/\sim$ would contain the essence of the behaviours exhibited in $\mathcal{M}$. It would be even more convenient if the resulting quotient naturally formed a structure, that is, if $\sim$ were a congruence with respect to the functions and relations of $L$.

However, the following observation tells us that no such congruence can exist.

**Proposition 3.4.1.** No non-trivial equivalence on any structure is a congruence with respect to equality.

**Proof.** Suppose that $\sim$ is a congruence and $a \sim b$. Then the relation $(a = b)$ will be congruent (i.e. equivalent) to $(a = a)$. Then $a = b$, so $\sim$ is trivial.

This fact tells us that even if, for example, $a$ and $b$ are indistinguishable, we should not expect the pairs $(a, a)$ and $(a, b)$ to be indistinguishable. Thus in order to obtain complete information about the different behaviours exhibited in a structure we must consider the distinguishability of *tuples* in addition to just elements. This motivates the following definitions.
We fix on a class of programs $C$ over a many sorted language $L$, closed under basic constructs.

**Definition 3.4.1.** Given a structure $\mathcal{M}$ over a language $L$, we define the set $\bar{M}$ to be the set of all sorted tuples of elements of $M$, i.e.

$$\bar{M} = \bigcup \{ (k; i_k, \ldots, i_k; m_i, \ldots, i_m \in Sort, m_j \in M_{i_1}, \ldots, m_k \in M_{i_k}) \}.$$ 

That is, $\bar{M}$ is the set of things we are going to be trying to distinguish.

We then define an equivalence relation $\sim$ on $\bar{M}$ as follows.

**Definition 3.4.2.** Let $\mathcal{M}$ be an $L$-structure and for some argument type $\tau = (k; i_1, \ldots, i_k)$, let $m_1$ and $m_2$ be elements of $M_{i_1} \times \cdots \times M_{i_k}$. We say $m_1$ and $m_2$ are indistinguishable and write $(\tau; m_1) \sim (\tau; m_2)$ if for each $\tau$-ary scheme $P$ from $C$,

$$m_1 \in \text{domain}(P^M) \Leftrightarrow m_2 \in \text{domain}(P^M).$$

The quotient $\bar{M}/\sim$ will be denoted $M^+$.

**Definition 3.4.3.** A structure $\mathcal{M}$ is said to be distinguishable if $\sim$ is trivial.

Then $M^+$ contains all the information concerning which 'types' of elements (and tuples of elements) are present in the structure $\mathcal{M}$.

We can try to think about comparing the sets $M^+$ and $N^+$ for two $L$-structures $\mathcal{M}$ and $\mathcal{N}$.

Before we do that, it simplifies matters to observe that the exact nature of the definitions in $C$ is not important, provided they are at least as powerful as loop free schemes.

**Proposition 3.4.2.** Let $C$ be a class of definable functions containing all loop free schemes. For any structure $\mathcal{M}$, the sets $M^+$ for $C$ and $M^+$ for loop free are the same.

**Proof.** It suffices to show that if two $\tau$-ary tuples are distinguishable using functions from $C$, they are distinguishable using loop free programs. Suppose $m_1$ and $m_2$ are $\tau$-ary tuples with $m_1 \in \text{domain}(P^M)$ but $m_2 \notin \text{domain}(P^M)$ for some scheme $P$ in $C$. Since $P$ is defined for
m_1$, there is a clause $E_1(x) \land ... \land E_k(x) \rightarrow t(x)$ of $P$ for which $E_1^m(m_1) \land ... \land E_k^m(m_1) \equiv$ true. Let $Q$ be the scheme consisting of just that clause, i.e.

$$Q = \{E_1(x) \land ... \land E_k(x) \rightarrow t(x)\};$$

then $m_1 \in \text{domain}(Q^m)$ but $m_2 \notin \text{domain}(Q^m)$. •

We also observe that given a scheme $P$ of argument sort $\tau$ over $L$, we can define $P^{M^+}$ to be the set of equivalences classes $(\tau; [m])$ for which $P^m(m)$ is defined, i.e.

$$P^{M^+} = \{ (\tau; [m]) \in M^+ \mid m \in \text{domain}(P^m) \}$$

Now for the equivalence.

**Definition 3.4.4.** Let $L$ be a language and $\mathcal{M}$ and $\mathcal{N}$ two $L$-structures. We say $\mathcal{M}$ and $\mathcal{N}$ are distinguishability equivalent and write $\mathcal{M} \equiv_{dt} \mathcal{N}$ if there is a bijection $f: M^+ \rightarrow N^+$ that respects the definedness of schemes, i.e.

for each $\tau$-ary loop free scheme $P$ and $(\tau; [m]) \in M^+$, $(\tau; [m]) \in P^{M^+} \Leftrightarrow (\tau; [f(m)]) \in P^{N^+}$.

In other words, for each tuple in $\mathcal{M}$, there is a tuple in $\mathcal{N}$ with the same definedness properties for every scheme and vice versa.

We can see immediately that this equivalence guarantees every te equivalence.

**Proposition 3.4.3.** Suppose that for two structures $\mathcal{M}$ and $\mathcal{N}$, $\mathcal{M} \equiv_{dt} \mathcal{N}$. Then $\mathcal{M} \equiv_{teG_0} \mathcal{N}$.

**Proof.** Suppose that for some $\tau$-ary scheme $P$, $P^m$ is not total. Then there is a tuple $m \in \mathcal{M}$ with $m \notin \text{domain}(P^m)$. Then any $(\tau; n) \in f((\tau; [m]))$ has the property that $n \notin \text{domain}(P^m)$. •

The reverse implication is not true. Proof is this is left to §5.

In its current form, it is not easy to fit this equivalence into the general framework of observations, each of which is a sentence in infinitary logic. We are going to see how this can be done, however. The idea is that the elements of $M^+$ are the sets of tuples with certain properties; and these properties can be expressed as atomic formulas or their negations. As far
as distinguishability is concerned, and therefore as far as termination equivalence is concerned, a tuple in a structure is nothing more than the set of atomic formulas it satisfies. We shall formalise this in the notion of type. ('Type' is used in a huge variety of contexts and means different things. We are stealing it in this instance from the notion of types in model theory — more about the relationship between this and that notion will be found in §4.4.)

**Definition 3.4.5.** Let \( \mathcal{M} \) be a structure over a language \( L \). Given a \( \tau \)-ary tuple \( \mathbf{m} \) for some argument type \( \tau \), the type of \( \mathbf{m} \), denoted \( \Phi_L(\mathbf{m}) \) (or, more simply, \( \Phi(\mathbf{m}) \) where no confusion arises) is the set of all \( \tau \)-ary atomic formulae \( \psi(x) \) for which \( \psi^\mathcal{M}(\mathbf{m}) \equiv \text{true} \).

The set of observations we are interested in is the following.

**Definition 3.4.6.** Given, for some sort type \( \tau \), a set \( \Phi \) of \( \tau \)-ary atomic or negated atomic formulae, the type-observation of \( \Phi \) is the observation \( \gamma_\Phi = \exists x . \bigwedge_{\phi \in \Phi} \phi(x) \). The set \( \Gamma_0 \) is the set of all such type-observations. The equivalence induced by \( \Gamma_0 \) (written \( =_{\Gamma_0} \)) will be referred to as type-equivalence.

Now we can show how \( dt \)-equivalence corresponds to \( \Gamma_0 \)-equivalence.

**Proposition 3.4.4.** \( dt \)-equivalence and \( \Gamma_0 \)-equivalence are the same.

**Proof.** Suppose \( \mathcal{M} \) and \( \mathcal{N} \) are structures of a language \( L \). We show \( \mathcal{M} \equiv_{dt} \mathcal{N} \iff \mathcal{M} \equiv_{\Gamma_0} \mathcal{N} \).

\( \Rightarrow \): Suppose \( \mathcal{M} \equiv_{\Gamma_0} \mathcal{N} \). Then there exists a \( \tau \)-ary tuple \( \mathbf{m} \in \mathcal{M} \) such that for every \( \mathbf{n} \in \mathcal{N} \), there is an atomic or negated formula \( \phi(x) \) with \( \Phi^\mathcal{M}(\mathbf{m}) \equiv \neg \Phi^\mathcal{N}(\mathbf{n}) \). Thus there is no \( (\tau, [\mathbf{n}]) \in N^+ \) with the property that for each loop free scheme \( P \), \( (\tau; [\mathbf{m}]) \in P^M^+ \iff (\tau, [\mathbf{n}]) \in P^{N^+} \).

So \( \mathcal{M} \equiv_{dt} \mathcal{N} \).

\( \Leftarrow \): Suppose \( \mathcal{M} \equiv_{dt} \mathcal{N} \). The there exists a \( (\tau, [\mathbf{m}]) \in M^+ \) such that there is no \( (\tau, [\mathbf{n}]) \in N^+ \) with the property that for each \( P \), \( (\tau, [\mathbf{m}]) \in P^M^+ \iff (\tau, [\mathbf{n}]) \in P^{N^+} \). Then there exists a tuple...
If \( m \in \mathcal{M} \) such that for every \( n \in \mathcal{N} \), there is a quantifier free formula \( \varphi(x) \) with \( \varphi(m) \equiv \neg \varphi(n) \). So \( \mathcal{M} \not\equiv_{\Gamma_0} \mathcal{N} \). •

This equivalence and the notion of types in general will become very important in the next chapter.

It might be fair to say that on a structure \( \mathcal{M} \), the set \( M^+ \) captures everything about the structure we are interested in – it says everything about the properties of the elements of \( \mathcal{M} \) which can be of interest to a computation. We will see how \( dt \)-equivalence is in fact the same as \( te_{\mathcal{C}_0} \)-equivalence and is therefore the strongest \( te \) equivalence we will be considering. We can see that it is different to isomorphism from the following example.

**Proposition 3.4.5.** There exist non-isomorphic structures \( \mathcal{M} \) and \( \mathcal{N} \) of a language \( L \) which are \( \Gamma_0 \)-equivalent.

**Proof.** Let \( L \) be a language with one sort and no operation symbols. Let \( \mathcal{M} \) be a countably infinite structure of \( L \), and \( \mathcal{N} \) an uncountable structure of \( L \). Then clearly \( \mathcal{M} \) and \( \mathcal{N} \) are not isomorphic. But \( \mathcal{M} \) and \( \mathcal{N} \) are \( \Gamma_0 \)-equivalent. For, let \( \tau \) be a sort type and \( \Phi \) a set of quantifier free formulae over \( \tau \). The only relations in the formulae are equalities; given a finite tuple from one structure, there will always be a tuple in the other structure in which the same equalities and inequalities hold. •

There are examples of structures of the same cardinality, both countable and uncountable, which are \( \Gamma_0 \)-equivalent but not isomorphic, but these are left to §5.1.

Now we look at the relationship between \( dt \)-equivalence and \( te \)-equivalences.

**Proposition 3.4.6.** The sets \( \Gamma_0 \) and \( te_{\mathcal{C}_0} \) of observations are similar.

**Proof.** We show that for any two \( L \)-structures \( \mathcal{M} \) and \( \mathcal{N} \), \( \mathcal{M} \equiv_{\Gamma_0} \mathcal{N} \iff \mathcal{M} \equiv_{te_{\mathcal{C}_0}} \mathcal{N} \).
Suppose $\mathcal{M} \equiv_{\Gamma_0} \mathfrak{N}$ and that for a $G_0$ scheme $P = \{ E_{1p}(x) \land \ldots \land E_{kp}(x) \rightarrow t_p(x) \mid p \in P \}$, $P^{\mathcal{M}}$ is not total, undefined at $m \in \mathcal{M}$, say. Then for every clause $E_{1p}(x) \land \ldots \land E_{kp}(x) \rightarrow t_p(x)$ of $P$, there is a conjunct $E_{ip}(x)$ for which $E_{ip}^{\mathcal{M}}(m) = \text{false}$. Let $\gamma$ be the $\Gamma_0$ observation $\gamma_\Phi$, where

$$\Phi(x) = \{ -E_{ip}(x) \mid p \in P \}.$$ 

Since $\mathcal{M} \models \gamma_\Phi$ we have $\mathfrak{N} \models \gamma_\Phi$ by $\Gamma_0$-equivalence; suppose that $n \in \mathfrak{N}$ satisfies $\Phi$. Then $P^{\mathfrak{N}}$ is undefined at $n$ and so is non-total. Thus $\mathcal{M} \equiv_{\text{le}_{\Gamma_0}} \mathfrak{N}$.

$\Leftarrow$: Suppose that $\mathcal{M} \equiv_{\text{le}_{\Gamma_0}} \mathfrak{N}$ and that $\mathcal{M} \models \gamma_\Phi$ for a type $\Phi(x) = \{ \phi_1(x), \ldots, \phi_i(x), \ldots \}$.

Let $P$ be the scheme $P = \{ -\phi_i(x) \rightarrow \text{true} \mid \phi_i(x) \in \Phi(x) \}$.

Then $P^{\mathcal{M}}$ is non-total and so $P^{\mathfrak{N}}$ is non-total by $\text{le}_{\Gamma_0}$-equivalence; thus we have $\mathfrak{N} \models \gamma_\Phi$ and so $\mathcal{M} \equiv_{\Gamma_0} \mathfrak{N}.$

We finish with an algebraic characterisation of $\Gamma_0$-equivalence.

Proposition 3.4.7. Let $\mathcal{M}$ and $\mathfrak{N}$ be structures of a language $L$. Then the following are equivalent.

1. $\mathcal{M} \equiv_{\Gamma_0} \mathfrak{N}$;
2. every finitely generated substructure of $\mathcal{M}$ is isomorphic to a substructure of $\mathfrak{N}$ and vice versa.

Proof. Let $n$ be a $\tau$-ary tuple in $\mathfrak{N}$. Since the types satisfied by $n$ are determined entirely by $\langle n \rangle$, the implication $2 \Rightarrow 1$ is clear.

In the other direction, suppose $\mathcal{M} \equiv_{\Gamma_0} \mathfrak{N}$ and that $m$ is a $\tau$-ary tuple in $\mathcal{M}$. We want to find a $\tau$-ary tuple $n \in \mathfrak{N}$ such that $\langle n \rangle \not= \langle m \rangle$. 

Chapter 3 – Algorithmic equivalences – Distinguishability
Let $\Phi(x) = \{ \phi_1(x), \phi_2(x), \ldots, \phi_p(x), \ldots \}$ be the collection of all atomic and negated atomic relations $\phi_p(x)$ with $\phi_p^M(m) = \text{true}$. Then $\Phi(x)$ is a type associated with $\Gamma_0$. Since $M$ satisfies $\Phi$ at $m$, there exists a point $n$, say, in $M$ with $n$ satisfying $\Phi$. But then $\langle n \rangle \neq \langle m \rangle$. •

We will be concerned with $\Gamma_0$-equivalence and other related equivalences in the next chapter.
3.5 Specification equivalence

The mathematical theory of program correctness concerns itself with proof systems for proving assertions of the form \( \{p\} S \{q\} \),

where \( S \) is a program in some programming language of interest, \( p \) is a formula holding before the execution of the program, and \( q \) is a formula intended to hold after termination of the program. We can think of the formula \( \{p\} S \{q\} \) as a specification of the program \( S \). These proof systems have been studied extensively, both in the standard case (such as in [de Bakker, 1980]) and in the presence of abstract data types (such as in [Tucker and Zucker, 1989]).

The proof systems required to manipulate these formulae are based on the constructs available in the programming language; there will, for example, be proof rules to handle each of the constructs for assignment, sequential composition, iteration, and so on. The nature of the formulae \( p \) and \( q \) also depend on the nature of the programming language. The presence or otherwise of abstract types will in particular influence the kinds of formulae appropriate.

Our aim in this section is to compare algebraic structures by the program specifications satisfied by them; in order to do this we need to carefully state exactly which assertions we wish to consider and how we decide whether or not a particular assertion is true on a given structure. Our aim is not, however, to study in detail the 'internal' concerns, such as the nature of the programming languages, and details of the proof systems. In particular, we do not wish to delve in to the details of how an assertion language for a machine with a stack would differ from that for a (computationally equivalent) recursive programming language. Certain aspects of the differences between programs whose 'correctness theories' (the set of input/output specifications they satisfy) are the same are discussed in [Bergstra, Tiuryn and Tucker, 1982]; we do not discuss these here. (The nature of the assertions we present and the assertions in that paper are slightly different; Lemma 2.2 fails to hold for our definition of partial correctness theories.) Therefore, we are going to limit the nature of the assertions we can make in order that their truth is independent of the particular choice of computational model. We define the set of assertions and their truth in the following way.
There are two types of formulae expressing properties of the state we are interested in: the case where we allow only quantifier-free formulae, and the case where we allow all first order formulae.

There are also two interpretations of the meaning of the triple \( \{ p \} S \{ q \} \):

1. the *partial correctness* interpretation, in which \( \{ p \} S \{ q \} \) is true when for each initial state satisfying \( p \), and whenever \( S \) terminates, the final state satisfies \( q \);
2. the *total correctness* interpretation, which in addition asserts that \( S \) does in fact terminate for each initial state satisfying \( p \).

Therefore, the specification equivalences we discuss will be divided according to the programming formalism, the expressive power of the formulae and whether we are interested in partial correctness or total correctness.

**Definition 3.5.1.** Let \( L \) be a many sorted language and \( C \) a class of programs over \( L \). A *specification* over \( C \) is a triple \( \{ p \} S \{ q \} \), where

1. \( S \) is a program from \( C \) with sort type \( \tau = (m; i_1, \ldots, i_m, i) \);
2. \( p \) is a first order formula over the language \( L \) with free variables \( x_1, \ldots, x_m \)
   of sorts \( i_1, \ldots, i_m \) respectively;
3. \( q \) is a first order formula over \( L \) with free variables \( x_1, \ldots, x_m \) of sorts \( i_1, \ldots, i_m \) respectively and a free variable \( x \) of sort \( i \).

An \( L \)-structure \( \mathcal{M} \) *partially satisfies* the formula \( \{ p \} S \{ q \} \) if

\[
\mathcal{M} \models \forall x_1, \ldots, x_m, y. (p(x) \land \text{defined}(S, x, y) \Rightarrow q(x, y)).
\]

An \( L \)-structure \( \mathcal{M} \) *totally satisfies* the formula \( \{ p \} S \{ q \} \) if

\[
\mathcal{M} \models \forall x_1, \ldots, x_m. p(x) \Rightarrow (\exists y. \text{defined}(S, x, y) \land q(x, y)).
\]
Two L-structures \( \mathcal{M} \) and \( \mathcal{N} \) are said to be \textit{partially specification equivalent for} \( C \) if for every specification \( \{ p \} S \{ q \} \), \( \mathcal{M} \) partially satisfies \( \{ p \} S \{ q \} \) if and only if \( \mathcal{N} \) partially satisfies \( \{ p \} S \{ q \} \).

Similarly, two structures \( \mathcal{M} \) and \( \mathcal{N} \) are said to be \textit{totally specification equivalent for} \( C \) if for every specification \( \{ p \} S \{ q \} \), \( \mathcal{M} \) totally satisfies \( \{ p \} S \{ q \} \) if and only if \( \mathcal{N} \) totally satisfies \( \{ p \} S \{ q \} \).

We can use specification equivalence to compare structures in a manner similar to termination equivalence. The case where we only allow quantifier-free formulae is easily dealt with.

**Proposition 3.5.1.** Let \( C \) be a class of definable functions over a language \( L \) closed under basic constructs. Then

1. total specification equivalence for \( C \) is identical to termination equivalence for \( C \);
2. partial specification equivalence for \( C \) is identical to termination equivalence for loop free.

**Proof.** (1) Suppose \( \mathcal{M} \) and \( \mathcal{N} \) are L-structures and that \( \mathcal{M} \) and \( \mathcal{N} \) are totally specification equivalent. Then for any \( P \in C \), \( P^{\mathcal{M}} \) is total iff \( \mathcal{M} \) totally satisfies the formula \( \{ \text{true} \} P \{ \text{true} \} \).

So total specification equivalence implies \( t_{\text{ec}} \)-equivalence. Now suppose that \( \mathcal{M} \equiv_{t_{\text{ec}}} \mathcal{N} \) and that \( \mathcal{M} \) totally satisfies the formula \( \{ p \} S \{ q \} \) for quantifier-free formulae \( p \) and \( q \), and for some \( \tau \)-ary scheme \( S \in C \).

Writing \( p = \bigvee_i p_i \) and \( q = \bigvee_j q_j \) in disjunctive normal form, let \( P \) and \( Q \) be the schemes

\[
P = \{ p_i \rightarrow \text{true} \}_i \quad \text{and} \quad Q = \{ q_j \rightarrow \text{true} \}_j.
\]

Let \( S' \) be the \( \tau \)-ary scheme \( S' = \bigwedge(P, S); Q \). The formula \( \{ p \} S \{ q \} \) will be satisfied on a structure precisely when the programs \( P \) and \( S' \) have the same domain on that structure. Then by 3.2.3 we have that \( \mathcal{N} \) totally satisfies \( \{ p \} S \{ q \} \).
(2) Suppose $\mathcal{M}$ and $\mathcal{N}$ are $L$-structures and that $\mathcal{M}$ and $\mathcal{N}$ are partially specification equivalent. Then for any $P \in C$, $P^\mathcal{M}$ is defined nowhere iff $\mathcal{M}$ satisfies the formula $\{\text{true}\} P \{\text{false}\}$. So partial specification equivalence implies $te_{\text{loop free}}$-equivalence.

Now suppose that $\mathcal{M} \equiv_{\text{loop free}} \mathcal{N}$ and that $\mathcal{M}$ partially satisfies the formula $\{p\} S \{q\}$ for quantifier-free formulae $p$ and $q$, and for some $\tau$-ary scheme $S \in C$.

Writing $p \equiv \bigvee_i p_i$ and $\neg q \equiv \bigvee_j q_j$ in disjunctive normal form, let $P$ and $Q$ be the schemes

$$P = \{ p_i \rightarrow \text{true} \}_{i} \quad \text{and} \quad Q = \{ q_j \rightarrow \text{true} \}_{j}.$$ 

Let $S'$ be the $\tau$-ary scheme $S' = \bigwedge (P, S); Q$. The formula $\{p\} S \{q\}$ will be satisfied on a structure precisely when the scheme $S'$ is defined nowhere on that structure. Then by 3.2.4 we have that $\mathcal{N}$ partially satisfies $\{p\} S \{q\}$. •

So now we can concentrate on the case where we allow any first order formulae to appear for the $p$ and $q$. Partial specification equivalence in this form has been studied in [Bergstra and Tucker, 1983] and the following identity is shown.

**Proposition 3.5.2.** Partial specification equivalence is identical to elementary equivalence. •

Our concern therefore will be with total specification equivalence. We have the following results immediately from 3.5.1.

**Proposition 3.5.3.** Let $L$ be a language and $C$ a class of definable functions closed under basic constructs. Then total specification equivalence for $C$ implies $te_C$-equivalence. •

So the specification equivalences relate to the termination properties of a structure; now we see that they are also related to logical equivalence.

**Proposition 3.5.4.** Total specification equivalence implies elementary equivalence.
Proof. Let L be a language and \( \phi \) a sentence of L. Let P be any basic scheme. An L-structure \( \mathcal{M} \) will satisfy \( \phi \) iff it totally satisfies the formula \( \{\phi\} P \) (true). Therefore specification equivalent structures will be elementarily equivalent.

From 3.5.2 we have:

**Proposition 3.5.5.** Total specification equivalence implies partial specification equivalence.

In §5.1 we will see that termination equivalence does not imply elementary equivalence; this will give us that neither total nor partial specification equivalence are implied by termination equivalence.

Since total specification equivalence for a class C of definable functions is the only equivalence we have not already got a name for (in the light of 3.5.1 and 3.5.2), we will write this as \( \text{sp}_C \)-equivalence.

It is an open problem as to whether or not termination equivalence and elementary equivalence imply total specification equivalence.

We can use structures with the unwind property to distinguish between different \( \text{sp}_C \)-equivalences; we will show how this is done in §5.1.7.
Chapter Four  Types and type based equivalences

4.1 Types

At the end of the last chapter, we introduced the notion of the type of an element and an equivalence between structures which depended on there being the same types of (tuples of) elements present. This gave rise to the strongest te-equivalence, that for the class \( G_0 \) of all definable functions. In this section, we are going to work toward weakening this definition by allowing only certain kinds of type observations, and finding type-based equivalences for the other te equivalences.

The idea is that each of the algorithmic equivalences we have looked at can be thought of as the correspondence of elements with particular properties; in the case of te equivalence, for example, we look for elements in domains of structures for which a particular program does not terminate, and in the case of nd-equivalence, elements for which it does terminate.

We defined a type observation to be a set of atomic relations or their negations over some fixed sort type \( \tau \). In this section, we show that if we are to come up with a set of type observations for te equivalence, for example, we cannot get away with just considering atomic relations, but must allow a type to be a set of quantifier-free formulae.

First, the necessary definitions.

**Definition 4.1.1.** A type over a sort type \( \tau = (m; i_1, \ldots, i_m) \) is a set of relations

\[
\Phi = \{ \phi_1, \ldots, \phi_k, \ldots \}
\]

where each \( \phi_i \) is of a certain form (see below) and over the tuple \( \mathbf{x} = (x_1, \ldots, x_m) \).
The type observation $\gamma_\Phi$ corresponding to $\Phi$ is the observation $\gamma_\Phi = \exists x \cdot \bigwedge_{\phi \in \Phi} \phi(x)$. We can clearly generate an equivalence given a set $\Xi$ of types; this equivalence is called the type equivalence for $\Xi$.

We consider two specific forms for the relations $\phi_i$, calling them (a) and (b).

(a) allows $\phi_i$ to be any quantifier-free formula over the variables $x$;
(b) restricts each $\phi_i$ to be either an atomic relation over the variables $x$ or its negation.

The most important difference between forms a and b is the presence of disjunctions in relations of form a which cannot appear in those of form b.

We can see that form a is adequate to generate an equivalence similar to $te_C$ for any class $C \subseteq G_0$.

Proposition 4.1.1. Let $L$ be a language and $C$ a class of definable functions over $L$. Then $te_C$ is similar to a set of types of form a.

Proof. Given a scheme $S = \{ E_1^s(x) \land \ldots \land E_k^s(x) \rightarrow t_s(x) \mid s \in S \}$, let the observation $\gamma_S$ be the type observation $\gamma_{\Phi_S}$, where

$$\Phi_S(x) = \{ \neg(E_1^s(x) \land \ldots \land E_k^s(x)) \mid s \in S \}. $$

Then for a class $C$ of definable functions, let $\Xi_C$ be the set

$$\Xi_C = \{ \gamma_{\Phi_S} \mid S \in C \};$$

clearly then

$$\Xi_C = \{ \neg \gamma \mid \gamma \in te_C \};$$

so $te_C$ and $\Xi_C$ are similar by 3.1.1. •
However, in this section we shall show that the conclusion of the above proposition is false if we restrict allowable types to be of form $b$ only, for any class of functions at least as rich as $f_{\text{ap}}$.

The proof is a counting argument. We give a flow chart definable function for which there are uncountably many different execution paths corresponding to non termination. There can, of course, only be countably many terminating paths. We show that for each non terminating path there exists a structure on which that path is followed. In addition, for every initial segment of a non terminating path, there exists a structure on which the function terminates. We use these two facts to show that a set of type observations of form $b$ must be uncountable if it is to distinguish at least as many structures as $te_{f\text{ap}}$. But then by counting equivalence classes, we see that any set of type observations which is strong enough must in fact be too strong.

**Proposition 4.1.2.** There is a $f\text{ap}$ $f_0$ such that if $C$ is a class of definable functions containing $f_0$ and there is a set of types of form $b$ generating a set of observations similar to $te_C$, then $C$ is uncountable.

![Flow Chart](image.png)

**Figure 4.1.1.** The program $f_0$. 
Proof. Let \( L \) be the language with one sort containing two unary function symbols \( f \) and \( g \) say, and two unary relation symbols \( \theta \) and \( \phi \) say. Let \( f_0 \) be the program shown in Fig. 4.1.1.

The uncountably many paths correspond to the infinite sequences of ways the program can happen to execute the body of the loop; if we think of every branch in the execution of the program as a node in a binary tree, the set of possible paths through the program might be visualised in the tree shown in Fig. 4.1.2.

![Figure 4.1.2. The paths through \( f_0 \).](image)

Suppose the evaluation of this function does not terminate on some structure. The route through the program follows an infinite path on this expanded tree. At every alternate node, the program is given an opportunity to stop, corresponding to the test of \( \theta \) in the flow chart. At the other alternate set of nodes however, the program can branch, and still follow a possibly non
terminating path. It is the infinite sequence of decisions at these nodes which give the uncountable set of paths.

We prove the result in two parts:

1. Any set of types $\Xi$ whose equivalence is at least as strong as $\text{tec}$ must contain uncountably many types;
2. If $\Xi$ is similar to $\text{tec}$, then the cardinality $2^\kappa \leq 2^C$ for some uncountable $\kappa$, and hence $C$ is uncountable.

1. Let $S$ be the set of all countably infinite sequences in which every element is either an $f$ or a $g$. We know that $|S| = 2^{\aleph_0}$. Given a sequence $a = (a_1, a_2, \ldots, a_i, \ldots)$ we say that the $i^{\text{th}}$ tail of $a$ is the sequence $(a_i, a_{i+1}, \ldots)$. Now we define an equivalence $\sim$ on $S$: $a \sim b$ if there exist $i, j$ such that the $i^{\text{th}}$ tail of $a$ is the same as the $j^{\text{th}}$ tail of $b$; these ideas are illustrated in Fig. 4.1.3.

![Figure 4.1.3. Infinite sequences of symbols.](image)

We can see that for any $a$, $[a]_\sim$ is countable since for a specific $i$ and $j$, the set of possible $b$ is countable. Thus there are uncountably many equivalence classes in $S/\sim$; let $\kappa$ be the cardinal of $S/\sim$.

Now we are going to define some structures. We define a structure $M_a$ for every finite and for every infinite sequence $a$ of $f$s and $g$s.

The carrier set of each is the same; the set $M$ of all finite sequences of $f$s and $g$s. The interpretations of $f$ and $g$ are as one might expect;
for every finite sequence s,  
\[ f^M_a(s) = s \wedge (f) \]
and  
\[ g^M_a(s) = s \wedge (g). \]

Let \( a \) be a specific sequence. The structure \( M_a \) is defined by the interpretations of \( \phi \) and \( \theta \) on \( M \). We put

\[ \phi^M_a(s) = \text{true} \text{ iff } s \text{ and } s \wedge (f) \text{ are both initial segments of } a; \]
\[ \theta^M_a(s) = \text{true} \text{ iff } s \text{ is a strictly initial segment of } a \text{ (i.e. is an initial segment of } a \text{ but is not the same as } a). \]

Thus for a finite sequence \( a \), the structure \( M_a \) might look like that shown in Fig. 4.1.4.

\[ \begin{align*}
\text{empty sequence, } \lambda \\
\text{the sequence } a \\
\end{align*} \]

\[ \begin{align*}
\bullet & \quad \phi \text{ and } \theta \text{ false} \\
\circ & \quad \phi \text{ false and } \theta \text{ true} \\
\circ & \quad \phi \text{ and } \theta \text{ true} \\
\end{align*} \]

**Figure 4.1.4.** The structure \( M_a \).
If $a$ is infinite, the picture looks much the same.

The following facts are now clear from the construction of $\mathcal{M}_a$.

1. If $a$ is finite, $f_0^0M$ is total. $f_0^0M(\lambda)$ terminates after $\|a\|$ times round the loop.
2. If $a$ is infinite, $f_0^0M$ is undefined at each initial segment of $a$. Elsewhere, $f_0^0M$ is defined.
3. If $a \neq a'$, then the structures $\mathcal{M}_a$ and $\mathcal{M}_{a'}$ are not isomorphic.

Now let $a$ be infinite. We consider the structure $\mathcal{N}_a$ which is the disjoint union

$$\mathcal{N}_a = \bigcup \{ \mathcal{M}_s \mid s \text{ is an initial segment of } a \}.$$

Let $\mathcal{R}_a$ be the disjoint union

$$\mathcal{R}_a = \mathcal{M}_a \cup \bigcup \{ \mathcal{M}_s \mid s \text{ is an initial segment of } a \}.$$

We note that:

1. $\mathcal{R}_a$ and $\mathcal{N}_a$ are not $te$-equivalent for any class of programs containing $f_0$;
2. they are basically (or $te_{\text{loop-free}}$) equivalent.

To see (2), suppose that for some sort type $\tau$, there is a $\tau$-ary quantifier-free formula $\phi(x)$, and a $\tau$-tuple $r \in \mathcal{R}_a$ with $\mathcal{M}_a \models \phi(r)$. Since there are only finitely terms occurring in $\phi$, the satisfaction of $\phi$ at an interpretation $r$ will depend upon the interpretation of $\theta$ and $\phi$ at only finitely many points in $\langle r \rangle$; so $\phi$ will be satisfiable in $\mathcal{N}_a$ by choosing elements from $\mathcal{M}_s$ instead of $\mathcal{M}_a$ for a sufficiently large initial segment $s$ of $a$.

Suppose $\Xi$ is a set of type observations of form $b$, $C$ is a class of definable functions containing $f_0$, and that $\Xi$-equivalence is at least as strong as $te_C$. Then we must have $\mathcal{R}_a \models_{\Xi} \mathcal{N}_a$. Since $\mathcal{R}_a$ and $\mathcal{N}_a$ are basically equivalent, they agree on every finite form $b$ type, so there must be an infinite form $b$ type, $\Phi$ say, over variables $x = (x_1, \ldots, x_m)$ such that $\mathcal{M}_a \not\models \gamma_\Phi$ but $\mathcal{R}_a \models \gamma_\Phi$ for some interpretation $r \in \mathcal{R}_a$ of the variables $(x_1, \ldots, x_m)$.
Consider the form of the formulae in $\Phi$. They might be

1. equalities $x_i = x_j$ or inequalities $x_i \neq x_j$;
2. $\theta(t(x_i))$ or its negation $-\theta(t(x_i))$ for some $x_i$ and term $t$ in $x_i$;
3. $\phi(t(x_i))$ or its negation $-\phi(t(x_i))$ for some $x_i$ and term $t$ in $x_i$.

Since $\Phi$ is not satisfiable in $\mathfrak{N}_a$, there must be a variable $x \in \{x_1, \ldots, x_m\}$ (which we will assume is $x_1$ w.l.o.g.) such that

1. its interpretation $r$, say, in $r$ is such that $\langle r \rangle$ (the substructure generated by $r$) is different to every $\langle n \rangle < \mathfrak{N}_a$; $r$ is therefore an initial segment of $a$ in $\mathfrak{M}$;
2. there are sufficiently many atomic relations over $x_1$ in $\Phi$ to enable $\Phi$ to distinguish $\langle r \rangle$ from every other $\langle n \rangle$; therefore there is an infinite set $\{t_1, \ldots, t_i, \ldots\}$ of terms in $x_1$ for which either $\Phi$ includes $\theta(t_i(x))$ or infinitely many $\phi(t_i(x))$ or both;
3. the elements $t_1 R_a(r), \ldots, t_i R_a(r), \ldots$ are all initial segments of $a$ in $\mathfrak{M}_a$ in $\mathfrak{N}_a$;
4. the terms $t_1, \ldots, t_i, \ldots$ define a unique infinite sequence of $fs$ and $gs$ $b$ say, which is a tail of $a$;
5. the set $\Phi$ of atomic and negated atomic relations define a unique element $[b]_\sim$ of $S/\~$. 

Hence no $\Phi$ in $\Xi$ can distinguish $\mathfrak{N}_a$ from $\mathfrak{N}_a$ for more than one $[b]_\sim \in S/\sim$. If $\Phi$ distinguishes $\mathfrak{N}_a$ and $\mathfrak{N}_a$, then $\Phi$ is supported on no $\mathfrak{N}_b$ with $b \not\in [a]_\sim$. Hence there must be at least $\kappa$ observations in $\Xi$.

It is clear that the maximum number of equivalence classes of structures under an observational equivalence is $2^\psi$ where $\psi$ is the number of observations, since each equivalence class is defined by a subset of the set of observations.
In the case of $\Xi$ it is easy to see that there are at least $2^K$ equivalence classes. For $s \in S$, let $\Phi_s$ denote an observation in $\Xi$ which distinguishes $R_a$ and $N_a$ for some $a \in s$. For any subset $\Pi$ of $S/\sim$, let $R_\Pi$ be a disjoint union

$$R_\Pi = \bigcup \{ R_a \mid \text{some } a \in \pi \in \Pi \};$$

then for each $s \in S$, $R_\Pi \models \Phi_s$ iff $s \in \Pi$. So for distinct $\Pi$, $R_\Pi$ are in distinct $\Xi$-equivalence classes. Thus if $\Xi$ is similar to $\epsilon_C$, $2^K \leq 2^C$.

It is interesting to note that this result does not hold if we restrict attention to computable structures only. In fact we have the following results.

**Proposition 4.1.3.** Let $L$ be a language, and $\mathcal{K}$ the set of all computable structures of $L$. Then

1. Let $\Xi$ be the set of all type observations $\gamma_\Phi$ for which $\Phi$ is a recursively enumerable type over the language $L$ of form $b$. The sets of observations $\Xi$ and $te_{eds}$ are $\mathcal{K}$-similar.
2. Let $\Xi$ be the set of all type observations $\gamma_\Phi$ for which $\Phi$ is a recursively enumerable type over the language $L$ of form $b$ with the property that there is a uniform upper bound $n_\Phi$ on the number of registers needed to evaluate each of the relations in $\Phi$. The sets of observations $\Xi$ and $te_{fapC}$ are $\mathcal{K}$-similar.

**Proof.** We prove (1) only, as the proof for (2) is similar.

Let $\Phi$ be an r.e. set of formulae of type $b$ over some sort type $\tau$. Let $f_\Phi$ be the $eds$

$$f_\Phi = \{ \phi(x) \rightarrow \text{true } \mid \phi(x) \in \Phi(x) \};$$

$f_\Phi$ will fail to be defined at an interpretation $m$ in a structure iff $\phi^{m}(m) \equiv \text{true}$ for each $\phi \in \Phi$.

Thus the set of observations $te_{eds}$ includes the set $\{ \neg \xi \mid \xi \in \Xi \}$.  

Chapter 4 - Types and type-based equivalences - Types
Conversely, let $M$ and $N$ be computable $L$-structures with $M \equiv N$. Suppose that for some eds $f$ of argument type $\tau = (m; i_1, \ldots, i_m)$, $f$ fails to be defined at some $m = (m_1, \ldots, m_m)$ in $M$.

Let $\Phi$ be the set of all $\tau$-ary atomic relations and negations which are true on $m$. Since $M$ is effective, $\Phi$ is recursively enumerable by 2.7.7, and is therefore an element of $\Sigma$. Then there will be a tuple $n$ satisfying $\Phi$ on $N$, and $f^N(n)$ will not be defined.
4.2 Other type equivalences

In the previous section, we discussed the ways in which we could represent termination equivalence as an equivalence obtained from a set of type observations. We saw that, in contrast to $\Gamma_0$-equivalence, which is the same as $te_{G_0}$-equivalence, we could not represent termination equivalences obtained from program schemes using observations in form $b$.

In this section, we will introduce another few sets of type observations in form $b$, and these will then necessarily be different from the termination equivalences we have seen. The reasons for introducing them is that they have a special relationship with various algorithmic equivalences in that they are the congruences generated by the join operation. Precise statements and proofs of these relationships can be found in §5.

Definition 4.2.1. Let $L$ be a many-sorted language with possibly infinitely many operation and relation symbols. The set $\Gamma_f$ (for finite language) of observations over $L$ is a set of type observations; the types allowed are those $\Gamma_0$ types which are over a finite language, i.e.

$$\Gamma_f(L) = \bigcup \{ \Gamma_0(L_0) \mid L_0 \text{ is finite and } L \text{ extends } L_0 \}.$$ 

We define the set $\Gamma_{bs}$ (for bounded space) of type observations to be those in $\Gamma_f$ whose types have uniformly bounded pebble complexity; formally as the union

$$\Gamma_{bs}(L) = \bigcup_{n \in \omega} \Gamma_f^{(n)}(L),$$

where $\Gamma_f^{(n)} = \{ \Phi \in \Gamma_f \mid \text{every term in } \Phi \text{ has pebble complexity } \leq n \text{ over its free variables} \}$.

Finally, the set $\Gamma_{lf}(L)$ (for loop free) of type observations is the set of observations $\gamma_0$ for finite types $\Phi = \{ \phi_1, \ldots, \phi_p \}$ of form $b$.

We now define a few more classes of definable functions.

Definition 4.2.2. Let $L$ be a language. The set $G_f$ of definable functions is the union

$$G_f(L) = \bigcup \{ G_0(L_0) \mid L_0 \text{ is finite and } L \text{ extends } L_0 \}.$$
The set $G_{bs}$ of definable functions is the union

$$G_{bs}(L) = \bigcup_{n \in \omega} G_{fn}^{(n)}(L)$$

where

$$G_{fn}^{(n)}(L) = \{ P \in G_{fn}(L) \mid \text{every term appearing in a clause of } P \text{ has pebble complexity } \leq n \}.$$

These correspond to the type observations in the way you might expect.

**Proposition 4.2.1.** The following sets of observations are similar.

1. $te_{G_{fn}}$ and $\Gamma_{fn}$.
2. $te_{G_{bs}}$ and $\Gamma_{bs}$.
3. $te_{loop\ free}$ and $\Gamma_{lf}$.

**Proof.** In each case we will show that for any two $L$-structures $\mathcal{M}$ and $\mathcal{N}$,

$$\mathcal{M} \models_{\Gamma} \mathcal{N} \iff \mathcal{M} \equiv_{te_{\mathcal{G}}} \mathcal{N}.$$ 

$\Rightarrow$: Suppose that $\mathcal{M} \models_{\Gamma} \mathcal{N}$ and suppose $P$ is a definable function of the appropriate form.

Let $P$ be the scheme $\{ E_{1s}(x) \land E_{2s}(x) \land \ldots \land E_{ks}(x) \rightarrow t_s(x) \mid s \in S \}$ and suppose that $P^M$ is non total, undefined at $m$, say.

Then for each clause in the definition of $P$, there is a conjunct $E_{js}(x)$ with $E_{js}^M(m) = \text{false}$. Now the collection $\Phi = \{-E_{js}(x) \mid s \in S \}$ gives a type observation $\gamma_\Phi$ in the correct form; so we have $P^M$ non total and therefore $\mathcal{M} \equiv_{te_{\mathcal{G}}} \mathcal{N}$.

Conversely, suppose $\mathcal{M}$ and $\mathcal{N}$ are $te_{\mathcal{G}}$ equivalent and that $\mathcal{M} \models_{\Gamma} \gamma_\Phi$ for some $\Gamma$ type observation $\gamma_\Phi$ derived from type $\Phi$. Suppose $\Phi = \{ \phi_1, \ldots, \phi_p, \ldots \}$; now let $P$ be the scheme

$$P = \{ \neg \phi_p(x) \rightarrow \text{true} \mid \phi_p(x) \in \Phi \}.$$
P is of the correct form and $P^M$ is non total. Therefore, by $te$ equivalence, $P^N$ is non total, and this can only happen if $N \models \gamma$. Hence $M \models N$. 

This does slightly more than 4.1.1 which showed every $te$ equivalence can be expressed as a set of type observations in form a, whereas these are in form b.

Given a class C of definable functions we can associate a set of type observations of form $a$, as in 4.1.1. We also note that given a set $\Xi$ of types in form $b$ we can associate a class $C_\Xi$ of definable functions such that the sets $te_{C_\Xi}$ and $\{ \gamma_\Phi \mid \Phi \in \Xi \}$ of observations are similar. We do this by putting

$$C_\Xi = \{ P_\Phi \mid \Phi \in \Xi \},$$

where for a type

$$\Phi(x) = \{ \bigwedge_{j \in J_1} \phi_{j_1}(x), \ldots, \bigwedge_{j \in J_i} \phi_{j_i}(x), \ldots \}$$

of quantifier-free formulae in conjunctive normal form,

$P_\Phi$ is the scheme

$$P_\Phi = \{ -\phi_{ij}(x) \rightarrow \text{true} \mid i \in \{1, \ldots, j \in J_i \}. \}$$
4.3 The unwind property again

In §2.5, we looked briefly at a result which told us that if a program defines a total function on every model of a first order theory, then it is equivalent (on that structure) to a loop free program. This led us to make a definition of a property called the *unwind property*, which, if it is held by a structure $\mathcal{M}$, asserts that every program unwinds on $\mathcal{M}$. The unwind property has been studied extensively by authors such as Kfoury and Urzyczyn because it enables them to relate the termination of programs to basic equivalence, as the next fact shows.

**Proposition 4.3.1.** Let $L$ be a language and let $\mathcal{M}$ and $\mathcal{N}$ be $L$-structures. Suppose that $\mathcal{M} \equiv_b \mathcal{N}$. Then

1. if a scheme $P$ unwinds on $\mathcal{M}$ then it unwinds on $\mathcal{N}$;
2. if $P$ unwinds and is total on $\mathcal{M}$, $P$ is total on $\mathcal{N}$;
3. if $P$ unwinds and is non-total on $\mathcal{M}$, $P$ is non-total on $\mathcal{N}$;
4. if $\mathcal{M}$ has the unwind property for a class $C$ of definable functions then $\mathcal{M} \equiv_{teC} \mathcal{N}$ and $\mathcal{N}$ has the unwind property for $C$.

**Proof.** (1) Let $P$ be a scheme over $L$ from $C$, where

$$P = \{ E_{1s}(x) \land E_{2s}(x) \land \ldots \land E_{ks}(x) \rightarrow t_s(x) \mid s \in S \}.$$  

Since $P$ unwinds on $\mathcal{M}$, there exists a finite $S' \subseteq S$ such that if $Q$ is the scheme

$$Q = \{ E_{1s}(x) \land E_{2s}(x) \land \ldots \land E_{ks}(x) \rightarrow t_s(x) \mid s \in S' \},$$

then $P^\mathcal{M} = Q^\mathcal{M}$.

Now, for each $s \in S \setminus S'$, let $R_s$ be the scheme with one clause

$$R_s = \{ E_{1s}(x) \land E_{2s}(x) \land \ldots \land E_{ks}(x) \rightarrow t_s(x) \};$$

then each of the $R_s$ is defined nowhere in $\mathcal{M}$. Since basic equivalence is the same as $nd$-equivalence, we have that each of the $R_s$ is defined nowhere in $\mathcal{N}$, and so $P^\mathcal{N} = Q^\mathcal{N}$. Therefore $P$ unwinds in $\mathcal{N}$.
(2) and (3): Now extend $Q$ in the above example to a basic scheme $Q_B$, where
\[
Q_B = \{ E_{1s}(x) \land E_{2s}(x) \land \ldots \land E_{ks}(x) \rightarrow t_s(x) \mid s \in S' \}
\]
\[
\cup \{ F_{t_1}(x) \land \ldots \land F_{kt}(x) \rightarrow \text{true} \mid t \in T \}
\]
by choosing the conjuncts $F(x)$ so that
\[
\forall t \in T (F_{t_1}(x) \land \ldots \land F_{kt}(x)) \equiv \neg \forall s \in S' (E_{1s}(x) \land E_{2s}(x) \land \ldots \land E_{ks}(x)).
\]
Now for each $t \in T$, let $R_t$ be the scheme with the single clause
\[
R_t = \{ F_{t_1}(x) \land \ldots \land F_{kt}(x) \rightarrow \text{true} \};
\]
each $R_t$ is defined nowhere on $\mathcal{M}$ iff it is defined nowhere on $\mathcal{N}$ by $nd$-equivalence.

(4) By (1), $\mathcal{N}$ has the unwind property; by (2), if $P$ is a scheme from $C$, $P^\mathcal{N}$ is total iff $P^\mathcal{M}$ is total. •

This result, together with proofs of the existence of structures with the unwind property, has been used by [Urzyczyn, 1983] to prove a result in dynamic logic; the proof involves showing that $te_{\text{fap}}$ and $te_{\text{fapC}}$ give rise to different equivalences. We will present that proof shortly, but start with the example of the unwind property it depends upon.

The example gives us a structure which is not locally finite, but where there exist fapC functions different to every fap. This structure is called $T_\omega$ and the following definitions come from that paper.

The structure $T_\omega$ is first defined in the form of an infinite tree, which we later explain how to view as a unary algebra. As a tree, $T_\omega$ will be obtained by taking the union of an infinite chain of finite trees $\{T_n \mid n \in \omega \}$.

A tree may be defined as a set of strings $T \subseteq \{f, g\}^*$ satisfying the following conditions:

(a) if $w, v \in \{f, g\}^*$ and $wv \in T$ then $v \in T$;
(b) for any \( w \in T \), either both \( f(w) \) and \( g(w) \) are in \( T \), or neither \( f(w) \) nor \( g(w) \) is in \( T \) (any nonleaf node has two successors).

The empty word, denoted \( \lambda \), refers to the root of any tree.

For the purpose of the definitions to follow, we distinguish between two kinds of leaves in the tree: the open leaves and the closed leaves. Let \( T \) and \( T' \) be two trees. The composition of \( T \) and \( T' \), denoted \( T' \circ T \), is the tree

\[
T' \circ T = T \cup \{ w'w \mid w' \in T' \text{ and } w \text{ is an open leaf of } T \}.
\]

That is, we obtain \( T' \circ T \) by placing a copy of \( T' \) (i.e. the root of \( T' \)) at every open leaf of \( T \). An element \( v \in T' \circ T \) is an open leaf iff it is of the form \( v = w'w \), where \( w \) and \( w' \) are open leaves of \( T \) and \( T' \), respectively.

The trees in the chain \( \{ T_n \mid n \in \omega \} \) are defined inductively as follows:

1. \( T_0 \) has two leaves, both of them open. (see Fig. 4.3.1)

\[
\begin{align*}
\lambda & \quad \downarrow \\
L & \quad \downarrow & \quad R
\end{align*}
\]

Figure 4.3.1. The tree \( T_0 \).

2. Assume that \( T_n \) is defined and that it has exactly two open leaves, both of them on the \( 2^n \)th level. Let \( T'_{n+1} = T_n \circ T_n \). Thus, \( T'_{n+1} \) has exactly 4 open leaves, all of them on the \( 2^{n+1} \)th level. To obtain \( T_{n+1} \), we close 2 of these 4 open leaves. The 2 we leave open are the second and third with respect to the anti-lexicographic order. (see Fig. 4.3.2)
Figure 4.3.2. The structure $T_{n+1}$.

Viewing $T_n$ as a set of strings, it is clear that $T_n \subseteq T_{n+1}$. The tree $T_\omega$ is the union of the infinite chain $T_0 \subseteq T_1 \subseteq \ldots \subseteq T_1 \subseteq \ldots$. A string $w \in \{f, g\}^\ast$ is a closed leaf of $T_\omega$ iff it is a closed leaf of some $T_n$. All the leaves of $T_\omega$ are therefore closed.

To turn $T_\omega$ into a structure, we view $f$ and $g$ as unary functions, and $\lambda$ as a constant. For all strings $w \in T_\omega$,

$$
\begin{align*}
    f(w) &= \begin{cases} fw & \text{if } fw \in T_\omega, \\ w & \text{otherwise}\end{cases}, \\
    g(w) &= \begin{cases} gw & \text{if } gw \in T_\omega, \\ w & \text{otherwise}\end{cases}.
\end{align*}
$$

The result proved is the following.

**Proposition 4.3.2.** (Urzyczyn). The structure $T_\omega$ has the unwind property for fap but not for fapC. *

We can use this result to show that the te equivalences are different for fap and fapC.

**Proposition 4.3.3.** (Urzyczyn). There exist structures $\mathcal{M}$ and $\mathcal{N}$ such that $\mathcal{M} \equiv_{\text{tefap}} \mathcal{N}$ but that $\mathcal{M} \not\equiv_{\text{tefapC}} \mathcal{N}$.
Proof. Let $L$ be the language with one constant symbol $\lambda$, and two unary function symbols, $f$ and $g$. Let $\mathcal{M} = T_{\omega_0}$, and $T$ be the theory $\text{Th}_L(\mathcal{M})$. Let $\mathfrak{N}$ be any nonstandard model of $T$. We are told of the existence of $\mathfrak{N}$ by the upward Löwenheim-Skolem theorem as $\mathcal{M}$ is infinite.

Since $\mathcal{M} \equiv_0 \mathfrak{N}$, by 4.3.1 we have that $\mathcal{M} \equiv_{\text{fap}} \mathfrak{N}$.

But now we can show that $\mathcal{M} \not\equiv_{\text{fap}C} \mathfrak{N}$; we do this by showing $\mathcal{M} \not\equiv_{\text{eds}} \mathfrak{N}$ and using 2.8.4.

Let $\{t_1, \ldots, t_k, \ldots\}$ be the (r.e.) set of ground terms (ordered according to some suitable Gödel ordering) of the algebraic sort $s$ over $L$. Let $P$ be the scheme of argument sort $(1; s)$ and input variable $x$

$$P = \{ x = t_1 \rightarrow \text{true} \} \cup \{ x \neq t_1 \land \ldots \land x \neq t_k \land x = t_{k+1} \rightarrow \text{true} | k \geq 1 \};$$

$P$ is an eds because the set of clauses is r.e.

On a model $\mathfrak{N}$ of $T$, $P^{\mathfrak{N}}$ will be total iff $\mathfrak{N} \equiv \mathcal{M}$; so $P^{\mathfrak{N}}$ is non total and $\mathcal{M} \not\equiv_{\text{fap}C} \mathfrak{N}$. •

In §5, we prove this result again, but with a different pair of structures. Our pair has the important property that they are both computable. In the above case, we know nothing about the structure $\mathfrak{N}$; we need the structures to be computable in order to be able to prove that $\text{fap}$ is not a congruence for computable structures (5.4.1). The proof relies heavily upon the structure $T_{\omega_0}$ all the same.

The existence of a structure like $T_{\omega_0}$ gives important insight in to the nature of implementability of the natural numbers using fap. Until this example was given, it was not known whether or not there existed a finitely generated infinite structural structure on which the numbers could not be implemented using fap.

Not much appears to be known about the class of structures whose properties lie in the area between implementability of counting and local finiteness.
In [Friedman, 1971], a condition on structures $\mathcal{M}$ known as co-richness was suggested as a condition sufficient to guarantee that $\text{FAP}(\mathcal{M}) = \text{FAPC}(\mathcal{M})$ and the author wondered to what extent it was necessary. We can show that it is plainly not sufficient in the case of void languages; we give the definition first.

**Definition 4.3.1.** Let $L$ be a language. A structure $\mathcal{M}$ of $L$ is termed **co-rich** if there exists an algebraic sort $i$, a ground term $t$ of sort $i$ and fap $P$ of sort type $(1; i, i)$ for which the set

$N = \{ t^\mathcal{M}, P^\mathcal{M}(t^\mathcal{M}), P^\mathcal{M}(P^\mathcal{M}(t^\mathcal{M})), \ldots \}$

is infinite.

The result proved in that paper is the following.

**Proposition 4.3.4.** Let $\mathcal{M}$ be an co-rich structure. Then $\text{FAP}(\mathcal{M}) = \text{FAPC}(\mathcal{M})$. •

From the definition, we can see that if a language $L$ has no constant symbols then no $L$-structure is co-rich; we do however, have the following.

**Proposition 4.3.5.** There exists a finitely generated infinite non-co-rich structure $\mathcal{M}$ for which $\text{FAP}(\mathcal{M}) = \text{FAPC}(\mathcal{M})$.

**Proof.** Take $L$ to be the language of arithmetic without the constant symbol zero. Let $\mathcal{M}$ be the reduct of the standard structure $\mathbb{N}$ to $L$. Then plainly $\text{FAP}(\mathcal{M}) = \text{FAPC}(\mathcal{M})$ because a program can use any of its input variables as a stand-in zero; the same nullary functions are computable by fap and fapC, namely none at all. •

Whilst this might seem to be splitting hairs, its purpose is to illustrate that we must be very careful about how we formulate the intuitive idea that the natural numbers are implementable. An extensive study of the questions in this area is beyond the scope of this work, but we present the following observation in order to introduce an algorithm we will be using at one point in §5.

**Definition 4.3.2.** Let $L$ be a language, $P$ a fap over $L$, and $\mathcal{M}$ an $L$-structure. Suppose that $P^\mathcal{M}$ fails to terminate for a point $m \in \mathcal{M}$. We say $P^\mathcal{M}(m)$ is **periodic** if there exists a program
Proposition 4.3.6. Let $L$ be a finite language and $\mathfrak{M}$ a structure of $L$ finitely generated by a tuple $c$, say. Suppose that for some $l \in \mathbb{N}$, $L^l \mathfrak{M}(c)$ (the set of interpretations $r^\mathfrak{M}(c)$ in $\mathfrak{M}$ of terms $t(x)$ of pebble complexity $\leq l$) is infinite. Then the following are equivalent.

1. there exists a non-total fap $P$ and an argument $m \notin \text{domain}(P^\mathfrak{M})$ with $P^\mathfrak{M}(m)$ non-periodic;
2. $\text{FAP}(\mathfrak{M}_c) = \text{FAP}(\mathfrak{M}_c)$;
3. $\mathfrak{M}_c$ is $\omega$-rich.

Proof. We show (3) $\implies$ (2), (2) $\implies$ (1) and (1) $\implies$ (3).

(3) $\implies$ (2): This is 4.3.4.

(2) $\implies$ (1): Since $L^l \mathfrak{M}(m)$ is infinite, we can reach infinitely many distinct values in $\mathfrak{M}_c$ using a fixed finite set of registers or program variables; furthermore, since there are finitely many sorts, there is one sort, $s_i$ say, for which there are infinitely many values reachable in $\leq l$ registers. Fix on a recursive set $D = \{d_1, \ldots, d_j, \ldots\}$ of codes $d_j$ for the terms $t_j$ of sort $s_i$ over $L_c$ whose pebble complexity is $\leq l$. Now there exists a fapC, $P$ say, which, given an element $m \in M_{s_i}$, finds the least $d_j$ for which $t_j^\mathfrak{M}_c = m$; then outputs the value $t_{p_0}^\mathfrak{M}_c$ of the term $t_{p_0}$ where $d_{p_0}$ is the least code $d_p$ whose value $t_p^\mathfrak{M}_c$ does not occur in $\{t_1^\mathfrak{M}_c, \ldots, t_j^\mathfrak{M}_c\}$. Successively applying $P$ to $t_1^\mathfrak{M}_c$ then generates an infinite set of values in $\mathfrak{M}_c$. By (2), there exists a fap $Q$, say, for which $P^\mathfrak{M}_c = Q^\mathfrak{M}_c$; now we can easily write a non-periodic fap using $Q$.

(1) $\implies$ (3). Suppose $P$ is a fap which does not terminate at $m$, say, and for which $P^\mathfrak{M}(m)$ is non-periodic. Then there is a particular program state which sees infinitely many different interpretations of the program variables, and they are all different, otherwise $P$ would be periodic. Moreover, there is a particular assignment statement which makes assignments to
infinitely many different values in $\mathcal{M}$. It is this set of values that we want to use as our natural numbers; we need a fap-computable successor taking us from one to the next. The problem we have in adapting $P$ is that the sequence of assignments will in general have repetitions.

Let $P$ have program instructions $(P_1, ..., P_t)$; suppose instruction $P_i$ is the assignment in question, $v_p := t(v)$, say. Suppose $P$ has input variables $I_1, ..., I_t$ and program variables $(v_1, ..., v_k)$. Let $t_{\text{first}}$ be a term in $L_\alpha$ whose interpretation is the first value to be assigned at $P_i$.

Let $T$ be the program with one input variable $I$, say, and program variables $(v_1, ..., v_k)$ obtained from $P$ which initially loads the input registers with terms in $L_\alpha$ giving the appropriate values from $m$ and by inserting an instruction $P_*$ between $P_i$ and $P_{i+1}$, where $P_*$ is the instruction

if $v_p = I$ then stop else go to $P_{i+1}$.

Let $T_{\text{next}}$ be the program obtained from $P$ by inserting a new first instruction go to $P_{i+1}$ and inserting a new instruction stop between $P_i$ and $P_{i+1}$.

The purpose of the program $T$ is to set up all of the program variables in the right way; the program $T_{\text{next}}$ takes the execution of $P$ through to the next $P_i$ assignment. Our successor program $S$ now, given a 'number' $x$, searches through the sequence of assignments until it encounters a value which did not occur before $x$. The flowchart for $S$ is shown in Fig. 4.3.3.

The fact that counting cannot be implemented on a structure with the unwind property for fap is represented in the following.

**Proposition 4.3.5.** Let $\mathcal{M}$ be a structure of a language $L$ with the unwind property. Then for every fap $P$ and every point $m$ at which $P^\mathcal{M}$ is undefined, $P^\mathcal{M}(m)$ is periodic.

**Proof.** Suppose not. Then by 4.3.4, $\text{FAP}(\mathcal{M}_m) = \text{FAPC}(\mathcal{M}_m)$, and we can write any non-unwinding routine on the naturals, given input variables corresponding to our numbers.
Figure 4.3.3. The successor program.
4.4 Saturation

We have used the word *type* to mean a set of quantifier-free formulae over a set of free variables \( x \) in a language. The reason we chose this word is that in model theory, there is a parallel notion in which a type is a maximal consistent set of formulae over a set \( x \) of free variables. In this section, we aim to reinforce the parallels between these contexts, and see if any of the related ideas or results in model theory are relevant or applicable here. The motivation in model theory is arguably the same as it was in our consideration of distinguishability; that a set of formulae over a set of free variables expresses everything you know about any point in a structure satisfying those formulae. Once again, in making direct comparisons to our work, we are forced to carefully consider where we can think in terms of many sorted languages, and where we are forced to work in single sorted languages. The first part of this section deals with the analogy between our application and model theory.

4.4.1. Model-theoretic saturation

The following definitions are from [Keisler, 1977].

**Definition 4.4.1.** A *type* over a complete theory \( T \) of a language \( L \) is a maximal consistent set of first order formulas \( \Phi(x) \) for some tuple \( x \) of variables over \( L \).

**Definition 4.4.2.** Let \( \kappa \) be a cardinal. A structure \( M \) is \( \kappa \)-saturated if for each \( X \subseteq M \) of cardinal less than \( \kappa \), every type over \( \text{Th}(M_X) \) is satisfied in \( M_X \). The structure \( M \) is saturated if it is \( \kappa \)-saturated where \( \kappa \) is the cardinal of \( M \).

We are going to look at what happens if we try to compute over a saturated structure, and what becomes of the notion of saturation if we just consider types from some algorithmic equivalence. First, however, some more definitions.

Recall the definition of unwinding of programs; we are going to generalise it to include programs with parameters.
Definition 4.4.3. Let $S$ be a definable function of argument sort $\tau = (m; i_1, \ldots, i_m)$. We say $f$ unwinds on a structure $\mathcal{M}$ if there exists a loop free (i.e. finite) $S' \subseteq S$ such that whenever $m$ is a $\tau$-ary tuple from $\mathcal{M}$, and $S^\mathcal{M}(m)$ is defined, $S'^\mathcal{M}(m)$ is defined.

Now consider $S$ as function with $t$ parameters, where $t \leq m$, and let

$$ p = (p_1, \ldots, p_t) \in M_{i_1} \times \ldots \times M_{i_t}; $$

we say $S$ unwinds at $p$ if there exists a finite $S' \subseteq S$ such that whenever $m \in M_{i_{t+1}} \times \ldots \times M_{i_m}$ and $S^\mathcal{M}(p, m)$ is defined, $S'^\mathcal{M}(p, m)$ is defined. Regarding $S$ as a function with $t$ parameters, we say that $S$ unwinds on $\mathcal{M}$ if $S$ unwinds at $p$ for each $p \in M_{i_1} \times \ldots \times M_{i_t}$.

Note that if $f$ unwinds (without parameters) then $f$ unwinds with $t$ parameters for each $0 < t \leq k$, but the converse is not necessarily true.

We shall be considering only $\omega$-saturated structures, but clearly if a structure is $\kappa$-saturated and $\kappa \geq \omega$ then it will be $\omega$-saturated too, so all our results will apply to other saturated structures.

Proposition 4.4.1. Let $\mathcal{M}$ be an $\omega$-saturated structure over a language $L$. Then, for any class of definable functions $C$, every scheme with (or without) parameters $S \in C$ for which with some choice of parameters $p$, $S(p)^\mathcal{M}$ is total, $S$ unwinds on $\mathcal{M}$ at $p$.

In other words, every total scheme unwinds, with or without parameters.

Proof. Let $p \in \mathcal{M}$ and suppose that $S$ does not unwind on $\mathcal{M}$ at $p$. Let $L_p$ be the language $L$ extended with constant symbols $P = \{p_1, \ldots, p_t\}$ for $p$; we therefore have that $S$ (as a function without parameters) does not unwind on the structure $\mathcal{M}_p$ of $L_p$. Suppose that $S$ is the scheme

$$ S = \{ E_{i_s}(x) \land E_{i_s}(x) \land \ldots \land E_{i_s}(x) \rightarrow t_s(x) \land s \in S \}; $$

let $\phi_s(x)$ be the quantifier-free relation

$$ \phi_s(x) = E_{i_s}(x) \land E_{i_s}(x) \land \ldots \land E_{i_s}(x) $$

Chapter 4 – Types and type-based equivalences – Saturation
over \( L_p \) which expresses the fact that \( S \) is defined at \( x \) with parameters \( p \) by clause \( s \). Let \( \Psi \) be the set
\[
\Psi = \{ -\phi_s(x) \mid s \in S \}.
\]
Following the proof of 2.5.1, we have that \( \Psi \) is consistent with \( \text{Th}(M_p) \) and extends to a type \( \Xi \) over \( \text{Th}(M_p) \); since \( M \) is saturated, \( \Xi \) is satisfied in \( M_p \), so \( S(p) \) is not total on \( M \). •

So, in a saturated structure, we have that every total definable function (with or without parameters) unwinds. The next result shows that if we restrict the notion of type to just those relevant to the type-based equivalences, the converse is true. In other words, the property analogous to saturation in the context of computability is strongly related to the unwind property. To do this we need some more definitions.

**Definition 4.4.4.** Let \( L \) be a language, and \( C \) a class of programs over \( L \). Let \( \Gamma_C \) be the set of type-observations (of form \( a \)) corresponding to \( te_C \) (in the sense of 4.1.1). A structure \( M \) of \( L \) is *saturated* w.r.t. \( C \) if for each sort type \( \tau = (m; i_1, \ldots, i_m) \) and every \( \tau \)-ary tuple \( m \in M \), every type observation from \( \Gamma_C \) over \( L_m \) consistent with \( \text{Th}(M_m) \) is supported in \( M_m \).

**Proposition 4.4.2.** Let \( C \) be a class of programs. Any \( \omega \)-saturated structure \( M \) is saturated w.r.t. \( C \).

**Proof.** Let \( \Phi \) be a type for \( \Gamma(L_m) \) consistent with \( M_m \). Then \( \Phi \) extends to a type (in the model-theoretic sense); so \( \Phi \) is satisfied in any \( \omega \)-saturated structure. •

As we suggested, the converse is true if we restrict attention to our own types.

**Proposition 4.4.3.** Let \( M \) be a structure over a language \( L \), and \( C \) a class of programs. Then the following are equivalent.

1. Every total function in \( C \) (with or without parameters) unwinds on \( M \).
2. The structure \( M \) is saturated w.r.t. \( C \).

**Proof.** Remember that the set of type-observations corresponding to the equivalence \( te_C \) over a language \( L_m \) is the set \( \{ \exists x. \text{undefined}(P, x) \mid P \text{ is a scheme from } C \text{ over } L_m \} \).
So \( \mathcal{M} \) saturated w.r.t. \( C \)

\[ \iff \forall \tau, \tau\text{-ary } m \in \mathcal{M}.
\forall \text{ type observations } \gamma \text{ over } L_m.
\gamma \text{ consistent with } \text{Th}(\mathcal{M}_m) \implies \mathcal{M}_m \models \gamma \]

\[ \iff \forall \tau, \tau\text{-ary } m \in \mathcal{M}.
\forall \text{ type observations } \gamma \text{ over } L_m.
\mathcal{M}_m \not\models \gamma \implies \gamma \text{ inconsistent with } \text{Th}(\mathcal{M}_m) \]

\[ \iff \forall \tau, \tau\text{-ary } m \in \mathcal{M}.
\forall \text{ type observations } \gamma \text{ over } L_m.
\mathcal{M}_m \not\models \gamma \implies \gamma \text{ finitely inconsistent with } \text{Th}(\mathcal{M}_m) \text{ (by compactness)} \]

\[ \iff \forall \tau, \tau\text{-ary } m \in \mathcal{M}.
\forall \text{ type observations } \gamma = \exists x. \bigwedge \{ \neg \phi_1(x), ..., \neg \phi_n(x) \} \text{ over } L_m.
\mathcal{M}_m \not\models \gamma \implies \text{Th}(\mathcal{M}_m) \vdash \forall x. \neg \bigwedge \{ \neg \phi_1(x), ..., \neg \phi_n(x) \} \text{ for some } n \]

\[ \iff \forall \tau, \tau\text{-ary } m \in \mathcal{M}.
\forall \text{ type observations } \gamma = \exists x. \bigwedge \{ \neg \phi_1(x), ..., \neg \phi_n(x) \} \text{ over } L_m.
\mathcal{M}_m \not\models \gamma \implies \text{Th}(\mathcal{M}_m) \vdash \forall x. \bigvee \{ \phi_1(x), ..., \phi_n(x) \} \text{ for some } n \]

\[ \iff \forall \tau, \tau\text{-ary } m \in \mathcal{M}.
\forall \text{ type observations } \gamma = \exists x. \text{undefined}(P, x), P \in C \text{ over } L_m.
\mathcal{M}_m \not\models \gamma \implies P \text{ unwinds on } \mathcal{M}_m \]

\[ \iff \forall t \text{ and set of parameters } m = \{ m_1, ..., m_m \}.
\forall \text{ functions } P \text{ from } C \text{ over } L_m.
P \text{ not total on } \mathcal{M}_m \implies P \text{ unwinds on } \mathcal{M}_m \]

\[ \iff \text{ every total function with or without parameters unwinds. } \]

Chapter 4 – Types and type-based equivalences – Saturation
So, now we can look at all the known results about saturated structures, and see which apply or
generalise to this case; for example, we have the following existence theorem.

Proposition 4.4.4. (Vaught, [1962]). Every complete theory $T$ with an infinite model has a
$(\kappa^*)$-saturated model of cardinal $2^\kappa$. (The cardinal $\kappa^*$ is the least cardinal greater than $\kappa$.) •

Restated in our terms, that is

For any class of programs $C$ and complete theory $T$ with an infinite model, there exists a model
$\mathcal{M}$ of $T$ of cardinal $2^\kappa$ which is saturated w.r.t. $C$.

In the case of model-theoretic saturation, there is no theorem positing the existence of a
countable saturated structure. However, in our case, we can now use this result to give us a
countable model of $T$ saturated w.r.t. $C$.

Proposition 4.4.5. Let $T$ be a theory over a language $L$ which has an infinite model, and $C$
a class of programs. Then there exists a countable model $\mathcal{M}$ of $T$ which is saturated w.r.t. $C$.

Proof. Let $T_1 = \text{Th}_L(\mathcal{R})$, where $\mathcal{R}$ is an infinite model of $T$. From the above proposition, we
have a model $\mathcal{N}$ of $T_1$ and hence of $T$ which is saturated w.r.t. $C$. We will extend $L$ by a
countable set of constants to $L'$ and extend $T_1$ to $T'$ over $L'$ such that any model $\mathcal{M}$ of $T'$ has
the property that its reduct to $L$ is $te_C$ to $\mathcal{N}$. Then the Löwenheim-Skolem theorem will give us a
countable model of $T$ with the required property.

Notice that if $\mathcal{M}$ is a model of $T_1$ and $P$ from $C$ unwinds on $\mathcal{N}$ then $P$ unwinds on $\mathcal{M}$ by 4.3.1;
further $P$ is total on $\mathcal{M}$ iff it is total on $\mathcal{N}$. So it is only the programs which do not unwind
which we need worry about.

Extend $L$ with one $\tau$-ary tuple of constant symbols $c_P$ for each $\tau$-ary scheme $P \in C$ which does
not unwind in $\mathcal{N}$. Since $\mathcal{N}$ is saturated w.r.t. $C$, $P$ is non-total on $\mathcal{N}$; expand $\mathcal{N}$ to $L'$ by
interpreting $c_P$ to any tuple $n \in \mathcal{N}$ for which $P^\mathcal{M}(n)$ is undefined. Now let $T' = \text{Th}_{L'}(\mathcal{M})$. 
Suppose $\mathcal{M}$ is a model of $T$, and $P$ is a program over $L$ in $C$. If $P$ is total on $\mathcal{N}$, then it unwinds in $\mathcal{N}$, so by 4.1.1 again, it unwinds in $\mathcal{M}$ and is total on $\mathcal{M}$. If $P$ is not total on $\mathcal{N}$ and $P$ does not unwind in $\mathcal{N}$, then there is a tuple of constant symbols $c_P$ for which $T \not\models undefined(P, c_P)$;

so $P$ is non-total on $\mathcal{M}$, and hence $\mathcal{N} \equiv_{ec} \mathcal{M}$. •

However, we shall also want effective saturated structures; some examples of these for specific classes of programs appear in the literature. The structure $T_0$ (§4.3), for example, has the unwind property and is reachable, and is therefore saturated.

Another result about the model-theoretic notion of saturation is the following uniqueness result.

**Proposition 4.4.6.** ([Vaught, 1962]). Any two elementarily equivalent saturated structures of the same cardinal are isomorphic. •

This suggests that saturation is quite a strong property: there is certainly no way two structures could be isomorphic if they were not elementarily equivalent or if they were not of the same cardinal.

Since our notion of saturation is not as strong, it is possible that there is not a corresponding uniqueness result, but this question is open. There are a few miscellaneous results about saturated structures which may prove useful.

Here is another result from model theory.

**Proposition 4.4.7.** ([Vaught, 1961]). Let $T$ be a complete theory. Then $T$ has a countable saturated model iff for each $n$, $T$ has countably many types in $n$ variables. •

**Corollary.** If a complete theory $T$ has only countably many nonisomorphic countable models, then $T$ has a unique countable model saturated w.r.t. every class of definable functions. •
Example. The theory $T$ of algebraically closed fields of characteristic $p$ has countably many countable models, one for each transcendence degree $n$ or $\omega$. Therefore $T$ has a countable saturated model. A model of transcendence degree $n$ omits any type of $n+1$ algebraically independent elements. Thus the fields of transcendence degree $\omega$ must be the countable saturated model (necessarily unique by the uniqueness of saturated models of given cardinality).

4.4.2. Completeness

We can think of saturation as a topological compactness or completeness-type property.

Consider the following example. Suppose that $\mathcal{M}$ and $\mathcal{N}$ be structures which are $te_{loop}$-*equivalent but not $te_{C}$-equivalent for some class $C$ of programs. Then we may assume w.l.o.g. that for some scheme $P \in C$, $P^\mathcal{M}$ is total and that $P^\mathcal{N}$ is not.

Now, for each $i \in \mathbb{N}$, since the structures are $te_{loop}$-*equivalent, there is a point $m_i$ in $\mathcal{M}$ for which $P$ fails to terminate after $i$ steps; and for each of these points $m_i$ in $\mathcal{M}$, there is a point $n_i$ in $\mathcal{N}$ for which $P^\mathcal{N}(n_i)$ takes the same number of steps to terminate as $P^\mathcal{M}(m_i)$. We can think of the sets \{m_i | i \in \mathbb{N}\} and \{n_i | i \in \mathbb{N}\} as sequences in the structures $\mathcal{M}$ and $\mathcal{N}$ respectively, and the sequence in $\mathcal{N}$ having a 'limit' (a point where $P^\mathcal{N}$ is undefined) (see Fig. 4.4.1).

![The structure $\mathcal{M}$ has a sequence but no limit.](image1)

![The structure $\mathcal{N}$ has a sequence and a limit.](image2)

**Figure 4.4.1.** Limits in structures.
A notion of completeness of a structure can be formed by insisting that every such sequence has a limit in the above sense. We will show that this idea is very closely related to that of saturation, and gives us an example of how we can distinguish termination equivalences.

In defining completeness, we need to use the notion of the set of types corresponding to a class C of definable functions, since there is no notion of 'number of steps' for these generalised models of definability.

**Definition 4.4.5.** Let L be a language and Ξ be a set of types over L. A structure M is said to be complete for Ξ if M ⊨ γφ whenever φ ∈ Ξ and M ⊨ γφ′ for every finite subset φ′ ⊆ φ.

We will talk about a structure being complete for a class C of definable functions if it is complete for the set of types ΞC corresponding to C.

A compactness argument similar to that in 2.5.1 gives us

**Proposition 4.4.8.** Let C be a class of definable functions. A structure M is complete for ΞC iff every C total on M unwinds on M.

So completeness is the same as saturation except that we are not concerned about parameters.

We can use this notion of completeness to separate some termination equivalences using the following result.

**Proposition 4.4.9.** Let L be a language and let C₁ and C₂ be classes of definable functions over L. Suppose that M is a structure which is complete for C₁ but not for C₂. Then the termination equivalences for C₁ and C₂ are different.

**Proof.** Extend L to L' by adding tuples of constants csᵢ for each sᵢ ∈ C₁ with sᵢ M non-total.

Let M' be M expanded to L' by interpreting csᵢ to be any point not in \( \text{domain}(sᵢ M) \); let T = Thₜₑₜ(M'). Since M is not complete for C₂, there exists a C₂ scheme S of argument type τ, say, which is total but which does not unwind on M. Following the proof of 2.5.1 we suppose that S is the scheme.
\[ S = \{ E_1(x) \land E_2(x) \land \ldots \land E_k(x) \rightarrow t_s(x) \mid s \in S \}. \]

Now extend \( L' \) to \( L'' \) by adding a \( \tau \)-ary tuple of constant symbols \( c \). Extend \( T' \) to \( T'' \) by adding the sentences \( \{ \neg (E_1(c) \land E_2(c) \land \ldots \land E_k(c)) \mid s \in S \} \). By compactness, since \( S \) does not unwind on \( \mathcal{M} \), we know that \( T'' \) is consistent; so let \( \mathcal{N}'' \) be any model of \( T'' \), and let \( \mathcal{N} \) be its reduct to \( L \).

Clearly \( \mathcal{M} \equiv_{teC_2} \mathcal{N} \) since \( P \) is total on \( \mathcal{M} \) but not on \( \mathcal{N} \). We will show that \( \mathcal{M} \equiv_{teC_1} \mathcal{N} \).

Suppose that a program \( P \in C_1 \) and that \( P^\mathcal{M} \) is total. Since \( \mathcal{M} \) is complete, \( P \) unwinds on \( \mathcal{M} \). Now \( \mathcal{N} \) is elementary equivalent to \( \mathcal{M} \) and therefore \( P \) unwinds and is total on \( \mathcal{N} \) by 4.3.1.

Conversely, suppose that \( P \in C_1 \) and that \( P^\mathcal{M} \) is non-total. Then \( P^\mathcal{N}'' \) is undefined at \( c_p^\mathcal{N}'' \); so \( P^\mathcal{M} \) is non-total.

Another intuition is that if a structure is complete then it remains complete if more points are added; this is reflected in the following fact.

**Proposition 4.4.10.** Let \( \mathcal{M} \) and \( \mathcal{N} \) be structures of a language \( L \), and suppose \( \mathcal{M} \) is a basic substructure of \( \mathcal{N} \). (A basic substructure is a basically equivalent substructure.) Then for any class of programs \( C \), if \( \mathcal{M} \) is complete for \( C \), then so is \( \mathcal{N} \), and \( \mathcal{M} \equiv_{teC} \mathcal{N} \).

**Proof.** Let \( S \) be from \( C \) over \( L \).

Then \( S \) total on \( \mathcal{N} \)

\[ \Rightarrow S \text{ total on } \mathcal{M} \quad \text{since } \mathcal{M} < \mathcal{N} \]

\[ \Rightarrow S \text{ unwinds on } \mathcal{M} \quad \text{since } \mathcal{M} \text{ is complete for } C \]

\[ \Rightarrow S \text{ unwinds on } \mathcal{N} \quad \text{by basic equivalence and 4.3.1.} \]

The idea behind many of the examples in §5.1 is the same; we have found a pair of structures by taking a non-saturated structure and adding a limit point. However, the examples there are not complete in this sense because we are only interested in adding in certain limit points, as
opposed to all of them. It is this idea that leads us to look at a notion of relative completeness or saturation next.

4.4.3. Relative saturation

There is another notion of saturation, a generalisation of the one we have given, and that is saturation relative to a class of structures.

Definition 4.4.6. Let $L$ be a language, and $K$ a class of structures over $L$. Let $C$ be a class of definable functions. A structure $M$ is $K$-saturated w.r.t $C$ if

for each $S \in C$, $S^M$ total $\Rightarrow$ (total for each $M \in K$).

This definition relates to the other one by the following.

Proposition 4.4.11. Let $M$ be a structure over a language $L$; for each $\tau$ and $\tau$-ary tuple $p$, let $K_p$ be the class of all models of $\text{Th}_L(M_p)$; let $C$ be a class of definable functions.

Then $M$ saturated w.r.t. $C$ $\iff$ for each $p \in M$, $M_p$ is $K_p$-saturated w.r.t. $C$.

Proof. $\Rightarrow$: Suppose $M$ saturated w.r.t. $C$. Pick $p \in M$, and $S \in C$ over $L_p$ total on $M_p$.

Let $\mathcal{N}_p$ be a model of $\text{Th}_L(M_p)$; we must show that $S_{\mathcal{N}_P}$ is total. But we know that $S^M$ with parameters $p$ is total, so since $M$ is saturated w.r.t. $C$, $S_p$ unwinds in $M_p$; now if $\mathcal{N}_p \in K_p$, $S$ is total on $\mathcal{N}_p$ by basic equivalence, and 4.3.1.

$\Leftarrow$: Conversely, let $S \in C$ over $L$ be total on $M$ with parameters $p$; we will assume $M$ does not unwind with parameters $p$, and show $M_p$ is not $K_p$-saturated w.r.t. $C$.

Let $T = \text{Th}_L(M_p)$; it is clear that $\text{Th}_L(M) \subseteq T$. Let $S_p$ be the scheme in $C$ over $L_p$ with constants $c$ substituted for the parameter input variables; then $S_p^{M_p}$ is total and does not unwind. Therefore, by 2.5.1 there exists a model $\mathcal{N}_p$ of $T$ on which $S_p$ is non total.

Therefore $M_p$ is not $K_p$-saturated w.r.t. $C$.

Chapter 4 – Types and type-based equivalences – Saturation 120
Proposition 4.4.12. Let $L$ be a language and $\mathcal{M}$ an $L$-structure. Then

1. $\mathcal{M}$ is $\{\mathcal{M}\}$ saturated w.r.t. $C$ for all $C$;
2. $\mathcal{M}$ is $K$-saturated w.r.t. $C$ for all $C$ if $K$ is the class of all substructures of $\mathcal{M}$;

Proof. Suppose $S$ is a definable function in $C$ and $S^\mathcal{M}$ is total. Then clearly $S^\mathcal{N}$ is total

1. for every $\mathcal{N} \in \{\mathcal{M}\}$;
2. for every $\mathcal{N} < \mathcal{M}$. •

In some ways, a $K$-saturated structure has every type of element present in any of the structures in $K$. This intuition is reinforced in the next two facts.

Proposition 4.4.13. Let $L$ be a language and $K$ a collection of finitely generated $L$-structures. Then the following are equivalent:

1. $\mathcal{M}$ is $K$-saturated w.r.t. $G_0$;
2. for each $\mathcal{N} \in K$, there is an isomorphic embedding $\theta: \mathcal{N} \to \mathcal{M}$.

Proof. $(1) \Rightarrow (2)$: Pick $\mathcal{N} \in K$; then $\mathcal{N}$ is finitely generated, by a $\tau$-ary tuple $n$, say. Let $\Phi(x) = \{\phi_1(x), \ldots, \phi_p(x), \ldots\}$ be the set of all atomic and negated atomic relations with $\phi_p^\mathcal{N}(n) = \text{true}$; let $P$ be the $G_0$ scheme

$$P = \{-\phi_p(x) \to \text{true} \mid 1 \leq p < \infty\}.$$ 

Thus $P^\mathcal{N}$ is non total, and so since $\mathcal{M}$ is $K$-saturated, $P^\mathcal{M}$ is also non total. Then there is a $\tau$-ary tuple $m \in \mathcal{M}$ with $\langle m \rangle \equiv \langle n \rangle \equiv \mathcal{N}$. The isomorphism $\theta: \langle n \rangle \to \langle m \rangle$ is an isomorphic embedding $\mathcal{N} \to \mathcal{M}$.

$(2) \Rightarrow (1)$: Suppose $P^\mathcal{M}$ is total; then for each $\mathcal{N}$, let $\mathcal{N}^* < \mathcal{M}$ be isomorphic to $\mathcal{N}$. Then $P^\mathcal{N}^*$ is total. •

The next fact is similar but involves computable structures.
Proposition 4.4.14. Let $L$ be a finite language and $K$ a collection of finitely generated computable $L$-structures. Then the following are equivalent:

(1) $M$ is $K$-saturated w.r.t. $\text{fapCS}$;
(2) for each $\mathfrak{M} \in K$, there is an isomorphic embedding $\theta: \mathfrak{N} \to \mathfrak{M}$.

Proof. Similar to the last result. The difference is that for a computable structure, the set of atomic relations satisfied at a point is recursive (2.7.7). So the scheme we obtain is an eds, since it involves only finitely many operation symbols. •

The implications (1) ⇒ (2) in the last two results fail if we drop the assumption about the structures in $K$ being finitely generated in 4.4.13 and 4.4.14; the latter also fails if we relax either of the conditions about computability or a finite language. For examples, the reader can pick appropriate ones from §5, but the basis of the argument is the following.

Proposition 4.4.15. Suppose that $C_1$ and $C_2$ are two classes of definable functions and that $\text{te}_{C_1}$ is different to $\text{te}_{C_2}$ for a language $L$. Then there exists a class $K$ of $L$-structures and structures $\mathfrak{M}$ and $\mathfrak{N}$ with

(1) $\mathfrak{M}$ $K$-saturated w.r.t. $C_1$;
(2) $\mathfrak{N}$ $K$-saturated w.r.t. $C_2$;
(3) an isomorphic embedding $\mathfrak{N} \to \mathfrak{M}$ for each $\mathfrak{N} \in K$;
(4) no isomorphic embedding $\mathfrak{N} \to \mathfrak{M}$ for some $\mathfrak{N} \in K$.

Proof. We observe first of all that if for two structures $\mathfrak{M}$ and $\mathfrak{N}$ there exist isomorphic embeddings $\mathfrak{M} \to \mathfrak{N}$ and $\mathfrak{N} \to \mathfrak{M}$, then $\mathfrak{M} \equiv_{\text{te}C} \mathfrak{N}$ for every class $C$. This is because any definable function $S$ with $S^\mathfrak{M}$ total has $S^\mathfrak{N}$ total for $\mathfrak{N} \equiv \mathfrak{N}^* < \mathfrak{M}$; similarly the other way around, so $\mathfrak{M}$ and $\mathfrak{N}$ are $\text{te}_{C}$ equivalent for all $C$. 

Chapter 4 - Types and type-based equivalences -- Saturation
Since the \( te \) equivalences are different, there exist structures \( M \) and \( N \) with \( M \equiv_{teC_1} N \) and \( M \not\equiv_{teC_2} N \). Suppose w.l.o.g. that there is no isomorphic embedding \( N \rightarrow M \). Now just put \( K \) to be the class \( \{ N \} \).

This idea of \( K \)-saturation turns out to be useful in proving a few results in \( \S 5 \) concerning \( te \) equivalences. The basis of the constructions we use is the following.

**Proposition 4.4.16.** Let \( L \) be a language and \( K \) a class of \( L \)-structures closed under disjoint unions. Then

1. there exists a structure \( M \in K \) which is \( K \)-saturated w.r.t. \( C \); and
2. any two such structures in \( K \) which are \( K \)-saturated w.r.t. \( C \) for some \( C \) are \( teC \)-equivalent.

**Proof.** (1) For \( C \) a class of definable functions, let \( S \subseteq C \) be the subset of those \( p \in C \) for which there exists a \( N_p \in K \) with \( p^{N_p} \) non total. Let \( M \) be the disjoint union

\[
M = \bigcup \{ N_p \mid p \in S \}.
\]

Then clearly \( M \) is \( K \)-saturated w.r.t. \( C \).

(2) Let \( M \) and \( N \) be \( K \)-saturated w.r.t. \( C \). Suppose \( p \in C \) and that \( p^M \) is total. Then \( p \) is total on each \( R \in K \). So \( p^N \) is total. Similarly, \( p^N \) total \( \Rightarrow \) \( p^M \) total. Therefore \( M \equiv_{teC} N \).

This is the basis of the following result which enables us to compare termination equivalences.

**Proposition 4.4.17.** Let \( K \) be a class of structures closed under disjoint unions and \( M \in K \). Suppose that for two classes of programs \( C_1 \) and \( C_2 \), \( M \) is \( K \)-saturated w.r.t. \( C_1 \) but not w.r.t. \( C_2 \). Then \( teC_1 \)-equivalence is different to \( teC_2 \)-equivalence.

**Proof.** By 4.4.15 there exists a structure \( N \) \( K \)-saturated w.r.t. both \( C_1 \) and \( C_2 \); then (by 4.4.15 again) \( M \equiv_{teC_1} N \) but \( M \not\equiv_{teC_2} N \).
In the examples in §5, the class $\mathcal{K}$ is typically the class of all expansions of a class of $L$-structures to an extended language $L'$. The main failing of the proofs of 4.4.9 and 4.4.17 is that they are non-constructive. In the proofs in §5, we are often able to exhibit structures which separate the termination equivalences. This is especially important when we need to know whether or not there exist computable structures separating the equivalences.
In this section, we aim to prove the following result.

Proposition 4.5.1. Let $\mathcal{M} = \langle m \rangle$ be a finitely generated (for a $\tau$-ary tuple $m$) structure of a finite language $L$. Then the following are equivalent.

1. $\mathcal{M}$ is computable;
2. there is an eds $P$ such that for every $L$-structure $\mathcal{N}$ and $\tau$-ary tuple $n \in \mathcal{N}$,

$$P^\mathcal{N}(n) \text{ is undefined } \iff \langle m \rangle \equiv \langle n \rangle.$$

Proof of one direction. By 2.7.7, the set $\Phi$ of all atomic relations and negated atomic relations $\Phi(x) = \{ \phi_1(x), \ldots, \phi_p(x), \ldots \}$ true on the tuple $m$ in a structure is r.e.; so there is an eds $P$, where

$$P = \{ -\phi_p(x) \rightarrow \text{true} \mid \phi_p(x) \in \Phi(x) \},$$

which fails to terminate iff they are all true at a point in a structure. But if they are all true at a point $n$ in $\mathcal{N}$, we have that $\langle n \rangle \equiv \langle m \rangle$. •

Conversely, suppose that $P$ is an eds, where

$$P = \{ C_s \mid s \in S \},$$

where $C_s$ is the clause $C_s = E_{1s}(x) \land E_{2s}(x) \land \ldots \land E_{ks}(x) \rightarrow t_s(x)$

which fails to terminate only at the points $x$ where $\langle x \rangle \equiv \langle m \rangle$, and prove that $\langle m \rangle$ is computable.

We will only be interested in the conjuncts

$$Q = \{ E_{1s}(x) \land E_{2s}(x) \land \ldots \land E_{ks}(x) \mid s \in S \}.$$

Now, if $P$ fails to terminate at a point $x$, then for each $s \in S$, there is an $E_{ps}(x)$ which fails. We will call a set of atomic or negated atomic relations $\{ -E_{ps}(x) \mid s \in S \}$ a complete branch in $P$ (by analogy with the path of execution of $P$ in a decision tree). A subset $B' \subseteq B$ of a complete branch will be called a branch.
A branch $\Phi = \{ -E_{ps}(x) \mid s \in S \}$ is *consistent* if there is a structure where $\Phi$ is satisfied.

Given a set of equalities $E$ between ground terms over $L_x$, let $E$ be the set of all equalities between ground terms that can be derived from them by reflexivity, symmetry, transitivity and substitution.

**Proposition 4.5.2.**

1. A branch $\Phi$ is inconsistent iff there is a finite $S' \subseteq S$ with $\{ -E_{ps}(x) \mid s \in S' \}$ inconsistent.
2. A branch $S'$ is inconsistent iff there exists a finite set $E$ of equalities and a relation symbol $\nu$ of $L$ with $\{ \psi(t_{1s1}, \ldots, t_{1sm}), \psi(t_{2s1}, \ldots, t_{2sm}) \} \cup E \subseteq S'$, where the equalities $(t_{1si} = t_{2si})$ are all in $E$;
3. The set of inconsistent finite branches is r.e..

**Proof.** (1) A branch $\Phi = \{ -E_{ps}(x) \mid s \in S \}$ is consistent iff the theory $\{ -E_{ps}(x) \mid s \in S \}$ is consistent over the language $L_x$. By compactness, if every finite subset of a first order theory is consistent, then the whole theory is consistent.

(2) Let $\Phi$ be a branch; by (1) we may assume $\Phi$ is finite. Let $L'$ be $L_x$ without the relation symbols and let $E$ be the set of all equalities in $\Phi$; let $M$ be the initial algebra of $E$ over the language $L'$. We will suppose that there is no relation symbol $\psi$ with

$$\{ \psi(t_{1s1}, \ldots, t_{1sm}), \psi(t_{2s1}, \ldots, t_{2sm}) \} \cup E \subseteq S',$$

where the equalities $(t_{1si} = t_{2si})$ are all in $E$ and give an expansion $\mathfrak{N}$ of $M$ to $L_x$ satisfying $\Phi$.

Given a relation symbol $\psi$ of $L$ of argument type $\tau = (m; i_1, \ldots, i_m)$ and a $\tau$-ary tuple $v$ of elements $v = (v_1, \ldots, v_m)$ of $M$, define

$$\psi^\mathfrak{N}(v_1, \ldots, v_m) = \begin{cases} \text{true} & \text{if there exist ground terms } t_1, \ldots, t_m \text{ over } L_x \\
\text{false} & \text{otherwise.} \end{cases}$$

with $\psi(t_1, \ldots, t_m) \in S'$ and $t_i^\mathfrak{N} = v_i$ for each $i$.\[\text{Chapter 4 — Types and type based equivalences — Computable structures}\] 126
Now \( \mathcal{M} \) will satisfy \( \Phi \) since whenever the atomic relations \(-\psi(t_1)\) and \(\psi(t_2)\) are in \( S' \), by initiality, \( \mathcal{M} \models t_1 \neq t_2 \).

(3) Given a finite set of simple equations \( E \), we know that the set of simple equations in \( E \) is r.e. So, let \( S' \) be a finite branch; the set of ground atomic relations entailed by \( S' \) is r.e.; so the set of finite branches \( S' \) entailing \( \bot \), i.e. inconsistent branches, is r.e..

The next steps are the following.

**Proposition 4.5.3.** Suppose that \( P \) fails to terminate at \( x \) iff \( (x) \equiv (m) \). Then

1. there exists a consistent complete branch;
2. every consistent complete branch decides every atomic relation;
3. there is only one consistent complete branch.

**Proof.** (1) The branch corresponding to the non termination of \( P \) at \( m \) on \( \mathcal{M} \) is complete and consistent.

(2) Suppose not; let \( \Phi \) be a branch and \( \psi(t_1, \ldots, t_m) \) an atomic relation such that both \( \Phi \cup \{\psi(t_1, \ldots, t_m)\} \) and \( \Phi \cup \{-\psi(t_1, \ldots, t_m)\} \) are consistent. Let \( \mathcal{R} \) and \( \mathcal{P} \) be structures such that \( \mathcal{R} \) satisfies \( \Phi \cup \{\psi(t_1, \ldots, t_m)\} \) and \( \mathcal{P} \) satisfies \( \Phi \cup \{-\psi(t_1, \ldots, t_m)\} \); then \( P \) fails to be defined on both of the (non-isomorphic) structures \( \langle x^\mathcal{R} \rangle \) and \( \langle x^\mathcal{P} \rangle \).

(3) Given two consistent branches \( \Phi_1 \) and \( \Phi_2 \), there is a clause \( s \) in \( P \) at which they disagree in that \( E_s(x) \) is in \( \Phi_1 \) and \( -E_s(x) \) is in \( \Phi_2 \). Let \( \mathcal{R} \) and \( \mathcal{P} \) be structures such that \( \mathcal{R} \) satisfies \( \Phi_1 \) and \( \mathcal{P} \) satisfies \( \Phi_2 \); then \( P \) fails to be defined on both of the (non-isomorphic) structures \( \langle x^\mathcal{R} \rangle \) and \( \langle x^\mathcal{P} \rangle \).

Now we fix on an enumeration \( s_1, \ldots, s_j, \ldots \) of \( S \). It is possible to do this in such a way that there is a program \( A \) which, given \( j \), generates (a code for) the clause \( E_{s_j}(x) \land \ldots \land E_{k_j}(x) \rightarrow t_j \) since the set of clauses in \( P \) is r.e.
Given a consistent finite branch $B = \{ \neg E_p(x) \mid s \in S \}$ we define the depth of $B$ to be the largest $y$ (if it exists) such that there is a consistent branch $B' = \{ \neg E_p(x) \mid s \in \{ s_1, ..., s_y \} \}$ extending $B$; if no largest $y$ exists we say $B$ has infinite depth.

**Proposition 4.5.4.** Every consistent finite branch $B = \{ \neg E_p(x) \mid s \in \{ s_1, ..., s_t \} \}$ of infinite depth extends to a consistent complete branch.

**Proof.** We will show that there exists a conjunct $E$ in $C_{s+t+1}$ such that $B \cup \{ \neg E \}$ is consistent and has infinite depth; this will give us the result by induction. Let $E_1, ..., E_k$ be the conjuncts in $C_{s+t+1}$. Since the set $B$ of finite branches $\{ \neg E_p(x) \mid s \in \{ s_1, ..., s_t \} \}$ extending $B$ is the union $B = B_1 \cup ... \cup B_k,$

where for each $i$, $B_i = \{ \text{finite branches extending } B \cup \{ \neg E_i \} \}$,

and $B$ contains branches of arbitrarily large size, there is a $B_{i_0}$ which contains branches of arbitrarily large size. (Otherwise (largest branch in $B$) = $\max_{1 \leq i \leq k} (\text{largest branch in } B_i)$.) So, pick $E$ to be $E_{i_0}$.

This tells us that if a branch $B = \{ \neg E_p(x) \mid s \in \{ s_1, ..., s_t \} \}$ does not extend to a consistent complete branch in $P$ then there is a number $y$ such that

\begin{equation}
(*) \quad B \text{ does not extend to a consistent complete branch in } B' = \{ C_s \mid s \in \{ s_1, ..., s_y \} \}.
\end{equation}

Since there are finitely many branches in any finite scheme $B'$, the set of those $y$ for which $(*)$ holds is r.e.; therefore the set of finite branches which do not extend to the consistent complete branch is r.e..

**Proof of other direction of 4.5.1.** We show that the set of atomic relations true at $m$ in $\mathfrak{M}$ is recursive. We will do this by giving an effective procedure that determines for each $j$, which of the conjuncts from $C_{s_j}$ is in the single consistent complete branch. Since every atomic
relation appears in the branch (up to equalities), the set of atomic relations true in \( M \) will then be recursive.

The procedure recursively determines the initial segment \( \{-E, ..., E_{j+1}\} \) of the complete branch. In the base case \( j=1 \) this will be empty. Then for each of the conjuncts \( E_i \) of clause \( C_{s_j} \), it tests which of the extensions \( B_i = \{-E_{s_1}, ..., E_{s_{j-1}}, \neg E_i\} \) are not extendable to the complete branch; since there are finitely many conjuncts, these tests will terminate for all but one of them, \( E_{i_0} \) say. Then \( \neg E_{i_0} \) is the contribution from \( C_{s_j} \) in the consistent complete branch.
Many statements and proofs of results have been deferred from earlier in the thesis to this chapter; this is because I feel that many of them are so closely related that it would be useful to collect them all together in one place. These results deal with two issues.

1. the relationship between different te and sp equivalences, especially concerning showing that equivalences obtained from different programming formalisms are indeed different;
2. the extent to which our equivalences are congruences with respect to the join datatype building operation; for those that are not, we look at the congruences they generate.

Results relating to (1) are presented in the first section, §5.1; those relating to (2) in the remaining sections.

5.1 Separation

A number of different equivalences have been introduced, and some have been shown to be similar in either every or some circumstances. In this section, we settle almost all of the questions as to whether the remaining equivalences are the same as one another, or different. As a summary, we will first recall the equivalences and their known relationships.

The first equivalences introduced were well known from classical logic –

- elementary equivalence (≡) wherein two equivalent structures satisfy the same first order sentences;
- basic equivalence (≡₀) wherein the same quantifier free formulas are satisfiable on two equivalent structures.
We know that $\equiv$ implies $\equiv_\delta$, but not the other way round.

Then we looked at termination equivalence $\text{te}_C$ for various classes of programs $C$. We observe that from 3.1.1 if $C_1$ is a class and $C_1 \subseteq C_2$, then $\equiv_{\text{te}C_2}$ implies $\equiv_{\text{te}C_1}$.

The classes $C$ we will be looking at in this section are

1. basic;
2. loop free;
3. fap;
4. fapS;
5. fapC;
6. eds;
7. $G_{bs}$;
8. $G_{fl}$;
9. $G_0$.

Since an item in this list is often included in an item further down the list, we know about several implications between the $\text{te}$ equivalences these generate.

We looked at an equivalence which involved computable functions being defined nowhere, $nd$-equivalence. We observed that, for any class $C$ containing loop free,

$$\equiv_{ndC} = \equiv_{\text{te} \text{ loop free}} = \equiv_\delta .$$

We looked at identifiability equivalence $id$, and showed that $\text{te}$-equivalence for loop free programs implies $id$-equivalence.

We looked at a distinguishability equivalence $dt$, and claimed it was the strongest $\text{te}$ (i.e. $\text{te}_{G_0}$) equivalence that we would be interested in.

We also looked at specification equivalence which was concerned with the program specifications true in a structure; we showed that $sp$ equivalence implies $\text{te}$ equivalence for most classes $C$ of definable functions.
We looked at type based equivalences, called $\Gamma_0$, $\Gamma_{fl}$, $\Gamma_{bs}$ and $\Gamma_{lf}$; showed how each of them could be represented as a $te$-equivalence if definable functions were generalised to the not-necessarily-computable case. We are aware of the inclusions $\Gamma_{lf} \subseteq \Gamma_{bs} \subseteq \Gamma_{fl} \subseteq \Gamma_0$, and the implications between the equivalences they give us.

We will, of course, be discussing the relationship all of these have to the most important equivalences of all: isomorphism ($\equiv$) and the trivial equivalence.

All of these relationships and the results of this section are summarised in Fig. 5.1.1.

The results are divided according to the nature of the difference between the programming formalisms, and are presented in separate subsections.
Implications along dotted lines are not known to be strict.

\[
\begin{align*}
\Gamma_0 &= t_{e_{G0}} = d_{t_{eds}} = d_{t_{loop\ free}} \\
\Gamma_{\infty} &= t_{e_{G\infty}} \\
\Gamma_{bs} &= t_{e_{G_{bs}}} \\
te_{eds} &= t_{e_{fapCS}} \\
te_{fapC} &= s_{p_{fapC}} \\
te_{fapS} &= s_{p_{fapS}} \\
te_{fap} &= s_{p_{fap}} \\
\Gamma_3 &= t_{e_{loop\ free}} = n_{d_{eds}} = \equiv_b = \text{etc.} \\
id_{G_0} &= id_{loop\ free} \\
te_{basic} &= \text{trivial}
\end{align*}
\]

- Congruence for all structures
- Congruence for computable structures

Add control
Add storage
Add logical equivalence
Add operation symbols

Figure 5.1.1. Relationships between different equivalences.
5.1.1 Adding iteration

Proposition 5.1.1. The following pairs of sets of observations are not similar.

(1) \( tef_{\text{ap}} \) and \( tef_{\text{loop free}} \).

Proof. (1) We use the fact that two structures are \( tef_{\text{loop free}} \)-equivalent iff the same quantifier-free formulas are satisfiable on them. (3.2.5). Let \( L \) be the language of arithmetic with constant zero and successor function; let \( \mathfrak{M} \) be the standard structure of \( L \) and \( \mathfrak{N} \) be any non-standard structure satisfying \( \text{Th}_L(\mathfrak{M}) \); we are told of the existence of \( \mathfrak{N} \) by compactness. We know then that \( \mathfrak{M} \equiv_{\text{teq loop free}} \mathfrak{N} \).

However, these are not equivalent by \( tef_{\text{ap}} \). We can define the fap \( f \) of one variable \( x \) given by the flowchart in Fig. 5.1.2. The relation is plainly total on \( \mathfrak{M} \), but is undefined somewhere on \( \mathfrak{N} \).

![Flowchart for program f](image)

**Figure 5.1.2.** The program \( f \).

We can also prove the result by exhibiting two computable structures.
Let $L$ be the language with one sort and one unary operation symbol, successor, and for each number $n \in \mathbb{N}$, let $M_n$ be the structure of $L$ whose carrier is the set $\{0, \ldots, n-1\}$; interpret successor by for each $x \in \{0, \ldots, n-1\}$, $\text{succ}^M_n(x) = x + 1 \pmod{n}$.

Let $M_\omega$ be the reduct of the standard structure of arithmetic to $L$.

Let $M$ be the disjoint union $M = \bigcup_{n \in \omega} M_n$, and $\mathfrak{N}$ the union $\mathfrak{N} = M_\omega \cup \bigcup_{n \in \omega} M_n$; then both $M$ and $\mathfrak{N}$ are computable by 2.7.5.

Clearly $M$ and $\mathfrak{N}$ are not $te_{lap}$ equivalent, as the program $g$ shown in Fig. 5.1.3 is total on $M$ but not on $\mathfrak{N}$.

![Figure 5.1.3. The program $g$.](image)

We will show that $M$ and $\mathfrak{N}$ are $te_{loop\_free}$-equivalent. Let $P$ be a loop free scheme

$$P = \{ E_1(x) \land \ldots \land E_k(x) \rightarrow t_p(x) \mid p \in P \}$$
over the variables \((x_1, \ldots, x_m)\) and suppose that \(P^M\) is non-total, undefined at a tuple \(n = (n_1, \ldots, n_m)\) in \(N\). We will suppose w.l.o.g. that for some \(k, 1 \leq k \leq m\), the elements \(n_1, \ldots, n_k\) are the numbers \(p_1, \ldots, p_k\) respectively drawn from \(M_\infty\) in \(N\), and the elements \(n_{k+1}, \ldots, n_m\) are not from \(M_\infty\). We will find a number \(t\) such that if we let the elements \(m_1, \ldots, m_k\) of \(M\) be the numbers \(p_1, \ldots, p_k\) from \(M_\infty\), then \(P^M\) is undefined at the tuple \((m_1, \ldots, m_k, n_{k+1}, \ldots, n_m)\).

Each of the relations \(E_{js}(x)\) is an equality or inequality; we need to choose \(t\) sufficiently large that, over the structures elements in question, the same inequalities are true. Let \(D\) be the set of numbers \(d\) in \(M_\infty\) for which there is a term \(t(x)\) in the scheme \(P\) with \(t^M(n) = d\); since \(P\) is a loop free scheme, we know that \(D\) is finite. Now choose

\[
t = \max(D \cup \{c \mid n_j \text{ is drawn from } M_c \text{ for } k < j \leq m \}) + 1.
\]
5.1.2 Adding control capability

**Proposition 5.1.2.** The following pairs of sets of observations are not similar.

1. \( \text{te}_{\text{fapC}} \) and \( \text{te}_{\text{fap}} \).

**Proof.** (1) This case is interesting. We are looking at two classes of computable functions, \( \text{fapC} \) and \( \text{fap} \). These are known to be different, as not all structures have sufficient complexity to be able to simulate the natural numbers. Structures on which these classes are different appear in e.g. [Friedman, 1971]. A very general result was proved by Kfoury in [Kfoury, 1983, Proposition 4.8(b)] extending the principles upon which such examples are based; first, however, the necessary definitions.

**Definition 5.1.1.** Let \( L \) be a many sorted language. Let \( \tau \) be a sort type \( \tau = (m; i_1, \ldots, i_m) \) and let \( x \) be a \( \tau \)-ary tuple of variables; the set \( L(x) \) is the set of all terms over \( L \) on \( x \). Let \( d \in \mathbb{N} \); the set \( L_d(x) \) is the set of all terms \( t(x) \) in \( L(x) \) which have pebble complexity \( \leq d \). Given a structure \( \mathcal{M} \) and a \( \tau \)-ary tuple \( m \) from \( \mathcal{M} \), the set of interpretations of terms in \( L_d(x) \) with \( x \) interpreted at \( m \) is denoted \( L_d^\mathcal{M}(m) \).

A structure \( \mathcal{M} \) is *locally finite* if for each \( \tau \) and each \( \tau \)-ary \( m \), \( \langle m \rangle \) is finite.

A structure \( \mathcal{M} \) is *uniformly locally finite* if for each \( \tau \) there exists a number \( p_\tau \), such that for every \( \tau \)-ary \( m \), \( |\langle m \rangle| \leq p_\tau \).

A structure \( \mathcal{M} \) is *locally finite w.r.t. bounded space* if, for each \( \tau \), \( 1 \); and each \( \tau \)-ary \( m \), \( L_1^\mathcal{M}(m) \) is finite.

A structure \( \mathcal{M} \) is *uniformly locally finite w.r.t. bounded space* if for each \( \tau \) and each \( d \), there exists a number \( p_{d, \tau} \) such that for every \( \tau \)-ary \( m \), \( |L_d^\mathcal{M}(m)| \leq p_{d, \tau} \).
Proposition 5.1.3. (Kfoury). Let \( L \) be a language and \( \mathcal{M} \) a structure of \( L \) with following properties:

1. \( \mathcal{M} \) is locally finite w.r.t. bounded space;
2. \( \mathcal{M} \) is not uniformly locally finite w.r.t. bounded space;

then there is a fap\(\text{C} \) not equivalent to any fap on \( \mathcal{M} \).

This means that there are a good deal of well understood structures separating fap\(\text{C} \) and fap. However, whilst the classes of functions are different, the termination equivalences they generate are identical on locally finite w.r.t b.s. structures.

Proposition 5.1.4. Let \( \mathcal{M} \) and \( \mathcal{N} \) be \( L \)-structures locally finite w.r.t. bounded space.

Then \( \mathcal{M} \equiv_{\text{fap} \text{C}} \mathcal{N} \iff \mathcal{M} \equiv_{\text{fap}} \mathcal{N} \).

Proof. Clearly \( \mathcal{M} \equiv_{\text{fap} \text{C}} \mathcal{N} \Rightarrow \mathcal{M} \equiv_{\text{fap}} \mathcal{N} \).

Now suppose that \( \mathcal{M} \equiv_{\text{fap}} \mathcal{N} \) and that a fap\(\text{C} \) \( P \) of argument sort \( \tau = (m; i_1, ... , i_m) \) is undefined at \( m = (m_1, ... , m_m) \) on \( \mathcal{M} \); we aim to show that \( P \) is undefined somewhere on \( \mathcal{N} \).

Let \( d \) be the number of algebraic program and input variables in \( P \); let \( L_0 \) be the finite sublanguage of \( L \) appearing in \( P \) (plus equality and non-algebraic operations); let \( p \) be the size of the set \( L_0^m(m) \).

We fix on a finite set of terms \( T = \{ t_1, ... , t_y \} \) such that

1. every \( m \in L_{0d} \mathcal{M}(m) \) is the interpretation \( \mathcal{M}(m) \) of a term \( t \in T \);
2. every \( t \in T \) has pebble complexity \( \leq d \);
3. every symbol \( x \in X \) and every constant symbol \( c \in L_0 \) is in \( T \).

For each operation and relation symbol \( f_j \) of type \( \tau_j = (m; i_1, ... , i_m, i) \) in \( L_0 \), we let the set \( S_{f_j} \) of relations be

\[
S_{f_j} = \{ t = f_j(t_1, ... , t_m) \mid t, t_1, ... , t_m \in T \text{ and } t_{mi_j} = f_j(t_{mi_1}, ... , t_{mi_m}) \}.
\]
We will show the following lemma.

**Lemma.** If a structure $\mathcal{M}$ satisfies each of the $S_{ij}(x)$ at $n$, then for every boolean valued term $\psi$ over $L_0$ of pebble complexity $\leq d$,

$$\psi_{\mathcal{M}}(m) = \psi_{\mathcal{N}}(n).$$

**Proof of lemma.** We will show by induction that for every pair $t_1(x)$ and $t_2(x)$ of terms over $L_0$ of pebble complexity $\leq d$,

$$t_1^{\mathcal{M}}(m) = t_2^{\mathcal{M}}(m) \iff t_1^{\mathcal{N}}(n) = t_2^{\mathcal{N}}(n);$$

the fundamental observation is that the set of terms with pebble complexity $\leq d$ is closed under subterms. It will suffice to show (*) for terms $t_2$ which are drawn from our specially chosen set $T$. For, let $t_1$ and $t_2$ be terms of pebble complexity $\leq d$; then there is a $t \in T$ for which $t_1^{\mathcal{M}}(m) = t^{\mathcal{M}}(m)$.

Suppose (*) holds in the restricted case; then

$$t_1^{\mathcal{M}}(m) = t^{\mathcal{M}}(m) \iff t_1^{\mathcal{N}}(n) = t^{\mathcal{N}}(n),$$

so

$$t_1^{\mathcal{N}}(n) = t^{\mathcal{N}}(n),$$

and

$$t_2^{\mathcal{M}}(m) = t^{\mathcal{M}}(m) \iff t_2^{\mathcal{N}}(n) = t^{\mathcal{N}}(n),$$

so

$$t_1^{\mathcal{M}}(m) = t_2^{\mathcal{M}}(m) \iff t_1^{\mathcal{N}}(n) = t_2^{\mathcal{N}}(n).$$

So, the base cases are where $t_1$ is a variable $x_j \in x$, and where $t_1$ is a constant symbol. In the case that $x_j$ is a variable, one of the relations $\text{true} \equiv (x_j = t_2)$ or $\text{false} \equiv (x_j = t_2)$ is in $S_n$, so since both $\mathcal{M}$ and $\mathcal{N}$ satisfy $S_n$, (*) will hold. If $t_1$ is a constant symbol $c$, one of the relations $\text{true} \equiv (c = t_2)$ or $\text{false} \equiv (c = t_2)$ will be in $S_n$, so (*) will hold.
Now inductively, suppose that $t_1$ is $f(s_1, \ldots, s_m)$ for terms $s_1, \ldots, s_m$; as we observed, each $s_j$ has pebble complexity $\leq d$. Then there are terms $t_{s_1}, \ldots, t_{s_m}, t$ for which (by the inductive hypothesis) both $t_{s_j}(m) = s_j(m)$ and $t_{s_j}(n) = s_j(n)$ and for which one of the relations

$$\text{true} \equiv (t_1 = t_2) \quad \text{or} \quad \text{false} \equiv (t_1 = t_2)$$

is in $S^\omega$;

so

$$t_{s_j}(m) = t_{s_j}(m) \iff t_{s_j}(n) = t_{s_j}(n).$$

Also, since

$$t(m) = f(t_{s_1}(m), \ldots, t_{s_m}(m)),$$

we have

$$t = f(t_{s_1}, \ldots, t_{s_m}) \in S_f,$$

so

$$t(n) = f(t_{s_1}(n), \ldots, t_{s_m}(n)).$$

Therefore

$$t_{s_j}(m) = t_{s_j}(m)$$

$$\iff t_{s_j}(n) = t_{s_j}(n)$$

$$\iff f(t_{s_1}(n), \ldots, t_{s_m}(n)) = t_2(n)$$

$$\iff f(s_1(n), \ldots, s_m(n)) = t_2(n)$$

$$\iff t_1(n) = t_2(n). \quad \bullet$$

We note that

(1) each $S_{f_j}$ is finite;

(2) there are finitely many such $f_j$;

(3) any $L$-structure $\mathcal{N}$ and $\tau$-tuple $n \in \mathcal{N}$ satisfying every relation in every $S_{f_j}$ will have $P^\mathcal{N}(n)$ undefined.

We can therefore write a loop free program $Q$ which tests all of the relations in the $S_{f_j}s$, and is defined iff they are all true. We will have $Q^\mathcal{M}$ defined at $m$. 

Chapter 5 - Separation and congruence - Separation - Adding control capability
But since $\mathcal{M} \equiv_{\text{refap}} \mathcal{N}$ and therefore $\mathcal{M} \equiv_{\text{ndloop free}} \mathcal{N}$ by 3.2.4 so $Q^n$ is defined somewhere at $n$, say. Hence $P^n(n)$ is undefined.

Therefore $\mathcal{M} \equiv_{\text{refapC}} \mathcal{N}$.

The conclusion of 5.1.4 tells us that we cannot use locally finite structures to separate the $te$ equivalences for fap and fapC.

Recall the structure $T_\omega$ from §4.3. $T_\omega$ is a rare example of a non-locally finite structure for which $\text{FAP}(T_\omega) \neq \text{FAPC}(T_\omega)$. In fact, $T_\omega$ is generated by one element, associated with the symbol $\lambda$. We will use $T_\omega$ to construct two structures which are $te$-equivalent for fap, but not for fapC.

**Proof of 5.1.2.** Let $L$ be the language containing one algebraic sort, on which there are two unary function symbols, $f$ and $g$. Let $L'$ be $L$, extended with two unary relation symbols, $\theta$ and $\phi$. Let $T$ be the reduct of $T_\omega$ to $L$; that is $T_\omega$ but without the constant symbol $\lambda$.

We will form structures of $L'$ by expanding $T$ to $L'$, i.e. interpreting the relation $\theta$ and $\phi$.

Let $\{t_1(\lambda),\ldots, t_i(\lambda), \ldots \}$ be the (r.e.) set of terms in $L$ over $\lambda$, ordered by a suitable Gddel ordering.

Now let $p \in \mathbb{N}$; we are going to define an expansion $\mathcal{M}_p$ of $T$ as follows.

We define

$$\theta^p(x) = \text{true} \iff x = \lambda$$

and

$$\phi^p(x) = \text{true} \iff x \in \{t_0, \ldots, t_p\}.$$

The structure $\mathcal{M}_p$ is shown in Fig. 5.1.4.
Once again, we define the structure $M_\infty$ in the obvious way, where $\phi$ is interpreted as

$$\phi_{M_\infty}(x) \equiv \text{true} \iff x \in \{t_0, \ldots, t_i, \ldots\}.$$ 

Let $\mathcal{M}$ be the union 

$$\mathcal{M} = \bigcup \{M_p \mid 0 \leq p < \infty\}$$

and $\mathcal{N}$ the union 

$$\mathcal{N} = M_\infty \cup \bigcup \{M_p \mid 0 \leq p < \infty\}.$$ 

These structures are shown in Fig. 5.1.5. We observe that they are both computable by 2.7.5; that is, there exists a uniform effective presentation for the collections $\{M_p \mid 0 \leq p < \infty\}$ and $\{M_p \mid 0 \leq p < \infty\}$ from the fact that $T_\omega$ is computable and from the simple nature of the interpretation of $\phi$.

We will show that $\mathcal{M} \not\cong_{\text{refap}} \mathcal{N}$ but that $\mathcal{M} \cong_{\text{refap}} \mathcal{N}$. 

\begin{figure}
\centering
\includegraphics[width=\textwidth]{structure_mp.png}
\caption{The structure $M_p$.}
\end{figure}
Let $Q$ be the scheme with one input variable $x$, say, given by

$$Q = \{ \theta(x) \land \neg \phi(t_1(x)) \} \quad \cup \quad \{ \theta(x) \land \phi(t_1(x)) \land ... \land \phi(t_k(x)) \land \neg \phi(t_{k+1}(x)) \mid 1 \leq k \};$$

then $Q$ is an eds since the set of clauses is r.e. Since $T_\omega$ is structural, $Q^{T_\omega}$ is fapC-computable by 2.8.4.

Clearly $Q^{\mathcal{M}}$ is total since for every $x$ satisfying $\theta(x) = \text{true}$, there is an $i$ for which $\phi(t_i(x)) = \text{false}$.

But $Q^{\mathcal{M}}$ is not total; it is undefined at $\lambda$ on $\mathcal{M}_{\omega}$. Therefore $\mathcal{M} \not\equiv_{\text{fapC}} \mathcal{N}$.

Now we aim to show that $\mathcal{M} \equiv_{\text{tefap}} \mathcal{N}$. We do this by showing that whenever $R$ is a fap and is undefined somewhere on $\mathcal{N}$, there is somewhere on $\mathcal{M}$ it is undefined.

This is done in several stages:

1. show that if $\mathcal{M} \not\equiv_{\text{tefap}} \mathcal{N}$ then there is a fap $R_1$ with $R_1^{\mathcal{M}}$ total but $R_1^{\mathcal{N}}(r)$ for a tuple $r$ of $\lambda$s on $\mathcal{N}$ drawn from distinct $\mathcal{M}_{p}$;

2. show that if such an $R_1$ exists then there is a fap $R_2$ with one input variable for which $R_2^{\mathcal{M}}$ is total but $R_2^{\mathcal{N}}$ is undefined at $\lambda$ in $\mathcal{M}_{\omega}$;
(1) Since $\mathcal{M}$ is a substructure of $\mathcal{N}$, any fap total on $\mathcal{N}$ will be total on $\mathcal{M}$; so if $\mathcal{M} \neq_{\text{fap}} \mathcal{N}$ then there is a fap $R$ which fails to terminate at some tuple $n = (s_1(\lambda), \ldots, s_m(\lambda))$ in $\mathcal{N}$, but $R^\mathcal{M}$ is total. We can write a fap $R_1$ which, given a tuple $x$, applies terms $s_1, \ldots, s_m$ to input variables $x_1, \ldots, x_m$ respectively, and then executes $R$. If $R_1$ fails to terminate for some tuple $l = (l_1, \ldots, l_m)$ of $\lambda$s in $\mathcal{M}$, so $R$ fails to terminate at the tuple $(s_1(l_1), \ldots, s_m(l_m))$ in $\mathcal{M}$.

(2) We write a fap $R_2$ that mimics the effect of $R_1$ by using values generated from the single input variable as surrogates for the values generated from different input variables in $R_1$; equality tests and $\phi$ tests in $R_1$ will then be replaced with tests in $R_2$ which mimic their effect. Suppose that the elements of the tuple $r = (r_1, \ldots, r_m)$ are drawn from substructures $\mathcal{M}_p$ for $p \in \{p_1, \ldots, p_k\}$, say. For every program and input variable $v$ in $R_1$ we have a pair $(v_a, v_b)$ of program variables in $R_2$. The idea is that we need to keep track of the 'origin' of each of the values in the $R_1$ variables by tagging them with a value and keeping a copy of this tag with the value at all times.

So, we fix on the set of terms $t_1(\lambda), \ldots, t_k(\lambda)$ as the tags, reserving $t_1(\lambda)$ to be the tag for $\mathcal{M}_1$. The first instructions of $R_2$, given the input variable $I$, makes a copy of its contents into every variable $v_a$ corresponding to an $R_1$ input register, and loads $v_b$ with the tag $t_j(\lambda)$ if the corresponding value in $r$ was drawn from $\mathcal{M}_p$.

The remaining instructions of $R_2$ are obtained by translating the corresponding $R_1$ instruction in the following way.

An assignment instruction $v := f(w)$ translates into the instructions $(v_a := f(w_a); v_b := w_b)$.

An equality test $(v = w)$ translates to the test $(v_a = w_a$ and $v_b = w_b)$;

A test $\theta(v)$ translates to the test $\theta(v_a)$.
The only difficulty are the $\phi$ tests.

We translate the test $\phi(v)$ into the loop free program

If $v_b = t_1$ then true
else if $v_b = t_2$ then
if $v_a = t_1$ then true
else if $v_a = t_2$ then true
else ...
else if $v_a = t_{p2}$ then true
else false
else if $v_b = t_3$ then
if $v_a = t_1$ then true
else if $v_a = t_2$ then true
else ...
else if $v_a = t_{p3}$ then true
else false
...
else if $v_b = t_k$ then
if $v_a = t_1$ then true
else if $v_a = t_2$ then true
else ...
else if $v_a = t_{pk}$ then true
else false.

In this way, the tags allow us to mimic the $R_1$ computation completely. So $R_2^M$ is total but $R_2^M$ is undefined at $\lambda$ on $M_{\infty}$.

(3) Let $R_2$ be a fap over $L'$ with one input argument which fails to terminate at $\lambda$ on $M_{\infty}$. We can translate $R_2$ into a fap $R^*$ over $L \cup \{\lambda\}$ by replacing every $\phi$ test with a true result (as will be the case on $M_{\infty}$) and every $\theta(x)$ test with a test $x = \lambda$. Then by 4.3.5 we have that the sequence of program states in the computation of $R^*$ at $\lambda$ in $T_\infty$ is periodic; so in the course of the computation on $T_\infty$, and hence the same is true for $M_{\infty}$, the set of domain elements appearing in any program variable is finite. Therefore there is a $q$ such that no term $t_i$ ever
appears in a program variable if i>q. Thus R fails to terminate at λ on \( M_p \) for every \( p \geq q \), and \( R \) is no total. •

The problem as to whether or not termination equivalences for \( \text{fapCS} \) and \( \text{fapS} \) are the same is open. The problem we are faced with is deciding whether or not there exists a non-locally finite structure \( M \) for which \( \text{FAPS}(M) \neq \text{FAPCS}(M) \); we know of examples which are locally finite with that property, but as we have seen, these are not suitable for showing that the equivalences are different.

Whilst that problem remains open, we can show that if such a structure does exist, then the termination equivalences are different.

**Proposition 5.1.5.** Suppose that \( L \) is a finite language \( L \) containing no operation symbols of algebraic value sort which have no algebraic argument sort. Suppose that there is a finitely generated infinite \( L \)-structure \( M = \langle m \rangle \) for which \( \text{FAPCS}(M_m) \neq \text{FAPS}(M_m) \). Then \( t_{\text{fapCS}} \neq t_{\text{fapS}} \).

**Proof.** Suppose that \( m \) is a \( \tau \)-ary tuple in \( M \) and that \( M = \langle m \rangle \). We will use a construction and proof similar to the last example. Let \( L' \) be the language obtained from \( L \) by introducing a new \( \tau \)-ary relation symbol, \( \theta \) say. Let \( M' \) be the \( L' \)-structure obtained from \( M \) by interpreting

\[
\theta_{M'}(x) \equiv \text{true} \iff x = m.
\]

Let \( L'' \) be the language \( L' \) extended with a unary relation symbol \( \phi \). We will consider expansions \( M_p \) of the structure \( M' \) to \( L'' \).

Let \( \{ t_0(x), \ldots, t_i(x), \ldots \} \) be the (r.e.) set of terms in \( L \) over \( m \), ordered by a suitable Gödel ordering.

Pick \( p \in \mathbb{N} \); we define \( M_p \) by interpreting \( \phi_{M_p}(m) \equiv \text{true} \iff m \in \{ t_0(M_p)(m), \ldots, t_p(M_p)(m) \} \).

The structure \( M_p \) is shown in Fig. 5.1.6.
Once again, we define the structure $M_p$ in the obvious way, where $\phi$ is interpreted as

$$\phi M^\infty(x) \equiv \text{true} \iff x \in \{t_0, \ldots, t_i, \ldots\}.$$

Let $R$ be the union

$$R = \bigcup \{ M_p \mid 0 \leq p < \infty \}$$

and $R$ the union

$$\mathcal{R} = M_{\infty} \cup \bigcup \{ M_p \mid 0 \leq p \leq \infty \}.$$
We will show that:

1. \( \mathcal{R} \not\equiv^{\text{efapCS}} \mathcal{N} \);
2. if \( \mathcal{R} \not\equiv^{\text{efapS}} \mathcal{N} \) then \( \text{FAPCS}(\mathcal{M}_m) = \text{FAPS}(\mathcal{M}_m) \).

1. Let \( Q \) be the scheme with a \( \tau \)-ary tuple of input variables \( x \), say, given by

\[
Q = \{ \emptyset(x) \wedge -\phi(t_1(x)) \} \\
\cup \{ \emptyset(x) \wedge \phi(t_1(x)) \wedge ... \wedge \phi(t_k(x)) \wedge -\phi(t_{k+1}(x)) \mid 1 \leq k \};
\]

then \( Q \) is an eds since the set of clauses is r.e. Clearly \( Q^\mathcal{M} \) is total since for every \( x \) satisfying \( \emptyset(x) \equiv \text{true} \), there is an \( i \) for which \( \phi(t_i(x)) \equiv \text{false} \).

But \( Q^\mathcal{N} \) is not total; it is undefined at \( m \) on \( \mathcal{M}_c \). Therefore \( \mathcal{M} \not\equiv^{\text{efapCS}} \mathcal{N} \).

2. Suppose that \( \mathcal{R} \not\equiv^{\text{efapS}} \mathcal{N} \); then there exists a fapS \( P \) say, and a point \( n \in \mathcal{N} \) for which \( P^\mathcal{R}(n) \) is undefined but that \( P^\mathcal{R} \) is total.

We know that at least one of the elements from \( n \) is drawn from \( \mathcal{M}_c \); we also know that during the course of the execution of \( P \) at \( n \), there is one assignment operation in \( P \) for which there are assignments of infinitely many distinct values \( n_0, \ldots, n_r, \ldots \) from \( \mathcal{M}_c \); otherwise we could interpret the appropriate elements from \( n \) on \( \mathcal{M}_p \) for sufficiently large \( p \) and obtain a point \( r \in \mathcal{R} \) for which \( P^\mathcal{R}(r) \) is undefined. It is these elements that we are going to use to construct a counter in \( \mathcal{M} \).

The problems we need to solve are

1. translating \( P \) into a fapS on \( \mathcal{M} \);
2. finding a means of implementing a successor function on \( \mathcal{M} \) without restricting the use of the stack by calling programs.
Suppose that the elements of $\mathbf{n}$ are drawn from $\mathcal{M}_j$ for $j \in \{\infty, \mathbf{p}_1, \ldots, \mathbf{p}_k\}$. Let $\mathcal{P}$ be the disjoint union of $k+1$ copies of $\mathcal{M}'$; let $\mathbf{p}$ be the tuple of elements of $\mathcal{P}$ which is the tuple of the $k+1$ copies of $\mathbf{m}$. We will show the following:

1. there is a fap$\mathcal{S}$ $\mathcal{Q}$ which, given $\mathbf{p}$, will not terminate and for which there is one assignment making assignments of infinitely many distinct values from $\mathcal{P}$;
2. there is a fap$\mathcal{S}$ $\mathcal{R}$ operating on $\mathcal{M}$ itself with that property;
3. $\text{FAPS}(\mathcal{M}) = \text{FAPCS}(\mathcal{M})$.

1. The program $\mathcal{Q}$ is a modified version of $\mathcal{P}$; the idea is that each $\mathcal{M}_p$ is mimicked by a copy of $\mathcal{M}'$. Since there are only ever finitely many or co-finitely many values in each $\mathcal{M}_p$ for which $\phi$ is true, if the program keeps track of the origin of the value in each of its registers using a suitable set of flags, any test for $\phi$ in $\mathcal{P}$ can be replaced by a straight-line program testing that value against a known finite set of values, as in the previous example. Initially, the registers are set up from the given variables $\mathbf{p}$ in order to generate $\mathbf{n}$; then the execution follows the modified version of $\mathcal{P}$.

2. This program is obtained from the one before. Each of the copies of $\mathcal{M}'$ is mimicked in the single copy of $\mathcal{M}$. Once again, the new program needs to keep track of the origin of each of its variables using a finite set of flags. The reason for this is that there might be tests of relations drawn from different copies of $\mathcal{M}'$ in $\mathcal{Q}$ and the outcome of the analogous tests in $\mathcal{R}$ will rely upon this information. Every test of this kind in $\mathcal{Q}$ can be replaced with a straight-line program in $\mathcal{R}$. Instead of being a program with input variables $\mathbf{p}$, it is a nullary program which starts off by generating all of its data from the constants $\mathbf{m}$ in $\mathcal{M}_m$.

3. We now need to use the program $\mathcal{R}$ to implement a counter. The result will be proved if we can exhibit an infinite set of values $m_1, \ldots, m_i, \ldots$ and a fap$\mathcal{S}$ of one variable which, given $m_i$, generates $m_{i+1}$ and leaves the stack as it found it.
We modify the program R to obtain $R^*$ in the following way; first it places a marker on the stack and replaces every test for an empty stack with a test for that marker. Every restore instruction is replaced with one that first tests to see whether or not the stack is empty, so that the effect of popping an empty stack is mimicked.

We need to write two programs $T$ and $T_{\text{next}}$, each with one input variable $t$ say. The effect of $T$ is to run through the execution of our program $R^*$ until this particular assignment statement assigns the value in the input variable $t$ and then terminates. No output is given; the only side-effect is to possibly leave elements on the stack and in program variables so that the computation can continue. The effect of $T_{\text{next}}$ is to resume the execution of $R^*$ at that point and to proceed until the next assignment is made at that particular instruction. The value of the following assignment is output.

Now we can actually implement the successor function. The main problem we have to overcome is that the sequence of assignments made at that instruction will in general contain repeats; so $T_{\text{next}}$ will not do as a successor.

The flow-diagram for the successor function is shown in Fig. 5.1.8.

In other words, repeatedly apply $\text{next}$ until a value occurs which did not occur before $x$. •
Figure 5.1.8. The successor function in the proof of 5.1.5.
5.1.3 Adding storage space

**Proposition 5.1.6.** The following pairs of sets of observations are not similar.

1. $\Gamma_{fl}$ and $\Gamma_{bs}$;
2. $te_{fapS}$ and $te_{fap}$;
3. $te_{fapCS}$ and $te_{fapC}$.

**Proof.** This cases require use of an argument based on a version of the pebble game.

Our aim will be to find a sequence of finite binary trees $T = (t_1, t_2, \ldots, t_i, \ldots)$ for which $t_i$ has a space requirement in the extended game of at least $i$, for each $i$. We will do this bit first, and then apply it to the problem later.

Just for reference, we will define the kind of trees we are looking at. A *binary tree* is an element of the smallest set $S$ with the following closure properties:

1. the null tree, $\bullet$, is in $S$,
2. whenever $t_1$ and $t_2$ are in $S$, then $join(t_1, t_2)$ is in $S$.

Examples of trees are shown in Fig. 5.1.9.

For $n \in \mathbb{N}$, we define the *complete binary tree of order $n$* as follows:

1. The null tree, $\bullet$, is the complete tree of order 0,
2. if $t$ is the complete tree of order $n$, then $join(t, t)$ is the complete tree of order $n+1$.

In pictures, the complete trees of small orders are shown in Fig. 5.1.10.
We will write $B_n$ to represent the complete tree of order $n$. The first fact is a particular instance of 2.8.3 and concerns these complete trees.

**Proposition 5.1.7.** The complete binary tree of order $n$ has a space requirement (in the non-extended game) of at least $n$.

**Proof.** By induction on $n$. Result is plainly true for $n = 0$. Otherwise, assume result true for some $n$. It suffices to show that the space requirement for $B_{n+1}$ is greater than that for $B_n$. Let $(C_0, \ldots, C_k)$ be a complete calculation on $B_{n+1}$ using $t+1$ pebbles; we will show there exists a complete calculation on $B_n$ using only $t$ pebbles. We can assume w.l.o.g. that no configuration in $C_0, \ldots, C_{k-1}$ has a pebble on the root of $B_{n+1}$, and that no configuration appears more than once in the list. Now consider Fig. 5.1.11.

![Figure 5.1.11. The complete tree $B_{n+1}$.](image)

Since none of the $C_i$, $i>0$ are empty, there exist $a, b \in \{1, \ldots, k-1\}$ with $a \neq b$ such that

1. $C_a$ is the last configuration in the calculation for which there are no pebbles on subtree $C$.
(2) $C_b$ is the last configuration in the calculation for which there are no pebbles on subtree $D$;
(3) $a < b$ (w.l.o.g.).

Then if we take the subsets of the configurations $C_b, ..., C_{k-1}$ on the subtree $D$ only, we obtain a complete calculation for $D \equiv B_n$ using not more than $t$ pebbles (because there is always at least one on the subtree $C$).

However, our main interest is in the extended pebble game, and our intent is essentially to prove the same result about the extended game. That is, there exist trees $t_1, t_2, ..., t_k, ...$ such that the space requirement for $t_i$ exceeds $i$. It will not work putting $t_i = B_j$, as all of the $B_i$ have space requirement of 3 in the extended game. Our approach will be to annotate the leaves of the $B_i$ with different subtrees so that the copying rule (3) no longer applies. It is important to us that the set $T = \{t_1, ..., t_i, ...\}$ is recursive, as we will need to enumerate those trees with a program.

**Definition 5.1.2.** The *depth* of a binary tree is defined as follows.

1. The depth of $\bullet$ is 0,
2. the depth of $\text{join}(t_1, t_2)$ is $\max(d_1, d_2) + 1$ where $d_i$ is the depth of $t_i$ $(i=1,2)$.

In order that there are enough trees to do the annotations we need the following.

**Proposition 5.1.8.** For each $n \in \mathbb{N}$, there exists an $m_n$ such that the number of distinct binary trees of depth $m_n$ exceeds $2^n$.

**Proof.** The complete tree $B_n$ has $2^{n-1}$ subtrees of the form $\text{join}(\bullet, \bullet)$. By replacing any number except all of them with $\bullet$, we get $2^{2^{n-1}} - 1$ distinct trees of the same depth as $B_n$, i.e. $n$. So choose $m_n \geq \log_2(n) + 1$.

It is clear that there is a program which, given $n$ and a (code for a) tree $t$, will decide if $t$ is one of the first $2^n$ such trees of depth $m_n$ for some fixed linear ordering on the trees.
**Definition 5.1.3.** A modified complete tree of order \( n \) is the tree \( B_n \), but with each leaf substituted by a different tree of depth \( m_n \). This is always possible from the definition of \( m_n \).

There is clearly a program which, for some fixed set \( T = \{ t_1, \ldots, t_i, \ldots \} \) of modified complete binary trees \( t_i \) of order \( i \), will, given an \( n \) and a tree \( t \), decide if \( t = t_i \).

**Proposition 5.1.9.** A modified complete tree of order \( n \) has space requirement for the extended pebble game of at least \( n \).

**Proof.** We show that the space requirement of a modified complete tree \( t \) of order \( n \) has space requirement for the extended game at least the space requirement of \( B_n \) for the basic game. Let \( C_1, C_2, \ldots, C_p \) be a complete calculation for \( t \) in the extended game. We obtain a calculation \( D_1, D_2, \ldots, D_p \) for \( B_n \) in the extended game simply by taking \( D_i \) to be the subset of \( C_i \) appearing in \( B_n \) (see Fig. 5.1.12).

![The tree t where n=2, m_n=3](image)

Figure 5.1.12. A calculation.

We now show that \( D_1, D_2, \ldots, D_p \) is a calculation in the basic game to complete the proof. The only problem is if rule (3) is used to place a pebble on a node in the upper, i.e. non-shaded region. However, we can now see that this is impossible; since all of the \( B_n \) annotations are of the same depth, a pebble could only be copied to another location at the same depth in \( B_n \). But
this is plainly not possible since no two nodes at the same level has identical subtrees. (It might be the case, as in the example, that is possible to employ rule (3) to add pebbles in the shaded region; however, these will only translate to null moves in the B_n calculation.)

Now we tackle the equivalences problem. Let L be the language containing only one algebraic sort, on which there is one binary function, join, and two unary relation symbols, θ and φ. Let L(x) be the set of terms of L over one variable, x. To each term we may naturally associate a binary tree (by associating • to the term x, etc.); then the number of registers required to evaluate any term in the free structure on {x} given the value of the variable x is the same as the space requirement of the tree we associate to that term in the extended game. The preceding proposition tells us that there is a sequence of terms t_1, ... for which the term t_i requires at least i registers to evaluate in that structure.

Let p ∈ N; we define the structure M_p of L as follows. The carrier set M is the set of all binary trees; we identify the null tree • with the symbol λ.

We interpret the relation θ on M_p as

θ_M(x) = true ⇔ x = λ,

and the relation φ by

φ_M(x) = true ⇔ x ∈ \{t_1, ..., t_p\}.

The structure M_p is shown in Fig. 5.1.13.

As before, we define the structure M_∞ where φ is interpreted to be true on all of the t_i. Now we define the structure M to be the union

M = ∪\{M_p | p ∈ N\};

and Μ to be the union

M = M_∞ ∪ ∪\{M_p | p ∈ N\}.

These structures are shown Fig. 5.1.14.
Once again, we observe that both $\mathcal{M}$ and $\mathcal{N}$ are computable structures by 2.7.5.

We will show that $\mathcal{M} \not\equiv_{f_{tefpaS}} \mathcal{N}$ but that $\mathcal{M} \equiv_{f_{bs}} \mathcal{N}$.

Let $P$ be the scheme

$$P = \{ -\theta(x) \rightarrow \text{true}, \theta(x) \land -\phi(t_1(x)) \rightarrow \text{true} \}$$
U \{ 0(x) \wedge \phi(t_1(x)) \land ... \land \phi(t_k(x)) \land \neg \phi(t_{k+1}(x)) \rightarrow \text{true} \mid 1 \leq k \}.

It is clear that

1. the set of clauses in P is r.e. since the set of terms t_j is recursive;
2. P^M is total;
3. P^N is non-total.

Since both M and N are free (in the sense that for any x \in N, N_x is \omega-rich), we have that there is a fapS Q such that P^M = Q^M and P^N = Q^N. Thus

1. M \not\models_T N;
2. M \not\models_{wcds} N;
3. M \not\models_{wefapS} N.

Now suppose that \Phi is a type-observation of \Gamma_{bs} of sort type \tau. Since M is a substructure of N, it is clear that any type supported by M is also supported by N. So, suppose \Phi is satisfied on N; we will show that it is also satisfied on M. But now this is easy; by the definition of \Gamma_{bs}, there is a number q such that each of the relations in \Phi has pebble complexity less than q. This means that none of the terms t_q, t_{q+1}, ... occur in any of the \phi_i. So for each q \geq q and i,

M_\omega \models \phi_i \iff M_q \models \phi_i.

Let n = (n_1, ..., n_m) be such that n satisfies \Phi in N. Suppose w.l.o.g. that n_1, ..., n_s are drawn from M_\omega for some 0 \leq s \leq m. We can choose q \geq q sufficiently large if necessary so that none of n_{s+1}, ..., n_m are in M_q. Let q_1, ..., q_s be the elements of M_q corresponding to n_1, ..., n_s respectively. Then the tuple (q_1, ..., q_s, n_{s+1}, ..., n_k) satisfies \Phi on M.
5.1.4 Adding operation symbols

Proposition 5.1.10. The following pairs of sets of observations are not similar.

(1) $\Gamma_0$ and $\Gamma_f$.

Proof. (1) Let $L$ be the language with one sort, on which there are two unary relation symbols, $\theta$ and $\phi$, and a countably infinite set $S = \{f_i \mid i \in \mathbb{N} \}$ of unary function symbols $f_i$. We will give two structures $M$ and $N$ and show that $M \models \Gamma_0$, $N \models \Gamma_0$ but $M \not\equiv_{\Gamma_0} N$.

Let $p \in \mathbb{N}$; we define $M_p$ to be the $L$-structure whose carrier $M$ is the set $\mathbb{N} \cup \{\lambda\}$, where $\lambda$ is a distinct single element. We define

$$f_i^M(\lambda) = i \text{ for each } i, \quad \text{and} \quad f_j^M(i) = i \text{ for any } i, j \in \mathbb{N}.$$

Set $\theta^M(x) \equiv \text{true} \iff x = \lambda$ and $\phi^M(x) \equiv \text{true} \iff x \in \mathbb{N}$ and $0 \leq x \leq p$.

Thus $M_p = <\lambda>$; it is shown in Fig. 5.1.15.

![Figure 5.1.15.](image)

In the same way we define the structure $M_{\infty}$ where $\phi^M(x) \equiv \text{true} \iff x \in \mathbb{N}$ and $0 \leq x \leq \infty$.

Now let the structure $M$ be the disjoint union

$$M = \bigcup \{M_p \mid 0 \leq p < \infty \},$$

Chapter 5 – Separation and congruence – Separation – Adding operation symbols 159
and let \( \mathfrak{N} \) be the union

\[
\mathfrak{N} = \bigcup \{ \mathfrak{M}_p \mid 0 \leq p \leq \infty \},
\]

The structures \( \mathfrak{M} \) and \( \mathfrak{N} \) are shown in Fig. 5.1.16.

We will show that \( \mathfrak{M} \not\equiv_{\Gamma_0} \mathfrak{N} \) and that \( \mathfrak{M} \equiv_{\Gamma_{\infty}} \mathfrak{N} \).

It is clear that the structures are not equivalent under \( \Gamma_0 \)-equivalence since the type

\[
\Phi = \{ \phi_0, \phi_1, \ldots, \phi_1, \ldots \}
\]

where \( \phi_0(x) = \theta(x) \) and for each \( i > 0 \), \( \phi_i(x) = \phi(\theta_i(x)) \), is satisfied at \( \lambda \) in \( \mathfrak{M}_{\infty} \) on \( \mathfrak{N} \) but nowhere on \( \mathfrak{M} \).

However, the structures are \( \Gamma_{\infty} \)-equivalent. Since \( \mathfrak{M} \) is a substructure of \( \mathfrak{N} \), it is clear that any types supported by \( \mathfrak{M} \) will also be supported by \( \mathfrak{N} \). Now suppose that for some sort type \( \tau \), a type \( \Phi = \{ \phi_0, \phi_1, \ldots, \phi_1, \ldots \} \) over \( \tau = (m; i_1, \ldots, i_m) \) is satisfied on \( \mathfrak{N} \) but not on \( \mathfrak{M} \).

Let \( \mathbf{n} = (n_1, n_2, \ldots, n_m) \) be a \( \tau \)-tuple from \( \mathfrak{N} \) satisfying each of the \( \phi_i \). Suppose w.l.o.g. that \( \{n_j \mid 1 \leq j \leq p \} \) are drawn from \( \mathfrak{M}_{\infty} \), for some \( p \leq m \). From the definition of \( \Gamma_{\infty} \), there is a finite set of operation symbols \( L' \) such that each of the \( \phi_i \) are over \( L' \). This means that \( \Phi \) must only use
finitely many of the $f_i$. Let $q \in \mathbb{N}$ be such that $f_i \in L' \Rightarrow i < q$. This means that as far as the relations $\phi_i \in \Phi$ are concerned, an $L$-structure $\mathcal{R}$ has $\mathcal{R} \models \phi_i$ iff the reduct $\mathcal{R}_{L'} \models \phi_i$.

We also note that for any $q' \geq q$, $\mathcal{M}_{q'|L} \equiv \mathcal{M}_q L$, and so for each $i$, $\mathcal{M}_{q'} \models \phi_i \iff \mathcal{M}_\infty \models \phi_i$.

Therefore, by choosing $q'$ sufficiently large (to get away from the interpretations of $x_{p+1}, \ldots, x_m$ if necessary) we can simply interpret the variables $x_1, \ldots, x_p$ on the structure $\mathcal{M}_{q'}$ instead of on $\mathcal{M}_\infty$; we interpret the variables $x_1, \ldots, x_p$ as the terms $q_1, \ldots, q_p$ in $\mathcal{M}_{q'}$ corresponding to $n_1, \ldots, n_p$ in $\mathcal{M}_\infty$. Then the tuple $(q_1, \ldots, q_p, n_{p+1}, \ldots, n_k)$ in $\mathcal{M}$ satisfies $\Phi$. •
5.1.5 Introducing non-recursiveness

Proposition 5.1.11. The following pairs of sets of observations are not similar.

(1) $\Gamma_{fl}$ and $te_{eds}$;

(2) $\Gamma_{bs}$ and $te_{fapC}$.

Proof. These cases are different to the others in that we do not constructively exhibit a pair of structures completely. We are still looking for a more constructive proof, as one of these is likely to be more illuminating.

Let $L$ be the language with one sort, on which there is one unary function symbol, succ, and one unary relation symbol, $\phi$.

A normal structure of $L$ is one with domain $\mathbb{N}$, on which succ is interpreted to be the successor function. We say nothing about the interpretation of $\phi$, so there are uncountably many normal structures.

We observe that the pebble complexity of any term $t$ in $L$ over a $\tau$-ary tuple of variables $(x_1, \ldots, x_m)$ is 2. Therefore, for $L$, $\Gamma_{fl} = \Gamma_{bs}$, and for any $L$-structure $\mathcal{M}$, $FAPC(\mathcal{M}) = EDS(\mathcal{M})$.

A composite normal structure of $L$ is the disjoint union of a finite collection of normal structures.

Given an eds $P$, a composite normal structure $\mathcal{M}$ satisfies $P$ if $P^\mathcal{M}$ is non-total.

A scheme $P$ is satisfiable if there exists a composite normal structure which satisfies $P$; we will on one set $S_P$ of normal structures such that $\bigcup S_P$ satisfies $P$ for every satisfiable $P$.

Let Sat be the set of all satisfiable computable functions; we define the saturated normal structure $\mathcal{M}$ to be the disjoint union

$$\mathcal{M} = \bigcup \{ \mathcal{M}_i \mid \mathcal{M}_i \in S_P \text{ for some } P \in \text{Sat} \}.$$
Now, \( \mathcal{M} \) is the disjoint union of countably many normal structures; so, since there are uncountably many different normal structures, there exists a normal structure \( \mathcal{M}_\infty \), say, isomorphically distinct from every substructure of each of the normal structures in \( \mathcal{M} \).

Let \( \mathcal{N} \) be the disjoint union of those normal structures in \( \mathcal{M} \) and \( \mathcal{M}_\infty \). The structures \( \mathcal{M} \) and \( \mathcal{N} \) are shown in Fig. 5.1.17.

These normal structures are grouped into the appropriate composite normal structures

We will show that \( \mathcal{M} \not\equiv_{bs} \mathcal{N} \) but that \( \mathcal{M} \equiv_{eqds} \mathcal{N} \).

Let \( \Sigma \) be the subset \( \phi_{\mathcal{M}_\infty} \) of \( \mathbb{N} \); let \( \Phi \) be the set of types over the variable \( x \)

\[
\Phi = \{ \phi_1, \ldots, \phi_p, \ldots \},
\]

where \( \phi_i(x) \) is the relation \( \Phi(\text{succ}^i(x)) = \text{true} \) if \( i \in \Sigma \), and the relation \( \Phi(\text{succ}^i(x)) = \text{false} \) otherwise. Then the type observation \( \gamma_\Phi \) is in \( \Gamma_{bs} \) and is supported in a normal structure \( \mathcal{N} \) iff \( \mathcal{M}_\infty \prec \mathcal{N} \).

Clearly we have that \( \mathcal{M} \not\vDash \gamma_\Phi \) but that \( \mathcal{N} \vDash \gamma_\Phi \). Thus \( \mathcal{M} \not\equiv_{bs} \mathcal{N} \).
Now let $P$ be an eds. Since $\mathcal{M}$ is a substructure of $\mathcal{N}$, we have that $P$ will be total on $\mathcal{M}$ whenever it is total on $\mathcal{N}$. So, suppose $P$ is undefined somewhere on $\mathcal{N}$. Thus $P \in \text{Sat}$, and there is a tuple $m \in \bigcup \{ \mathcal{M}_i \mid \mathcal{M}_i \in S_P \} < \mathcal{M}$ for which $P$ is undefined at $m$. Thus $\mathcal{M} \equiv_{\text{eds}} \mathcal{N}$.

So we have

(1) $\mathcal{M} \not\equiv_{\text{fr1}} \mathcal{N}$ and $\mathcal{M} \equiv_{\text{eds}} \mathcal{N}$;

(2) $\mathcal{M} \not\equiv_{\text{frs}} \mathcal{N}$ and $\mathcal{M} \equiv_{\text{efapC}} \mathcal{N}$ since $\text{frs}$ implies $\text{efapC}$.
5.1.6 Elementary equivalence

We also need to show that elementary equivalence is independent of termination equivalence (except $te_{\text{loop free}}$).

**Proposition 5.1.12.** (1) Elementary equivalence does not imply $te_{\text{fap}}$-equivalence;

(2) $\Gamma_0$-equivalence does not imply elementary equivalence.

**Proof.** (1) This is a restatement of 2.5.1; we need to find a structure and a program which is total and which does not unwind on that structure. There are any number examples of these on the standard structure of arithmetic, such as the program $f$ in the proof of 5.1.1.

(2) Let $L$ be the language with one sort, one constant symbol, $c$ say, and one binary relation symbol, $\prec$. We will exhibit two $L$-structures $\mathcal{M}$ and $\mathcal{N}$ which are $\Gamma_0$-equivalent but not elementarily equivalent.

We will define the structures as sets of intervals in the rationals $\mathbb{Q}$, ordered by inclusion.

For $\mathcal{M}$, we define $c$ to be the unit interval $(0,1)$; the other elements are

- lower half of interval $(0, 1/2)$;
- divide into two $(0, 1/4)$, $(1/4, 1/2)$;
- lower halves $(0, 1/8)$, $(1/4, 3/8)$;
- divide $(0, 1/16)$, $(1/16, 1/8)$, $(1/4, 5/16)$, $(5/16, 3/8)$;

etc.

We define $\mathcal{N}$ to be the substructure $\mathcal{M} \setminus \{(0,1/2)\}$.

The $L$-structures $\mathcal{M}$ and $\mathcal{N}$ are illustrated if Fig. 5.1.18.
A point \( a \) is \(<\ a \) point \( b \) if there is a sequence of lines joining them and \( a \) is below \( b \).

**Figure 5.1.18.** The structures \( \mathcal{M} \) and \( \mathcal{N} \).

There is clearly an isomorphic embedding \( \mathcal{N} \rightarrow \mathcal{M} \); the map \( \theta: \mathcal{M} \rightarrow \mathcal{N} \) given by

\[
\theta: (0,1) \mapsto (0,1)
\]

and

\[
\theta: (a,b) \mapsto (a/4,b/4) \text{ for } (a,b) \neq (0,1)
\]

is an isomorphic embedding \( \mathcal{M} \rightarrow \mathcal{N} \).

Therefore by 3.4.7 we have that \( \mathcal{M} \equiv_{\Gamma_0} \mathcal{N} \); but the two structures are not elementarily equivalent since the sentence

\[
\exists x. \forall y. (y < c \iff (y < x \lor y = x))
\]

is satisfied in \( \mathcal{M} \) but not in \( \mathcal{N} \).
5.1.7 Specification equivalence

We need to separate the equivalences \( sp_C \) for different classes \( C \) of definable functions.

We will use the following result which gives conditions sufficient to distinguish the equivalences for most of the examples.

**Proposition 5.1.13.** Let \( L \) be a language and \( \mathcal{M} \) an \( L \)-structure which has the unwind property for a class of definable functions \( C \). Then for any \( L \)-structure \( \mathcal{N} \),

\[
\mathcal{M} \equiv_{sp_C} \mathcal{N} \iff \mathcal{M} \equiv \mathcal{N}.
\]

That is, for structures with the unwind property \( sp \)-equivalence is the same as elementary equivalence.

**Proof.** We already know that \( sp \)-equivalence implies elementary equivalence, so suppose that \( \mathcal{M} \equiv \mathcal{N} \); then by 4.3.1 we know that \( \mathcal{N} \) also has the unwind property. Suppose that \( \mathcal{M} \) totally satisfies the formula \( \{ p \} S \{ q \} \) for formulae \( p \) and \( q \), and some scheme \( S \in C \), where

\[
S = \{ E_1(x) \land E_2(x) \land \ldots \land E_{k_s}(x) \to t_s(x) \mid s \in S \}.
\]

Then since \( S \) unwinds on \( \mathcal{M} \), there is a finite subset \( S' \subseteq S \) with

\[
\mathcal{M} \models \forall x. p(x) \Rightarrow ((\forall s \in S'. E_1(x) \land E_2(x) \land \ldots \land E_{k_s}(x)) \land q(t_s(x)));
\]

therefore the same is true on \( \mathcal{N} \) by elementary equivalence and therefore \( \{ p \} S \{ q \} \) is satisfied on \( \mathcal{N} \).

By reversing the argument, we see that \( \mathcal{M} \equiv_{sp_C} \mathcal{N} \). \( \star \)

We can use this fact to separate \( sp \) equivalences.

**Proposition 5.1.14.** Suppose that a structure \( \mathcal{M} \) has the unwind property for a class \( C_1 \) but not for a class \( C_2 \) of definable functions; suppose also that there is a \( P \in C_2 \) such that \( P \) does not unwind on \( \mathcal{M} \) and \( P^\mathcal{M} \) is total. Then \( sp_{C_1} \)-equivalence is different to \( sp_{C_2} \)-equivalence.
Proof. By 2.5.1 there exists a structure \( \mathcal{N} \) elementarily equivalent to \( \mathcal{M} \) on which \( \mathcal{P}^\mathcal{M} \) is non-total. Then \( \mathcal{M} \models \text{spC}_1 \mathcal{N} \) but \( \mathcal{M} \not\models \text{spC}_2 \mathcal{N} \) and so \( \mathcal{M} \not\equiv \text{spC}_2 \mathcal{N} \). *

Using this, we get the following.

Proposition 5.1.15. The following pairs of equivalences are different.

1. \( \text{sp}_\text{fap} \) and \( \text{sp}_\text{loop free} \);
2. \( \text{sp}_\text{fapC} \) and \( \text{sp}_\text{fap} \);
3. \( \text{sp}_\text{fapS} \) and \( \text{sp}_\text{fap} \);
4. \( \text{sp}_\text{fapCS} \) and \( \text{sp}_\text{fapC} \);
5. \( \text{sp}_\text{G}_0 \) and \( \text{sp}_\text{G}_1 \).

Proof. (1) We need a structure on which every loop free program unwinds and there is a total fap which does not unwind. The standard structure \( \mathcal{N} \) of arithmetic with program \( f \) from the proof of 5.1.1 will do.

(2) The structure \( T_0 \) with the fapC which searches through terms (as in 5.1.2) will do.

(3) and (4): It suffices to find a structure \( \mathcal{M} \) for which \( \text{FAPC}(\mathcal{M}) = \text{FAP}(\mathcal{M}) \) satisfying the conditions for 5.1.14 with \( C_1 = \text{fap} \) and \( C_2 = \text{fapS} \). Such an example is the structure \( \mathcal{N} \) of §3 from [Kfoury, 1983].

(5) It suffices to find a structure \( \mathcal{M} \) over a language \( L \) for which \( \mathcal{M}|_L \) is uniformly locally finite for every finite sublanguage \( L' \subseteq L \) and a \( G_0 \) program \( P \) which is total on \( \mathcal{M} \) and does not unwind in \( \mathcal{M} \). We will construct \( \mathcal{M} \) as a disjoint union \( \mathcal{M} = \bigcup_{1 \leq p < \omega} \mathcal{M}_p \).

Let \( L \) be the language with one sort, one unary relation symbol \( \theta \) say, and countably many unary operation symbols \( f_1, \ldots, f_i, \ldots \). For \( p \in \mathbb{N} \), the \( L \)-structure \( \mathcal{M}_p \) has algebraic domain elements \( \{0, \ldots, p\} \).

We define \( \theta \) by \( \theta^\mathcal{M}_p(x) = \text{true} \iff x = 0 \);
define \( f_i \) by
\[
\begin{align*}
&f_i(0) = 1 \quad \text{for } i \leq p, \\
&f_i(x) = x \quad \text{if } x \neq 0 \text{ or } i > p;
\end{align*}
\]

Then \( \mathcal{M} \) is the disjoint union
\[
\mathcal{M} = \bigcup_{1 \leq p < \infty} \mathcal{M}_p.
\]

Clearly \( \mathcal{M}_{\mathcal{L}^*} \) is uniformly locally finite for any finite \( \mathcal{L}' \subseteq \mathcal{L} \) (for any \( x, |x| \leq p+1 \), where \( p = \max(\{f_i | f_i \text{ is in } \mathcal{L}'\}) \)).

The \( G_0 \) scheme
\[
\{ \neg \theta(x) \rightarrow \text{true } \} \cup \{ \theta(x) \land f_1(x) \neq x \land ... \land f_s(x) \neq x \land f_{i+1}(x) = x \rightarrow \text{true } | s > 0 \}
\]
does not unwind on \( \mathcal{M} \) and is total on \( \mathcal{M} \).

We use a different argument to separate other pairs of equivalences.

**Proposition 5.1.16.** The following pairs of equivalences are different.

1. \( sp_{G_1} \) and \( sp_{\text{fapCS}} \);
2. \( sp_{G_2} \) and \( sp_{\text{fapC}} \).

**Proof.** We will show that for some structural (i.e. unary) language \( \mathcal{L} \), the equivalences \( sp_{G_1} \) and \( sp_{\text{fapCS}} \) are different; this will tell us that \( sp_{G_2} \) and \( sp_{\text{fapC}} \) are different since there are uniform translations \( \text{fapCS} \rightarrow \text{fapC} \) and \( G_1 \rightarrow G_2 \) for a structural language by 2.8.4.

The proof is by a counting argument; we will show that the two equivalences give rise to different (numbers of) equivalence classes. We will show that the number of \( sp_{\text{fapCS}} \) equivalence classes \( \leq 2^\kappa \), whereas the number of \( sp_{G_1} \) equivalence classes \( = 2^{2^\kappa} \).

The first of these is easy; since there are countably many eds and logical formulae, there are only countably many specifications. Since an equivalence class is characterised by the set of observations it supports, if the set of observations has cardinality \( \kappa \), the set of equivalence classes can have cardinality at most \( 2^\kappa \); hence \( sp_{\text{eds}} \) has \( \leq 2^\kappa \) equivalence classes.
For the second, we will show that \( te_{G_1} \) gives rise to \( 2^{2^\omega} \) equivalence classes; then we obtain the result from the facts that \( te \)-equivalence implies \( sp \)-equivalence and that there are \( 2^\omega \) specifications for \( G_\Pi \) programs. It will suffice to give a set \( D \) of \( 2^\omega \Gamma_0 \) type-observations and a means of constructing a structure \( \mathcal{M}_S \) supporting just those observations in \( S \) for any given subset \( S \subseteq D \).

Let \( L \) be the language with one sort, two unary relation symbols \( \theta \) and \( \phi \) say, and one unary operation symbol, \( f \) say. A *normal structure* \( \mathcal{M} \) of \( L \) is one with carrier set \( \mathbb{N} = \{0, 1, \ldots \} \) with the interpretations

\[
\theta^\mathcal{M}(x) \equiv \text{true} \iff x = 0;
\]

\[
f^\mathcal{M}(x) = x + 1.
\]

A normal structure \( \mathcal{M} \) is therefore determined by the subset \( \phi^\mathcal{M} \subseteq \mathbb{N} \); the set of normal structures therefore has cardinality \( 2^\omega \). Given a subset \( \Sigma \subseteq \mathbb{N} \), we write \( \mathcal{M}_\Sigma \) to be the normal structure with \( \phi^\mathcal{M}_\Sigma = \Sigma \). Let \( E \) be the set of all normal structures.

Given a subset \( \Sigma \subseteq \mathbb{N} \), we define the type observation \( \gamma_\Sigma \) to be the set

\[
\gamma_\Sigma(x) = \{ \theta(x) \} \cup \{ \phi(f^s(x)) \mid s \in \Sigma \} \cup \{ -\phi(f^s(x)) \mid s \not\in \Sigma \}.
\]

We define \( D \) to be the set of all such observations, i.e. \( D = \{ \gamma_\Sigma(x) \mid \Sigma \subseteq \mathbb{N} \} \). The set \( D \) therefore has cardinality \( 2^\omega \). The set of subsets of \( D \) (remembering an equivalence class corresponds to a subset of the set of observations) is \( 2^{2^\omega} \). Now we need to show that, given a subset \( S \subseteq D \), there exists a structure \( \mathcal{M}_S \) in which just those \( D \)-observations in \( S \) are supported; but we can do this, by defining the structure \( \mathcal{M}_S \) to be the disjoint union

\[
\mathcal{M}_S = \bigcup \{ \mathcal{M}_\Sigma \mid \gamma_\Sigma \in S \}.
\]

It is clear that for every observation \( \gamma_\Sigma \in D \), \( \mathcal{M}_S \models \gamma_\Sigma \iff \Sigma \in S \).
5.1.8 Isomorphism

**Proposition 5.1.17.** There exist non-isomorphic countable L-structures $\mathcal{M}$ and $\mathcal{N}$ for some $L$ with $\mathcal{M} \equiv_{\Gamma_0} \mathcal{N}$.

**Proof.** Let $L$ be the language with one sort, on which there is just one unary relation symbol $f$, say. Pick $p \in \mathbb{N}$; we will define the structure $\mathcal{M}_p$ of $L$.

The carrier set is the set $\{0, \ldots, p\}$. The function $f$ is interpreted as

$$f_{\mathcal{M}_p(x)} = 0 \quad \text{for all } x.$$

The structure $\mathcal{M}_p$ is illustrated in Fig 5.1.19.

![Figure 5.1.19. The structure $\mathcal{M}_p$.](image)

We define the structure $\mathcal{M}_\omega$ in the obvious way: the carrier set is the whole of $\mathbb{N}$, and $f$ is interpreted as

$$f_{\mathcal{M}_\omega(x)} = 0 \quad \text{for all } x.$$

Now define $\mathcal{M}$ to be the disjoint union

$$\mathcal{M} = \bigcup \{ \mathcal{M}_p \mid 0 \leq p < \infty \}$$

and $\mathcal{N}$ the disjoint union

$$\mathcal{N} = \bigcup \{ \mathcal{M}_p \mid 0 \leq p < \infty \}.$$
The structures $\mathcal{M}$ and $\mathcal{N}$ are shown in Fig. 5.1.20.

We will prove that $\mathcal{M}$ and $\mathcal{N}$ are not isomorphic, but that $\mathcal{M} \equiv_{10} \mathcal{N}$.

Firstly, it is clear that they are not isomorphic; for each element $y$ in $\mathcal{M}$, there are at most finitely many $x$ for which $f(x) = y$, but the same is not true in $\mathcal{N}$.

Secondly, observe that every finitely generated substructure of $\mathcal{N}$ is isomorphic to a substructure of $\mathcal{M}$; hence $\mathcal{M} \equiv_{10} \mathcal{N}$ by 3.4.7. •
5.2 Inner congruences

In the remaining sections of this chapter we look at the extent to which te-equivalences are preserved through the join operation. Some of the equivalences are not in fact congruences, so we need to consider the congruences generated both outside (strongest congruence implied by) and inside (weakest congruence implying) them. It turns out that we have met most of these equivalences already.

Let C be a class of computable functions (either fap, fapC, or fapCS). Suppose we have two many sorted languages, L_1 and L_2, and structures M_1, N_1 of L_1 and M_2, N_2 of L_2. Suppose also that M_1 =_{teC} N_1 and that M_2 =_{teC} N_2. Is it the case that when we take the joined structures M_1 + M_2 and N_1 + N_2 they are equivalent by teC? An equivalence with this property will be referred to as a congruence. If the answer to the first question is No, then what is the smallest congruence containing teC? What is the largest equivalence contained in teC? The answers to the second question will be discussed in the next section; we aim to answer the first of these questions in this section. Briefly, the positive answers we obtain are:

1. \( te_{loop\ free} \) is a congruence,
2. \( te \) is not a congruence for any of fap, fapC, eds,
3. \( \Gamma_{fl} \) is the weakest congruence implying \( te \) for eds,
4. \( \Gamma_{bs} \) is the weakest congruence implying \( te \) for both fap and fapC.

These results explain why we introduced \( \Gamma_{fl} \) and \( \Gamma_{bs} \) observations and their equivalences in the first place.

The proofs of some of the above results involve the use of non-effective structures, so in §5.4 we look at whether or not the equivalences are congruences when attention is restricted to effective structures only. The answers contrast to those presented here.

We start with a definition of congruence.
Definition 5.2.1. Let $\sim : \mathcal{L} \mapsto \sim \subseteq \text{Struct}(\mathcal{L}) \times \text{Struct}(\mathcal{L})$ be a scheme associating to each many sorted language $\mathcal{L}$, an equivalence relation $\sim$ between $\mathcal{L}$-structures. We say $\sim$ is a congruence with respect to join if for every pair of languages $\mathcal{L}_1$ and $\mathcal{L}_2$, and all structures $\mathcal{M}_1$, $\mathcal{N}_1$ of $\mathcal{L}_1$ and $\mathcal{M}_2$, $\mathcal{N}_2$ of $\mathcal{L}_2$,

$$\mathcal{M}_1 \sim_{\mathcal{L}_1} \mathcal{N}_1 \text{ and } \mathcal{M}_2 \sim_{\mathcal{L}_2} \mathcal{N}_2 \implies \mathcal{M}_1 + \mathcal{M}_2 \sim_{\mathcal{L}_1 + \mathcal{L}_2} \mathcal{N}_1 + \mathcal{N}_2.$$ 

We will always insist that $\sim$ includes isomorphism; that is, if two $\mathcal{L}$-structures $\mathcal{M}$ and $\mathcal{N}$ are isomorphic, then $\mathcal{M} \sim \mathcal{N}$.

Then the first result we stated.

Proposition 5.2.1. The equivalence $te_{\text{loop free}}$ is a congruence for all structures.

Proof. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be many sorted languages. It suffices to show that if $\mathcal{M}_i$ and $\mathcal{N}_i$ are $\mathcal{L}_i$-structures with $\mathcal{M}_i \equiv_{te_{\text{loop free}}} \mathcal{N}_i$ ($i=1,2$) and a finite type $\Phi = \{\phi_1, \phi_2, \ldots, \phi_p\}$ is satisfied by $\mathcal{M}_1 + \mathcal{M}_2$ then it is also satisfied by $\mathcal{N}_1 + \mathcal{N}_2$.

So, assume that $\Phi$ is satisfied on $\mathcal{M}_1 + \mathcal{M}_2$. We can assume w.l.o.g. that $\Phi$ is a type of form $b$ (see §4.1), i.e. that each $\phi_i$ is an atomic relation or its negation. Now, since each $\phi_i$ is just an atomic relation and there are no operation symbols operating on sorts from both languages, the set $\Phi$ can be partitioned into those atomic relations over $\mathcal{L}_1$ and those over $\mathcal{L}_2$. We therefore assume w.l.o.g. that $\Phi_1 = (\phi_1, \phi_2, \ldots, \phi_q)$ is a type over $\mathcal{L}_1$ and $\Phi_2 = (\phi_{q+1}, \ldots, \phi_p)$ is a type over $\mathcal{L}_2$. But then $\mathcal{M}_1$ satisfies $\Phi_1$ and $\mathcal{M}_2$ satisfies $\Phi_2$. By $te_{\text{loop free}}$-equivalence we have that $\mathcal{N}_1$ satisfies $\Phi_1$ and that $\mathcal{N}_2$ satisfies $\Phi_2$; so $\mathcal{N}_1 + \mathcal{N}_2$ satisfies $\Phi$.

The key there was that we were able to split up the set $\Phi$ because all of the relations $\phi_i$ were of form $b$. We cannot do this with the type based equivalences corresponding to $te$ equivalence for the other classes of programs because they cannot be represented in form $b$ (4.1.2). It is essentially for this reason that those equivalences turn out not to be congruences whilst the stronger ones $\Gamma_0$, $\Gamma_\Pi$ and $\Gamma_{bs}$ (which can be represented in form $b$) are.
Proposition 5.2.2. Let \( \sim \) be an assignment of equivalences to languages which includes isomorphism. Then \( \sim \) is a congruence if and only if

\[
\text{(*) for all languages } L_1 \text{ and } L_2, \text{ and for all structures } M_1 \text{ and } M_1 \text{ of } L_1, R \text{ of } L_2,
\]

\[M_1 \sim_{L_1} M_1 \text{ implies } M_1 + R \sim_{L_1 + L_2} M_1 + R.
\]

Proof. If \( \sim \) is a congruence then (*) is clearly true. Conversely, suppose (*) and let structures \( M_1, M_1 \) of \( L_1 \) and \( M_2, M_2 \) of \( L_2 \) be such that \( M_1 \sim_{L_1} M_1 \) and \( M_2 \sim_{L_2} M_2 \). Putting \( R = M_2 \) in (*), we get \( M_1 + M_2 \sim_{L_1 + L_2} M_1 + M_2 \).

Now putting \( R = M_1 \) in (*) we get \( M_1 + M_2 \sim_{L_1 + L_2} M_1 + M_2 \) since join is commutative up to isomorphism. Hence, by transitivity of \( \sim \), we get \( M_1 + M_2 \sim_{L_1 + L_2} M_1 + M_2 \).

Now we can start the real work.

Proposition 5.2.3. The weakest congruence implying \( te_{eds} \)-equivalence is \( \Gamma^R \)-equivalence.

Proof. Suppose for a set of observations \( \Gamma \), \( \Gamma \)-equivalence is the congruence generated by a given equivalence, \( K \)-equivalence, say (which we will assume includes isomorphism). This means that \( \Gamma \)-equivalence satisfies three conditions:

1. \( \Gamma \)-equivalence is a congruence;
2. \( \Gamma \)-equivalence holding between two structures guarantees \( K \)-equivalence, (typically \( \Gamma \subseteq K \));
3. whenever \( \Gamma' \) has properties (1) and (2) then \( \Gamma \)-equivalence is implied by \( \Gamma' \).

In effect, \( \Gamma \) is the (necessarily non-empty) intersection of a chain

\[K = E_0 \supseteq E_1 \supseteq ... \supseteq E_i \supseteq ...,\]

where \( E_{i+1} \) is obtained from \( E_i \) by

\[\text{for any two structures } M \text{ and } N, M \equiv_{E_{i+1}} N \iff \text{for every structure } N, M + N \equiv_{E_i} N + N.
\]
If we put $\Sigma_0 = te_{eds}$, we find the following facts.

1. $\Sigma_1 \subseteq \Gamma_{fl}$;
2. $\Gamma_{fl}$ is a congruence itself, so equals $\Sigma_1, \Sigma_2, \Sigma_3, \ldots$, so equals $\Gamma$;
3. we already know $\Gamma_{fl} \subseteq te_{eds}$.

We prove each of the facts in turn.

1. Fix the language $L$. Let $M$ and $N$ be structures of $L$ such that for every language $L'$ and structure $\mathfrak{R}$ of $L'$, we have that $M + \mathfrak{R} \equiv_{te_{eds}} N + \mathfrak{R}$, i.e. that $M \equiv_{\Sigma_1} N$.

We show that $M \equiv_{\Gamma_{fl}} N$ which gives us (1).

Let $\Phi = \{\phi_1(x), \phi_2(x), \ldots, \phi_i(x), \ldots\}$ be a type of sort type $\tau = (m; i_1, \ldots, i_m)$ whose observation $\gamma_{\Phi}$ is in $\Gamma_{fl}$. From the definition of $\Gamma_{fl}$, $\Phi$ is a type of form $b$ and there are finitely many operation symbols among the $\phi_i$. Suppose $M \models \gamma_{\Phi}$; we aim to show that $N \models \gamma_{\Phi}$.

Let $L'$ be the language with one algebraic sort, with one constant symbol, 0, one unary function symbol, $\text{succ}$, and one additional unary relation symbol, $\psi$. Let $\mathbb{N}$ be the standard model of arithmetic; we expand $\mathbb{N}$ to a structure $\mathfrak{R}$ of $L'$ by interpreting $\psi$ on a certain subset of $\mathbb{N}$.

We take a G"odel numbering assigning a unique code $c_i$ to each atomic relation $\phi(x)$ or negated atomic relation over $L'_x$. We set $\psi^\mathfrak{R} = \{c_i \mid \phi_i(x) \in \Phi\}$.

Now we write an eds $P$ which, for any $L$-structure $M$, is total on $M + \mathfrak{R}$ iff $M \not\models \gamma_{\Phi}$. This will give us the result, since $M + \mathfrak{R} \equiv_{te_{eds}} N + \mathfrak{R}$.

The program $P$ has a $\tau$-ary tuple of input variables $x$. It searches through the elements of $\mathfrak{R}$ starting at 0 and using $\text{succ}$, until it finds an element $n$ with $\psi(n)$ true. So $n$ is $c_i$ for some $i$. It then decodes $n$ and builds up the corresponding term $\phi_i(x)$ and tests to see if $\phi_i^M(x) \equiv \text{true}$. If it is, then it continues its search for numbers $n$ with $\psi(n)$ true. If $\phi_i^M(x) \equiv \text{false}$, $P$ terminates. In this way $P$ will terminate at $m \in M + \mathfrak{R}$ iff $m$ does not satisfy $\Phi$; so $P$ will be
total on a structure \( P + \mathcal{R} \) iff \( \Phi \) is not satisfied in \( \mathcal{P} \). So since \( \mathcal{M} + \mathcal{R} \) and \( \mathcal{N} + \mathcal{R} \) are \( te_{eds} \)-equivalent we have \( \mathcal{P} \mathcal{M} + \mathcal{R} \) non-total and therefore \( \mathcal{N} \not\models \Phi \). Therefore \( \mathcal{M} \equiv_{\Gamma_{\Pi}} \mathcal{N} \).

(2) This is essentially the same as the \( te_{loop\ free} \) example. It suffices to show that if \( \mathcal{M}_i \) and \( \mathcal{N}_i \) are structures of a language \( L_i \) and \( \mathcal{M}_i \equiv_{\Gamma_{\Pi}} \mathcal{N}_i \) \((i=1,2)\) and a type \( \Phi = \{ \phi_1, \phi_2, \ldots, \phi_p, \ldots \} \) is satisfied by \( \mathcal{M}_1 + \mathcal{M}_2 \) then it is also satisfied by \( \mathcal{N}_1 + \mathcal{N}_2 \).

We know that \( \Phi \) is a type of form \( \mathfrak{b} \). So, assume that \( \Phi \) is satisfied on \( \mathcal{M}_1 + \mathcal{M}_2 \). Now we partition \( \Phi \) into those relations over \( L_1 \) and those over \( L_2 \); let \( \Phi = \Phi_1 \cup \Phi_2 \), where \( \Phi_1 = (\theta_1, \theta_2, \ldots, \theta_i, \ldots) \) is a type over \( L_1 \) and \( \Phi_2 = (\psi_1, \psi_2, \ldots, \psi_i, \ldots) \) is a type over \( L_2 \). But then \( \mathcal{M}_1 \) satisfies \( \Phi_1 \) and \( \mathcal{M}_2 \) satisfies \( \Phi_2 \). By \( \Gamma_{\Pi} \)-equivalence we have that \( \mathcal{N}_1 \) satisfies \( \Phi_1 \) and that \( \mathcal{N}_2 \) satisfies \( \Phi_2 \); so \( \mathcal{N}_1 + \mathcal{N}_2 \) satisfies \( \Phi \).

This tells us that the weakest congruence implying \( te \) for \( eds \) is \( \Gamma_{\Pi} \); because we already know (5.1.11) they are different, we see that \( te_{fapCS} \)-equivalence is not a congruence. The next fact is similar, but for \( fap \) and \( fapC \).

**Proposition 5.2.4.** The inner congruences generated by \( te_{fap} \) and \( te_{fapC} \) are the same and equal to \( \Gamma_{bs} \)-equivalence.

**Proof.** This proof is essentially the same as in the previous example. The difference is in the fact that we are dealing with bounded space in the computable functions, which is why we end up with \( \Gamma_{bs} \)-equivalence instead of \( \Gamma_{\Pi} \). In the case of \( fap \), it would seem at first that there is not enough computational power to run the executive program, but don't forget that we have a copy of \( \mathbb{N} \) working in the composite structure, so this can take care of all the real work.

The fact which makes it work is that if a term has pebble complexity \( n \), then each of its subterms has pebble complexity no more that \( n \), so it is possible in bounded space to build up each of the terms required for \( P \) to operate.
The above results tell us indirectly once again that the equivalences given by $te_{fap}$ and $te_{fapC}$ are not congruences – otherwise they would both be equal to $\Gamma_2$, and we know they are not from 5.1.11.

Finally, we prove that $\Gamma_0$-equivalence is a congruence.

**Proposition 5.2.5.** $\Gamma_0$ equivalence is a congruence.

**Proof.** Once again, this is similar to the case of $te_{\text{loop free}}$ because we are dealing with types in form $b$. It suffices to show that if $\mathcal{M}_i$ and $\mathcal{N}_i$ are structures of a language $L_i$ and $\mathcal{M}_i \equiv_{\Gamma_0} \mathcal{N}_i$ (i=1,2) and a type $\Phi = \{\phi_1, \phi_2, \ldots, \phi_p, \ldots\}$ is satisfied by $\mathcal{M}_1 + \mathcal{M}_2$ then it is also satisfied by $\mathcal{N}_1 + \mathcal{N}_2$.

We know that $\Phi$ is a type of form $b$. So, assume that $\Phi$ is satisfied on $\mathcal{M}_1 + \mathcal{M}_2$. Now we partition $\Phi$ into those relations over $L_1$ and those over $L_2$; let $\Phi$ be the union $\Phi = \Phi_1 \cup \Phi_2$, where $\Phi_1 = \{\theta_1, \theta_2, \ldots, \theta_i, \ldots\}$ is a type over $L_1$ and $\Phi_2 = \{\psi_1, \psi_2, \ldots, \psi_i, \ldots\}$ is a type over $L_2$. But then $\mathcal{M}_1$ satisfies $\Phi_1$ and $\mathcal{M}_2$ satisfies $\Phi_2$.

By $\Gamma_0$-equivalence we have that $\mathcal{N}_1$ satisfies $\Phi_1$ and that $\mathcal{N}_2$ satisfies $\Phi_2$; so $\mathcal{N}_1 + \mathcal{N}_2$ satisfies $\Phi$. •
5.3 Outer congruences

In the previous section, we looked at inner congruences. Their application might be as follows: suppose I want an implementation of a module on which I demand certain termination properties; suppose there is a standard, and the termination properties of an acceptable implementation coincide with those of the standard. Suppose also that I build my implementation up from smaller modules, using join and possibly other structure-building operations. Then in order to guarantee my implementation is te equivalent to the standard, the modules I start out with must be \(\Gamma\) equivalent by the appropriate \(\Gamma\), namely the inner congruence generated by my te.

The outer congruences tell the story the other way around. Suppose I have two te equivalent implementations of a module, and I intend to use each of them in some larger system. Then the best equivalence I can guarantee in general of the larger system will be the outer congruence for my te. In this section, we attempt to characterise the outer congruences for the te equivalences in the same way as we did for the inner congruences.

This is not as easy as it sounds, however. We are not able to find the outer congruence. What we do instead is consider equivalences based on type observations only, and which satisfy certain uniformity conditions on the way they relate to different languages. We argue that these conditions are the least you would expect in any sensible notion of equivalence. Then we show that the strongest type based equivalence implied by \(te_{eds}\) satisfying these conditions which is a congruence is in fact \(te_{loop\ free}\).

Our first example is of an unsatisfactory equivalence.

**Proposition 5.3.1.** There exists a congruence \(\sim\) implied by \(te_{eds}\) and strictly stronger than \(te_{loop\ free}\).

**Proof.** Let \(\sim\) be based on sets of observations. For a language L, these observations are \(te_{eds}\) observations if L has a single algebraic sort, and \(te_{loop\ free}\) observations otherwise.
Clearly ~ is a congruence; if $\mathcal{M}$ and $\mathcal{N}$ are ~-equivalent then they are $te_{\text{loop}}$ free equivalent, so for any $\mathcal{R}$, $\mathcal{M} + \mathcal{R}$ is $te_{\text{loop}}$ free equivalent to $\mathcal{N} + \mathcal{R}$, and therefore ~ equivalent since they now have at least two algebraic sorts.

But we also have examples of structures of single sorted languages which are $te_{\text{loop}}$ free equivalent and not $te_{\text{eds}}$ equivalent, so they are not ~ equivalent; so ~ is strictly stronger than $te_{\text{loop}}$ free.

But examples such as these are not very helpful. It seems unreasonable to me to have an equivalence that agrees with $te_{\text{eds}}$ on single sorted structures but not on others. So we will look at what happens if we impose some restrictions on the nature of ~ in order to rule out this possibility.

The first restriction we make is that we are only going to consider type-based equivalences. Whilst it could be argued that this is less general than might be desirable, we saw in §4.1 that all of our common notions of computability and definability have $te$-equivalences which can be expressed as type-based equivalences. So we will be looking at schemes which assign to each language and each sort type $\tau$ over that language a set of types over $\tau$.

The first condition is that the sets of types should respect language extensions; if $L'$ is an extension of $L$ and $\Phi$ is a $\tau$-ary type over $L$ associated with our equivalence, then $\Phi$ is allowed as a type in $L'$.

Similarly, the sets of types should respect reducts; if $L'$ is an extension of $L$ and $\Phi$ is a type associated with $L'$ but only involving symbols from $L$, then $\phi$ should also be associated with $L$.

Another condition is that we should be able to uniformly substitute sorts and operation symbols in a type; but we will need something more flexible still.

In a computation, the notion of vectorisation is well understood; its analogy in this context is to duplicate a set of tests or operations in some uniform way across the type. We expect to be able to do this in any well behaved set of types; we can do it in all of the sets of types we have been
considering so far, for example. Another thing we commonly do in the context of computations is to uniformly substitute a single operation with a straight line program over the same arguments and additional global arguments. We demand that we are able to do the same thing in our well behaved types.

We will formalise these conditions now.

**Definition 5.3.1.** Let $L$ and $L'$ be languages, $L = (\text{Sort}, \text{Func})$ and $L' = (\text{Sort}', \text{Func}')$. A *language* or *signature morphism* from $L$ to $L'$ is a pair $(\theta, \Theta)$, where

(a) $\theta : \text{Sort} \rightarrow \text{Sort}'$;

(b) $\Theta : \text{Func} \rightarrow \text{Func}'$

such that

whenever $(f, \tau) \in \text{Func}$ and $\tau = (m; i_1, \ldots, i_m, i),$

then $\Theta(f, \tau) = (f', \tau') \in \text{Func}'$ with

$$\tau' = \theta(\tau) = (m; \theta(i_1), \ldots, \theta(i_m), \theta(i)).$$

We will write $\Theta(f)$ to represent $f'$ if $\Theta(f, \tau) = (f', \tau').$

Let $\tau = (m; i_1, \ldots, i_m)$ be a sort type over $L$.

Given a signature morphism $(\theta, \Theta)$, we write $\Theta(\tau)$ to mean the sort type $(m; \theta(i_1), \ldots, \theta(i_m))$.

Given a tuple $x = (x_1, \ldots, x_m)$ of variables of sort type $\tau$, we will write $\Theta(x)$ to mean a tuple $x$ of variables $(x_1, \ldots, x_m)$ of sort type $\Theta(\tau)$.

Let $L(x)$ be the set of terms in $L$ over some tuple $x$ of sort type $\tau$. Given a signature morphism $(\theta, \Theta)$, we define the map $\Theta^* : L(x) \rightarrow L(\Theta(x))$

inductively as follows:

1. if $c$ is a constant symbol, then $\Theta^*(c) = \Theta(c)$;
2. if $x_i$ is a variable, then $\Theta^*(x_i) = \Theta(x_i) = x_i$ say;
3. if $f$ is an operation symbol, $\Theta^*(f(t_1, \ldots, t_k)) = \Theta(f)(\Theta^*(t_1), \ldots, \Theta^*(t_k)).$
Definition 5.3.2. Let $\Pi: L \mapsto \Gamma_L$ be a map from many sorted languages to sets of types over $L$. We say $\Pi$ is uniform if

1. whenever $L'$ is an extension of $L$ and $\Phi$ is a type over $L$,
   $\Phi \in \Gamma_L$ if and only if $\Phi \in \Gamma_{L'}$;
2. whenever $\tau'$ is a sort type extending $\tau$ and $\Phi$ is a $\tau$-ary type, then
   $\Phi \in \Gamma_L$ if and only if $\Phi' \in \Gamma_{L'}$,
   (where $\Phi'$ is just $\Phi$ regarded as a $\tau'$-ary type);
3. whenever $f(y)$ is an operation symbol in $L$ of type $(n; j_1, \ldots, j_n, j)$
   and $\Phi(x) = \{\phi_1(x), \ldots, \phi_{i}(x), \ldots\}$ of type $(m; i_1, \ldots, i_m)$ is in $\Gamma_L$,
   and $t(x, y)$ is a term in $L$ of type $(m+n; i_1, \ldots, i_m, j_1, \ldots, j_n, j)$
   then the type $\Phi'(x, y) = \{\phi_{i}'(x, y), \ldots, \phi_{i}'(x, y), \ldots\}$ is in $\Gamma_L$,
   where for each $i$, $\phi_{i}'$ is $\phi_i$ with $t(x, \_)$ uniformly substituted for $f(\_)$ throughout.
4. whenever $t(y)$ is a term in $L'$ of type $(m; i_1, \ldots, i_m, B)$
   and $(\Theta_1, \Theta_1), \ldots, (\Theta_m, \Theta_m)$ are signature morphisms mapping $L$ to $L'$
   and for each $s \in \{1, \ldots, m\}$, $\Theta_s(B) = i_s$
   and $\Phi$ is a $\tau$-ary type $\Phi(x) = \{\phi_1(x), \ldots, \phi_{i}(x), \ldots\}$ in $\Gamma_L$,
   then the type $t(\Theta(\Phi))$ is in $\Phi_{L'}$, where
   $t(\Theta(\Phi(x))) = \{t(\Theta(\phi_1(x))), \ldots, t(\Theta(\phi_i(x))), \ldots\}$
   and for each $i$, $t(\Theta(\phi_i(x))) = t(\Theta_1^*(\phi_1(x)), \ldots, \Theta_m^*(\phi_i(x)))$.

We are now in a position to prove the result we wanted to.

Proposition 5.3.2. Suppose that $\Pi$ is a uniform map whose equivalence lies between those of $te_{eds}$ and $te_{loop\ free}$. Then $\Pi$ is not a congruence unless $\Gamma_L$ is similar to $te_{loop\ free}$ for every $L$.

Proof. The idea of the proof is as follows. We suppose that $\Gamma_L$ is not similar to $te_{loop\ free}$; then there is an infinite type $\Phi$ say and a structure $\mathcal{M}$ on which $\Phi$ is satisfied. We use the terms in $\Phi$ to define an infinite set of distinct values in a new structure; with those values, we can
start to apply some of the more familiar techniques used in this chapter. We are able to form a saturated structure by taking countably many copies of our free structure, each with a different interpretation of a new relation symbol \( \sigma \). By adding one more such structure, we obtain a non isomorphic structure which is \( t_{eds} \)-equivalent, by saturation. These are the two structures we must distinguish using join and \( \Gamma_L \)-equivalence to complete the proof; uniformity enables us to get a \( \Gamma_L \) type which distinguishes them.

So here we go.

Suppose \( \Gamma_L \) is strictly stronger than \( t_{loop \ free} \) for some \( L = (Sort, Func) \).

Then there exist structures \( M \) and \( N \) for which \( M =_{t_{loop \ free}} N \), but for which there exists a type \( \Phi = \{ \phi_1, \ldots, \phi_p, \ldots \} \) of \( \Gamma_L \) of sort type \( \tau = (m_1, \ldots, i_m) \) say, separating the structures \( M \) and \( N \), i.e. with \( M \not\models \gamma_\Phi \) but \( N \models \gamma_\Phi \).

There must be infinitely many different \( \phi_p \), (going solely on their syntactic structure as terms) as otherwise \( \Phi \) would be a \( \Gamma_3 \) type and hence satisfied on \( M \).

We can now forget \( M \) and \( N \), as they have told us all we need to know.

From now on, we will assume that there are no operation symbols in \( L \) which have algebraic value sort but no algebraic argument sorts, such as constant symbols. Otherwise, if \( f(y) \) is such an operation, we replace it with a new operation \( f^*(x_1, y) \), where \( x_1 \) is a variable of sort \( i_1 \) (the sort of the first argument of \( \Phi \)); we accordingly obtain a new type \( \Phi^* \) by application of clause (3) of the definition of uniformity.

Now we want to construct a new language and a structure over that language where the \( \phi \) are our domain elements and there are infinitely many of them.

Our new language \( L' = (Sort', Func') \) has

1. all of the algebraic and non-algebraic sorts of \( L \);
2. a new sort \( B \) which will act as a surrogate boolean sort;
We define a map \( \theta: \text{Sort} \to \text{Sort}' \) by mapping every sort to itself except for \( B \), which is mapped to \( B \).

The operations of \( L' \) are

1. the usual boolean operations \( \text{and} \) and \( \text{not} \);
2. for every operation symbol \( f \) of type \( \nu = (n; j_1, \ldots, j_n, j) \) a new operation symbol \( f_B \) whose sort type is \( \theta(\nu) \);
3. a relation symbol \( \iota \) of type \( (m; i_1, \ldots, i_m, B) \), (where \( (m; i_1, \ldots, i_m) \) is the type of \( x \))

We define \( \Theta: \text{Func} \to \text{Func}' \) in the natural way; i.e. each operation \( f \) is mapped to \( f_B \).

Now we define our structure; \( \mathfrak{N}_{L' \setminus \{\iota\}} \) is the free \( L' \setminus \{\iota\} \)-structure over the variables \( x \) (together with the standard boolean domain \( B \)); the relation \( \iota \) (for \text{initial}) is defined by

\[ \iota_{\mathfrak{N}} = \{ x \}. \]

The translation \( \Theta(\Phi) \) of the type \( \Phi \), \( \Theta(\Phi(x)) = \{ \Theta(\phi_1(x)), \ldots, \Theta(\phi_i(x)), \ldots \} \) is a subset of the domain \( N_B \) of sort \( B \) in \( \mathfrak{N} \).

We will now extend our language to \( L'' \). All we add is a unary relation symbol \( \sigma \) on sort \( B \).

Now for some definitions we have seen before.

A \textit{normal structure} \( \mathfrak{N} \) of \( L'' \) is any expansion \( \mathfrak{N} \) of \( \mathfrak{N} \) to \( L'' \) for which \( \sigma_{\mathfrak{N}} \subseteq \Theta(\Phi(x)) \). Since the set \( \Theta(\Phi(x)) \) is infinite there are uncountably many normal structures.

A \textit{composite normal structure} of \( L'' \) is the disjoint union of a finite set of normal structures of \( L'' \).

We are going to put together a saturated structure from composite normal structures.
Let $P$ be an eds over $L$; we say $P$ is *satisfiable* if there exists a composite normal structure $\mathcal{P}$ on which $P^\mathcal{P}$ is non total. A composite normal structure $\mathcal{P}$ *satisfies* $P$ if $P^\mathcal{P}$ is non total; for each satisfiable scheme $P$, fix a set $S_p$ whose disjoint union satisfies $P$.

Now make $\mathcal{D}$ the disjoint union

$$\mathcal{D} = \bigcup \{ S_p \mid P \text{ is satisfiable} \}.$$ 

Since there are only countably many programs, but uncountably many normal structures, there is a normal structure, $\mathcal{P}$, say, isomorphically distinct from each of the normal structures from which $\mathcal{D}$ is built. We put $\mathcal{P}$ to be the union

$$\mathcal{P} = \mathcal{P} \cup \bigcup \{ S_p \mid P \text{ is satisfiable} \}.$$ 

These structures are shown in Fig. 5.3.1.

**Figure 5.3.1.** The structures $\mathcal{P}$ and $\mathcal{D}$.
The idea is that no eds can distinguish $\mathcal{P}$ and $\mathcal{D}$ because they are saturated, so they will be $\Gamma_L$-equivalent. However, if the structure $\mathcal{P}$ is joined alongside them both, the information 'coded' into the interpretation of $\sigma$ in $\mathcal{P}$ can be used to distinguish them; furthermore, since we coded $\sigma$ into the type $\Theta(\Phi(x))$, this difference can be detected with a type from $\Gamma$.

We first show that $\mathcal{P} =_{\text{eds}} \mathcal{D}$, and then show $\mathcal{P} + \mathcal{P} =_{\Gamma_L} \mathcal{D} + \mathcal{P}$.

So, let $P$ be a program which fails to terminate somewhere on $\mathcal{D}$; since $\mathcal{D} < \mathcal{P}$, we have that $P$ fails to terminate somewhere on $\mathcal{P}$. Conversely, suppose $P$ fails to terminate somewhere on $\mathcal{P}$. Then $P$ is satisfiable and $S_P < \mathcal{D}$; so there's a point in $\mathcal{D}$ at which $P$ fails to terminate.

Now we are going to join each of these structures $\mathcal{P}$ and $\mathcal{D}$ with $\mathcal{P}$. Remember that the join regards the sorts of the two structures as being distinct, so we get a structure with more sorts than the one we started out with. The point of the whole proof is that when one of these structures is joined with another similar one, it is a fairly easy matter to identify the similar points with a type. The structures $\mathcal{P} + \mathcal{P}$ and $\mathcal{D} + \mathcal{P}$ are illustrated in Fig. 5.3.2.

![Diagram](image)

The interpretations of $\sigma$ in these two parts coincide.

Figure 5.3.2. The structures $\mathcal{P} + \mathcal{P}$ and $\mathcal{D} + \mathcal{P}$. 

Chapter 5 - Separation and congruence - Outer congruences 186
Our aim is to find a $\Gamma$ type separating the joined structures $\mathcal{P} + \mathcal{D}$ and $\mathcal{D} + \mathcal{P}$.

We had a signature morphism $(0, \Theta)$ mapping $L$ to $L'$; now denote by $(0_1, \Theta_1)$ and $(0_2, \Theta_2)$ the two analogous morphisms mapping $L$ into the two copies of $L'$ in $L'' + L''$.

By uniformity there is a type $\Psi(x) = \{\psi_1(x), \ldots, \psi_i(x), \ldots\}$ in $\Gamma_L$ where, for each $i$, $\psi_i(x) \equiv t(x)$, using clause (3) of the definition.

By uniformity again, we obtain the following type as a $\Gamma_{L'' + L''}$ type:

$$\Pi(\Phi) = \{ \Pi(\phi_1), \ldots, \Pi(\phi_p), \ldots \}$$

where

$$\Pi(\phi_p(x)) = \Theta_1(\Psi(x)) \land \Theta_2(\Psi(x)) \land (\sigma(\Theta_1(\phi_p(x))) \leftrightarrow \sigma(\Theta_2(\phi_p(x))))$$

using clause (4) of the definition.

This type is realised at a point $(x, y)$ if and only if:

1. $x$ is the initial point in the $L''$ structure $\mathcal{P}$;
2. $y$ is an initial point in the $L''$ structure $\mathcal{P}$ or $\mathcal{D}$;
3. the interpretations on $\sigma$ on the substructures $\langle x \rangle$ and $\langle y \rangle$ are the same.

Therefore the only point this is realised at is where $x$ is the initial point in $\mathcal{P}$, and $y$ is the initial point of the copy of $\mathcal{P}$ in $\mathcal{D}$.

So its realised somewhere on $\mathcal{P} + \mathcal{D}$, but nowhere on $\mathcal{D} + \mathcal{P}$. So $\mathcal{P} + \mathcal{D} \neq \Gamma_{L'' + L''} \mathcal{D} + \mathcal{P}$.

What we did was suppose $\Gamma$-equivalence is a congruence implied by $te_{eds}$. The structures $\mathcal{P}$ and $\mathcal{D}$ are $te_{eds}$-equivalent, so they must be $\Gamma$-equivalent. But then $\mathcal{P} + \mathcal{P}$ and $\mathcal{D} + \mathcal{P}$ should be $\Gamma$-equivalent; but they are not; therefore there is no uniform congruence implied by $te_{eds}$ but strictly stronger than $te_{loop \ free}$. 
This result (in the presence of 5.2.1) generalises the one proved earlier (5.1.11) that \( t_{\text{eds}} \) is different to \( \Gamma_f \). They use the same saturation technique, but this one in the more general case is a little more messy.

This result of course gives us the following.

**Proposition 5.3.3.** The equivalence \( t_{\text{loop free}} \) is the strongest uniform congruence implied by

\begin{enumerate}
\item \( t_{\text{fapC}} \);
\item \( t_{\text{fap}} \).
\end{enumerate}

**Proof.** Suppose \( \Gamma \) is a uniform congruence implied by either \( t_{\text{fapC}} \) or \( t_{\text{fap}} \); then it is implied by \( t_{\text{fapCS}} \) and is similar to \( t_{\text{loop free}} \).
5.4 Congruences for computable structures

In the proof that the equivalences $\Gamma_\mathfrak{H}$ and $te_{eds}$ were different, we used a counting argument to prove the existence of a structure with particular properties. We know nothing else about that structure except that it exists. We do not know whether or nor it is computable, for example.

In the case of computable structures, that proof will not work, because there are only countably many computable structures of any given language. In this section, we show that the conclusion of that result is actually false for computable structures, and that $te_{eds}$ is indeed a congruence. The same proof was used to prove that $te_{fapC}$ is not a congruence for all structures; once again, we show that it is for computable structures.

In contrast, the proof that $te_{fap}$ is different to $te_{fapC}$ used no such arguments; we exhibited structures $\mathfrak{M}$ and $\mathfrak{N}$ which separated those equivalences. We observed that those structures are indeed computable, and so now have that $te_{fapC}$ is the inner congruence generated by $te_{fap}$ for computable structures.

Proposition 5.4.1.

(1) $te_{fapC}$ is a congruence for computable structures;

(2) so is $te_{fapCS}$;

(3) the inner congruence for computable structures generated by $te_{fap}$ is $te_{fapC}$.

Proof. The first thing to observe is that it actually makes sense to talk about congruences in the case of effective structures; i.e. if $\mathfrak{M}$ and $\mathfrak{N}$ are effective, then so is $\mathfrak{M} + \mathfrak{N}$, from 2.7.2.

(1) It is sufficient to show by 5.2.1. that if $\mathfrak{M}$ and $\mathfrak{N}$ are $te_{fapC}$ equivalent $L$-structures and $\mathcal{R}$ is an effective $L'$-structure, then $\mathfrak{M} + \mathcal{R}$ and $\mathfrak{N} + \mathcal{R}$ are $te_{fapC}$-equivalent.

So, we suppose they are not, and let $P$ be a fapC which we may assume w.l.o.g. is total on $\mathfrak{M} + \mathcal{R}$, but undefined somewhere, at $(n, r)$ say, on $\mathfrak{N} + \mathcal{R}$. Our aim is to construct a fapC $Q$ which is total on $\mathfrak{M}$, but undefined at $n$ on $\mathfrak{N}$.
Since $R$ is computable, we can identify each sort domain with either the first $m$ natural numbers, or the whole of $\mathbb{N}$. Further, on these domains, the interpretations of the operations and relations in $L'$ (the language of $R$) are recursive. The tuple $r = (r_1, \ldots, r_t)$ corresponds in this isomorphism to the tuple $p = (p_1, \ldots, p_t) \in \mathbb{N} \times \ldots \times \mathbb{N}$.

We obtain $Q$ from $P$ is a straightforward way. $Q$ has the same set of algebraic input and program variables over $L$ as $P$, and a counter variable for every counter variable of $P$, and an additional program counter variable for every algebraic program and input variable over $L'$ in $P$.

$Q$ starts off by placing numbers $p_1, \ldots, p_t$ in the counter variables corresponding to the input variables over $L'$ in $P$. Then $Q$ executes the following 'translation' of $P$:

1. variables over $L'$ in $P$ are translated to the corresponding counters in $Q$;
2. operations and relations of $L'$ used in $P$ are translated to their recursive counterparts as subroutines in $Q$;
3. all instructions are translated to themselves, with the appropriate substitution of variables.

An induction over the length of the execution shows that for any tuple $n$ from either $M$ or $N$, the machine state after any execution of $P$ on $M + R$ or $N + R$ corresponds to (by the isomorphism) the machine state after the corresponding execution of $Q$ on $M$ or $N$ respectively.

Therefore $Q^M$ is total, whereas $Q^N$ is undefined at $n$.

2. We can do the same thing with a machine with stacks, by first translating into a machine with natural number stacks, and then appealing to classical results that number stacks can be recursively implemented in numbers.

3. Suppose two structures $M$ and $N$ have the property that for every effective structure $R$, the composites $M + R$ and $N + R$ are $te_{fap}$ equivalent. In particular, by taking $R$ to be the standard structure $\mathbb{N}$, we need $M + \mathbb{N} \equiv_{te_{fap}} N + \mathbb{N}$. But this is just $M \equiv_{te_{fap}} N$. Chapter 5 – Separation and congruence – Congruences for computable structures
Recall that in §5.2 we had that the inner congruence $\Gamma$ generated by an equivalence $\Xi_0$ is the intersection of a chain

$$\Xi_0 \supseteq \Xi_1 \supseteq \ldots \supseteq \Xi_i \supseteq \ldots$$

where $\Xi_{i+1}$ is obtained from $\Xi_i$ by

for any two structures $M$ and $N$, $M \equiv_{\Xi_{i+1}} N \iff$ for every structure $R$, $M + R \equiv_{\Xi_i} N + R$.

So in this case we have $te_{\text{fapC}} \supseteq \Xi_1$. But since $te_{\text{fapC}}$ is itself a congruence, we see that it equals $\Xi_1, \Xi_2, \ldots$ and so it is the congruence generated by $te_{\text{fap}}$ for effective structures.
DISCUSSION AND CONCLUSIONS

The thesis is principally the study of a notion I have called termination equivalence. This was initiated in order that I could start to think about how different algebraic structures, and therefore different implementations of axiomatic specifications, compared when they were used as the basis for computations. The main contributions of the thesis are the following.

(1) I have made a comprehensive study of the termination equivalences arising from the most familiar and relevant programming formalisms. I have established the relative strengths of each of them and their relation to other commonly found equivalences, such as elementary equivalence. In addition, I have made a comprehensive study of a property I have called congruence. The relationship between our equivalences and the congruences associated with them are established.

(2) In the course of doing (1), I have developed and independently studied other ideas related to termination equivalence. The most important instance of this is the concept of saturation, and the proof methods and constructions it leads to.

I will discuss aspects of (1) and (2) above in more detail in the chapter-by-chapter discussion which follows.

The thesis fails to address the potential practical applications of the theory. Although the work was motivated by problems in an application area, I do not feel that the notion of termination equivalence is sufficiently well understood, or indeed appropriate, to be able to apply it to those areas of concern at the outset.

There are also numerous open problems and these provide areas for future work. In the following discussion, I will look at some of them and see how they might be approached. The
whole area is very broad, however, and this will not be comprehensive; I have tried to address only the most directly relevant questions.

Open problems will be examined in the context of the sections to which they relate in the following discussion, together with other points raised.

**Discussion**

**Chapter 2**

§2.1 According to the definition of many sorted structures, they each have a boolean sort. The reason for this is that it simplifies work with computable functions over structures, avoiding the need to find surrogate booleans when switching between programs computing functions and relations. However, the inclusion of a natural number sort is optional since we wish to study structures with insufficient richness to implement counting using fap. This contrasts with the definitions in [Tucker and Zucker, 1988] in which the natural number sort is always assumed to be present.

Whilst it is often said the many sorted case is a trivial generalisation of the single sorted case, it was decided at an early stage that a great deal of the work cannot be formalised in the single sorted case. The reason for this is the interest in combining datatypes with the join operation (§2.2.1). The join of two single sorted structures will always be two sorted, and there is no natural way of representing it as a single sorted structure in such a way that the definitions of computability and definability of partial functions on it remain the same.

§2.2. There was no motivation given for the definition of the disjoint union of a set of structures when it was introduced. I hope that the subsequent use of the construction in numerous examples has provided adequate justification.

§2.3. Several models of computability are introduced, but there are many more proposed in the literature. These include extensions to machine based models such as the inclusion of arrays,
queues, one counter, two counters and others. I have chosen not to include these in this study because I felt that we had a sufficient breadth of interest already to cover the main issues such as storage space, control capability, etc. Where these models translate into models we have looked at, we will of course have results relating their termination equivalences; where they do not, there is the scope for further work.

§2.4. The classes $G_0$, $G_1$ and $G_{bs}$ appear to be new; do they appear anywhere else in the study of definability? There is a large body of literature on generalised recursion theory which proposes different models of recursiveness which are not actually effectively computable (such as [Moschavakis, 1969] and [Grilliot, 1974]). These are certainly different to my definitions, as there are uncountably many schemes in my model but only countably many functions in the other models. I have not studied the relationship between them in any other way.

§2.5. The result 2.5.1 states that the only schemes which are total on every model of a first order theory $T$ are $T$-equivalent to a loop free scheme. This was first proved in the context of recursive schemas, and therefore only for schemes which are eds. The result is clearly true for all schemes, however, and it seemed pointless to only attribute the weaker statement to Kfoury and Park.

§2.6. The language $AP$ defined by Engeler contains formulae but no sentences; by contrast, we have quantifiers in our language and $SP$ contains only sentences. There are two reasons for this:

1. whilst $AP$ only expressed universal properties, we wish to express both universal and existential properties;
2. we insist on observations being sentences, since we use the property that for any sentence $\gamma$ and structure $M$, $M \models \gamma \iff M \models -\gamma$.

§2.7. The problem we highlight over the inadequacy of the usual definition of the computability of a structure arose not because of any lack of insight on the part of other
researchers, but because it is unusual to consider structures over infinite languages. In other words, the problem is not with the existing definition but with the way we expected it to generalise to the infinite case. None of the results we present in this section are surprising or illuminating but we include them for completeness.

Chapter 3.

§3.1. The definition of the observable equivalence $=_{\Gamma}$ from a set $\Gamma$ of observations comes from [Sanella and Tarlecki, 1987]. The facts 3.1.1 (1)-(3) also appear in that paper.

We do not use the notion of $\prec_{\Gamma}$-refinement explicitly, even though there are times when it might be convenient. For example, a fact we often use is that for any two structures $M$ and $N$,

$$M \prec N \implies M \prec_{g_0} N.$$ 

One problem is that the set of observations may be changed in such a way that the equivalence it gives remains the same but the refinement is different. For example, with termination equivalence for $G_0$ schemes we know that $te_{G_0}$ is similar to $\Gamma_0$; however, we also have

$$M \prec_{te_{G_0}} N \iff N \prec_{\Gamma_0} M.$$ 

§3.2. The reason the the inclusion of all of the equivalences $te$, $nd$, $nf$ and $sd$ is partly historical. These were all suggested as being interesting equivalences in the context of datatypes in that they all preserved useful properties and it was only later that the relationships between them (and in particular, that they are all instances of $te$) were discovered.

§3.2.2: The algebraic specification community are very interested in minimal algebras for a variety of reasons. One very important but often undervalued reason is the result (stronger than the conclusion of 1.1.1) asserting that every computable minimal algebra over a language $L$ is (the reduct of) the initial model of an equational theory over $L$ extended with finitely many operation symbols. On the other hand, 3.2.1 tells us that minimal algebras are not the most interesting structures to study in the context of termination equivalence. It is interesting to
observe that the kind of 'behavioural specification' we were looking at in the introduction in relation to random number generators could be of two types: those where we insist that a program is total (corresponding to the assertion that there are no elements with particular properties, analogous to a minimality requirement) but also those where we insist that a particular program is non-total. Instances of these occur in the randomness tests we did not look at in detail, and are the exact opposite of a minimality condition; before I came across this example, I had assumed that these kinds of requirements were of little interest other than as a theoretical possibility.

For example, consider the frequency test for the random number generator; suppose that there is a set $D$ of $N$ symbols from which our numbers are drawn. Suppose that we repeatedly draw random numbers and obtain the sequence $d_1, d_2, ..., d_i, ...$. The frequency test demands that for each symbol $d \in D$, the limit

$$\lim_{n \to \infty} \frac{\# \{ i \in \{1, \ldots, n\} \mid d_i = d \}}{n} = \frac{1}{N}.$$ 

This is equivalent to

$$\forall m \in \mathbb{N}. \exists n_0 \in \mathbb{N}. \forall n \geq n_0. \ f(n) < \frac{1}{m}$$

where

$$f(n) = \left| \frac{\#\{ i \in \{1, \ldots, n\} \mid d_i = d \}}{n} - \frac{1}{N} \right| .$$

Now let $P_{m, d}$ be the program with one natural number input variable $x$ say, which repeatedly draws random numbers and terminates when the bound $f(n) < 1/m$ is exceeded $x$ times; if for some $m$, $P_{m, d}$ is total then the limit of the frequency of occurrences of $d$ does not equal $1/N$. Therefore every $P_{m, d}$ will be non-total precisely when that limit does indeed equal $1/N$ and the
generator passes the test. The suite \( \{ P_{m,d} \mid d \in D, m \in \mathbb{N} \} \) of programs (and the requirement that none of them be total) can therefore be regarded as a behavioural specification for the frequency test.

§3.3: There is no type-based equivalence equivalent to identifiability equivalence. For, consider the following example. Let \( L \) be the language with one algebraic sort, one constant symbol, \( c \) say and one unary relation symbol, \( \phi \) say. We define the following structures of \( L \).

The structure \( \mathcal{M} \) has countably many elements; one is the interpretation \( c^{\mathcal{M}} \) of the constant symbol \( c \), and \( \phi^{\mathcal{M}} \) is false everywhere.

The structure \( \mathcal{N} \) is obtained from \( \mathcal{M} \) by adding one more element to the carrier set and interpreting \( \phi \) to be true on that element; the structure \( \mathcal{P} \) is obtained from \( \mathcal{M} \) by adding two such elements to the carrier set. Clearly we have that

1. \( \mathcal{M} \prec \mathcal{N} \prec \mathcal{P} \);
2. \( \mathcal{M}^* \) has one element, \( c^{\mathcal{M}} \);
3. \( \mathcal{N}^* \) has two elements, \( c^{\mathcal{N}} \) and the element \( n \) say with \( \phi^\mathcal{N}(n) = \text{true} \);
4. \( \mathcal{P}^* \) has one element, \( c^{\mathcal{P}} \);
5. \( \mathcal{M} \equiv_{id} \mathcal{P} \) and \( \mathcal{M} \not\equiv_{id} \mathcal{N} \).

Now suppose that \( S \) is a set of type-observations which distinguishes \( \mathcal{M} \) and \( \mathcal{N} \); since \( \mathcal{M} \prec \mathcal{N} \), we know that there is an observation \( \gamma \) with \( \mathcal{M} \vdash \gamma \) and \( \mathcal{N} \vdash \gamma \). But since \( \mathcal{N} \prec \mathcal{P} \) we have that \( \mathcal{P} \vdash \gamma \) and so \( \mathcal{M} \not\approx_{S} \mathcal{N} \) so \( S \) is not similar to \( id \).

§3.5: I have already mentioned the open problem of whether or not \( te_C \) together with elementary equivalence imply \( sp_C \). In order to show that it is not true, we might try to construct examples separating them. It might be that the two are different for some programming formalisms but the same for others.

The need is for two structures \( \mathcal{M} \) and \( \mathcal{N} \), with \( \mathcal{M} \equiv \mathcal{N} \) and \( \mathcal{M} \equiv_{te_C} \mathcal{N} \) for some class \( C \) of programs, \( \text{fap} \) say. We then need a triple \( \{ p \} \not\approx \{ q \} \) satisfied on \( \mathcal{M} \) but not on \( \mathcal{N} \).
It is clear that:

1. $p$ must not be equivalent to any quantifier-free formula;
2. $S$ must not unwind on the set $p^M$;
3. $q$ is irrelevant as $\text{Th}(M)$ will contain the assertions that $p \Rightarrow q$ for every appropriate terminating path of $S$.

Also, we can see that $S^M$ must not be total, as then we would have it total on $\mathcal{N}$; elementary equivalence will mean that the consistency or otherwise of $p$ is the same on $M$ and $\mathcal{N}$. Our usual tools of structure-building are using the compactness theorem (as in §2.5) and disjoint union; our use of these will be restricted in this example as union does not preserve elementary equivalence and elementary equivalence does not preserve re-equivalence. A better all-round understanding of model theory and in particular the relationship between first-order definable and program-definable sets seems to be necessary.

Chapter 4

§4.3: The whole area of implementability of arithmetic becomes very confusing in the region between local finiteness and $\omega$-richness, as in the case of $T_\omega$. A comprehensive study of the problems would be a significant step.

§4.4: There are a number of questions raised here and not answered. An interesting point is that it seems that in the examples used in the proof of results in §5.1, the larger structure of the pair appears to be an entirely natural extension of the smaller structure. Is there any notion of a 'canonical completion' of a structure?

§4.5: In this section it is proved that finitely generated structures that are computable are precisely those with certain termination properties. This was originally proved in order to prove one of the separation results in 5.1, but that proof did not work. We are left with an interesting fact for which we have not found an application. It is possible that this has already been proved in another context.
§5.1: In this section we prove that a number of pairs of equivalences are different. There is one pair whose relative strength we were not able to settle, namely the termination equivalences for fapS and fapCS; we obtained a partial result (5.1.5) which stated that the equivalences could be separated if there exists a finitely generated non-locally finite structure \( \mathcal{M} \) for which \( \text{FAPS}(\mathcal{M}) \neq \text{FAPCS}(\mathcal{M}) \). The question has been decided for every other pair of equivalences and we have essentially used three different kinds of proof:

1. using the unwind property; this is the technique used by Uryuczyn as discussed in 4.3. To separate \( \text{te}_{C_1} \) and \( \text{te}_{C_2} \), find a structure \( \mathcal{M} \) which has the unwind property for \( C_1 \) but not for \( C_2 \). If you can then find a total \( C_2 \) program which does not unwind, compactness gives another structure \( \mathcal{M}_1 \), elementary equivalent and hence \( \text{te}_{C_1} \)-equivalent to \( \mathcal{M} \) but not \( \text{te}_{C_2} \)-equivalent.

2. by constructing a pair of structures, one of which contains the 'limit' (in the sense of §4.4.2) of a sequence of points in the other structure. This is the method we have used most often, and is used in, for example, 5.1.2 and 5.1.6. We pointed out that, in order to prove the separation results for computable structures we were unable to use the first method, as the structures obtained by compactness are very often not computable (see, for example, [Tennebaum, 1959]).

3. using a relative saturation construction, as in the examples separating \( \text{te}_{G_{fl}} \) and \( \text{te}_{eds} \). We noted that this method also has the drawback of being non-constructive, but we can observe here that the first method (using the unwind property) could not work: for, by a result of Kfoury (Lemma 2.4, [Kfoury, 1983]), every structure which has the unwind property for recursive programs (and equivalently, eds) is uniformly locally finite (over every finite sublanguage) and therefore has the unwind property for \( G_{fl} \). We still do not have a constructive proof of these results.
The case of fap vs. fapC for locally finite structures raises an interesting question. Are there programming formalisms for which the computing powers are the different but the te-equivalences are the same for all structures? It is possible that the undecided question of fapS vs. fapCS would be an example of this.

To what extent is the specification equivalence for a class C determined by the termination equivalence? In particular, is it true that for any two classes C₁ and C₂ of programs containing fap, \( sp_{C_1} \neq sp_{C_2} \iff te_{C_1} \neq te_{C_2} \)? Once again, the difficulty is that our usual structure building techniques are of limited use in constructing examples. We cannot use the usual saturation techniques because disjoint union does not preserve elementary equivalence.

Throughout §5.1 it has been implicitly assumed that in order to separate the equivalences \( te_{C_1} \) and \( te_{C_2} \), it is neccessary to find at least one structure \( M \) for which \( C_1(M) \) (the class of functions computable on \( M \) using programs from \( C_1 \)) is different to \( C_2(M) \); however, I have been unable to prove this.

§5.3: In this section we showed that the strongest uniform type-based congruence implied by \( te_{fap} \) is \( te_{loop \; free} \); we left open the problem of determining exactly what the strongest congruence implied by \( te_{fap} \) is. I have made very little progress on this, except to observe that it is different to \( te_{loop \; free} \).

§5.4: In this section we determined some many termination equivalences that were not congruences for all structures are indeed congruences for computable structures; we also determined that the inner congruence for \( te_{fap} \) for computable structures is still \( te_{fapC} \). However, we do not know what the outer congruence for \( te_{fap} \) is for computable structures, nor do we know what it is under the restrictions of uniform type based equivalences, since the proof of 5.3.2 involved the use of possibly non-computable structures.

Are any of the termination equivalences decidable, or indeed undecidable?
BIBLIOGRAPHY


19 Garland and Luckham [1973]: Program schemes, recursion schemes, and formal languages, JCSS 7.

20 J. Goguen [1989]: Observability in abstract data types, BCTCS5, RHBNC.


24 J. V. Guttag [1975]: The specification and application to programming of abstract data types, Ph.D. Thesis (University of Toronto).


37 A. J. Kfoury [1985a]: Definability by deterministic and non-deterministic programs (with applications to first order dynamic logic), *Inf. and Control* 65 98-121.

38 A. J. Kfoury [1985b]: The pebble game and logics of programs in: *Harvey Friedman's research on the foundations of mathematics* L. A. Harrington et al. (Eds.) (North Holland).


47 B. Mahr and J. A. Makowsky [1983]: Characterising specification languages which admit initial semantics, *Proc. 8th CAAP*, LNCS 159 300-316 (Springer-Verlag).


56 F. Orejas [1985]: On implementability and computability in abstract data types, in (Demetrovics et al. eds.) *Algebra, logic and combinatorics in computer science* (North Holland).


58 D. L. Parnas [1972b]: On the criteria to be used in decomposing systems into modules, *Comm. ACM* 15(12) 1053-1058.


60 M. Paterson and C. Hewitt [1970]: Comparative schematology, *MIT AI Lab Technical Memo 201*.


63 A. Pnueli [1986]: Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends, in LNCS 224 (Springer).


71  J. Tiuryn [1981]: Logic of effective definitions, *Fundamenta Informaticae* IV(3) 629-659.


76  W. M. Turski and T. S. E. Maibaum [1987]: *The specification of computer programs* (Addison Wesley).


80  P. Urzyczyn [1987]: Deterministic context-free dynamic logic is more expressive than deterministic dynamic logic of regular programs, *Fundamenta Informaticae* X 123-142.


INDEX OF DEFINITIONS

Every term we attach a technical meaning to is listed, together with the definition number (or subsection number if none exists) in which it is defined. Symbols appear at the end.

AP, 2.6.1
argument type (of function symbol), §2.1.1
B, Boolean sort, §2.1.1
basic
constructs, §2.4.2
equivalence, 3.1.2
scheme, §2.3.3
branch (of scheme), 4.5.1
calculation (in pebble game), 2.8.1
complete (branch), 4.5.1
complete (structure), 4.4.5
computable
function, §2.3.1
structure, 2.7.1
congruence, 5.2.1
definable function, 2.4.1
defined, 2.6.2
disjoint union (of structures), §2.2.2
distinguishable (structure), 3.4.3
domain
of sort, §2.1.2
of program, §2.3.1
of scheme, §2.3.3
domain, 2.6.2
doms, effective definitional scheme, §2.3.3
effective coordinatisation, 2.7.1
effectively presented, 2.7.1
equivalence
basic, 3.1.2
dt, 3.4.3
distinguishability, 3.4.3
elementary, 3.1.2
id, 3.3.4
identifiability, 3.3.4
observable, 3.1.1
sd, 3.2.3
sp, 3.5.1
specification, 3.5.1
te, 3.2.1
termination, 3.2.1
type, 3.4.6
expansion, §2.1.3
extended pebble game, 2.8.1
extension (of language), §2.1.3
fap, fapC, fapCS, fapS, §2.3.1
Func, function set of language, §2.1.1
G₀, definable functions, 2.4.1
G₉, G₉₀, 4.2.2
generated substructure, §2.1.4
homomorphism, §2.1.6
if, §2.4.2
indistinguishable, 3.4.2
interpretation
of function symbol, §2.1.2
of program, §2.3.1
of scheme, §2.3.3
isomorphic, isomorphism, §2.1.6
join (of structures), §2.2.1
kk-saturated, 4.4.6
loop free (schemes), §2.3.3
N, natural number sort, §2.1.1
nd, 3.2.4
nf, 3.2.2
non-total, 2.6.2
partial (function), §2.3.4
pebble complexity, 2.8.1
pebble game, 2.8.1
periodic, 4.3.2
reduct, §2.1.3
refines (observable), 3.1.1
saturated
from model theory, 4.4.2
w.r.t. C, 4.4.4
for K, 4.4.6
sd, 3.2.3
signature morphism, 5.3.1
similar (sets of observations) 3.1.1
Sort, sort set of language, §2.1.1
sort type (of function symbol), 2.1.1
SP, 2.6.1
sp, 3.5.1
space requirement, 2.8.2
specification, 3.5.1
partial ~ equivalence, 3.5.1
total ~ equivalence, 3.5.1
structural (map, structure), 2.8.3
support (observation), 3.1.1
total (function), 2.3.4
tot, 2.6.2
truth-table property, 2.5.3
translation, 2.3.5
type 3.4.5, 4.1.1, 4.4.1
equivalence, 3.4.6
observation, 3.4.6, 4.1.1
undefined, 2.6.2
uniform (set of types), 5.3.2
uniformly effectively presented, 2.7.2
uniquely identifiable, 3.3.1
uniquely identified, 3.3.1
unwinding, unwind property, 2.5.1, 2.5.2
with parameters, 4.4.3
void (language), §2.1.3
\(\omega\)-rich (structure), 4.3.1
\(\text{tt}, \text{ff}\), boolean values, §2.1.2
\(\langle X \rangle, \langle x \rangle\), substructure generated, §2.1.4
\(\equiv\), isomorphism, §2.1.6
+, join, §2.2.1
\(\bigcup\), disjoint union, §2.2.2
\(\bot_{\mathcal{F}}\), distinguished elements, §2.2.2
\{E \to t\}, clause of scheme, §2.3.3
\(\langle \rangle, ;\), scheme constructs, §2.4.2
\(\Gamma_{\text{basic}}\), basic observations, 3.1.2
\(\equiv\), elementary equivalence, 3.1.2
\(\equiv\), logical identity, §2.1.1
\(\equiv_{\text{bp}}\), basic equivalence, 3.1.2
\(\mathcal{M}^*\), identifiable substructure, 3.3.3
\(\Phi(m)\), type of \(m\), 3.4.5
\(\gamma_{\Phi}\), type observation for \(\Phi\), 3.4.6
\(\Gamma_{\text{f}}, \Gamma_{\text{bs}}, \Gamma_{\text{lf}}\), types, 4.2.1
\(T_{\omega}\), structure, §4.3
\(\mathbb{N}\), natural numbers.
\(\omega\), natural numbers.

Definitions