VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Deriving language recognition algorithms: A case study in combining program specialisation and data refinement

Lindsay Groves*

Technical Report CS-TR-94/8
January 1994

Abstract

We show how several language recognition algorithms can be derived from a single abstract algorithm. This is done using a strategy that involves doing the initial development with a generalised specification, and then specialising the specification, and the development to that point, when this is necessary to expedite later steps — in particular, to allow data refinements. This strategy makes it clear which steps depend on which parts of the specification; we believe it also reflects more accurately the way in which many algorithms are actually discovered. This approach also focusses our attention on finding a common abstraction from which several familiar algorithms can be derived, which increases our understanding of the relationships between these algorithms, and suggests directions for deriving other related algorithms.

Publishing Information

To appear in: David Till and Roger C. F. Shaw (Editors), *Proceedings of the 6th Refinement Workshop*, London, 5-7 January 1994, Springer-Verlag, 1994.

*Email: `lindsay@comp.vuw.ac.nz`.

1 Introduction

One of the major problems in software development is that of controlling complexity. In a large software system there are an enormous number of details to be attended to and numerous interactions between related parts, and hence between design decisions, to be considered. This problem is even more apparent when we attempt to derive software formally — though as Dijkstra points out, the complexity was always there; the use of formalism only makes it more evident. The most effective way known of addressing this problem is separation of concerns: breaking a problem into parts that can be addressed separately and then combined. This division can be either *vertical*, leading to a modular structure, or *horizontal*, leading to a system composed of several layers. In practice both approaches are used in concert, and in both cases we need some kind of strategy for determining how to make the necessary divisions.

One strategy for horizontal division of a system is the use of data abstraction. We initially ignore details of data representation and derive an abstract program which does not depend on any particular data representation; we then introduce the details of a particular data representation, and perhaps make various optimisations admitted by the chosen representation. This process has been studied extensively and has become a major part of modern software design methods. Facilities for supporting various forms of data abstraction have been incorporated in many modern programming languages. The use of data abstraction has been formalised as *data refinement* (e.g. [16], [17]).

An alternative and complementary approach is to begin by considering a *generalisation* of the problem, obtained by ignoring some of the requirements; in particular we can ignore some of the assumptions about the inputs. We can then develop software for the generalised problem, and later *specialise* it to address the omitted requirements. Making use of these additional assumptions often allows the software to be made more efficient — in particular it may allow data refinements that were not previously possible — and may make it feasible. A useful strategy for deciding what to omit is to retain for the initial specification those parts that are common to a family of related problems, so the issue becomes one of identifying a suitable family of problems and finding suitable abstractions with which to describe them.

The use of program specialisation in the context of program refinement has been described by Gravell [7], who shows that a number of different search algorithms and some arithmetic algorithms can be obtained by specialising a generic search algorithm. In this paper, we illustrate the use of program specialisation in tandem with data refinement to derive a family of language recognition algorithms. We begin by considering some simple language recognition problems and derive a recognition algorithm for a generalisation of these. We then specialise this algorithm to obtain algorithms for a number of specific classes of languages. The derivations are presented fairly informally, to emphasis to strategy being used; they can all be formalised in the refinement calculus [15]. As well as demonstrating the utility of these derivation techniques, this also shows some interesting relationships between the algorithms derived.

We begin by formulating the language recognition problem in a very general way (Section 2) and deriving a generic language recognition algorithm (Section 3). We then consider a number of restrictions on the language handled, and data refinements that they allow, to obtain concrete algorithms. Section 4 introduces the techniques of program specialisation and data refinement while discussing the trivial class of singleton languages. Sections 5 through 7 deal with more interesting classes of languages: finite languages, regular languages and a subclass of context-free languages. In Section 8 we present our conclusions. Throughout, we assume standard terminology and definitions from language theory (e.g., see [10] or [11]).

2 Defining the problem

We wish to derive a family of language recognition algorithms, by first deriving a generic recognition algorithm and then specialising it to obtain the various particular algorithms. The kinds of algorithms we wish to derive are those used in common language processing programs such as compilers. In particular, we include recognisers for regular and (some) context-free languages. It is also instructive to consider the simpler cases of finite languages and singleton languages.

The common theme of these language recognition problems is that we are given some alphabet $\mathcal{A}$ (i.e. a finite set of symbols), a string s over $\mathcal{A}$ and a description of a language L over $\mathcal{A}$, and we require to determine whether s is a sentence in L. The problems differ in the class of language to which L must belong; and the details of the resulting algorithms depend heavily on the class of language for their operation.

We will begin with a very general statement of the language recognition problem, ignoring the issue of what kind of language we are recognising. At this stage, we will just take a language to be a set of strings; restrictions to particular classes of languages will be considered later. Thus, our initial statement of the problem is simply: Given a string s and a language L, both over an alphabet $\mathcal{A}$, we wish to determine whether $s \in L$.

We can now formalise this as a specification statement [15]. A specification statement $w\!:\![\, Pre\,,\, Post\,]$ will establish the postcondition $Post$ altering only variables in w, provided the precondition Pre is true initially. If we return the result as a boolean variable, r, we can express our specification as:

$$r\!:\![\, s \in \mathcal{A}^* \;\wedge\; L \subseteq \mathcal{A}^*\,,\, r \equiv s \in L\,]$$

We will assume that string comparison is not a primitive operation, so we are not able to simply compare s with each string in L — in any case, L may not be finite. Instead, we will inspect one symbol of s at a time, and use that to successively narrow down the part of L in which s could occur. We will look at symbols in s from left to right, since that is the way that most of the algorithms we are interested in operate, though other choices are certainly possible. We could formalise these assumptions in terms of a definition of the target programming language in which the solution is to be expressed (e.g. by specifying the operations that can be applied to strings); we prefer, however, to simply use this as guidance in deriving the initial algorithm.

We will introduce one further assumption at this stage: We assume that s and every string in L is terminated with a special "sentinel" symbol (for which we will use \$) which cannot occur anywhere else in s or L. We will formalise this requirement by defining, for any alphabet $\mathcal{A}$, the set of \$-terminated strings to be $\Sigma_{\mathcal{A}}$. Formally[1]:

$$\Sigma_{\mathcal{A}} \;\widehat{=}\; \{\, w \in \mathcal{A}^* \mid \forall i \in 1\mathinner{..}\#w \bullet w[i] = \$ \;\equiv\; i = \#w \,\}$$

Our assumption can now be expressed as $s \in \Sigma_{\mathcal{A}} \;\wedge\; L \subseteq \Sigma_{\mathcal{A}}$.

The main reason for making this assumption is that it simplifies our reasoning later, since it avoids having to make a special case when s is a prefix of some string in L but is not itself a sentence in L (see Section 3.2). It is common in automata theory to assume that the

[1] We index strings from 1 and write $\#s$ for the length of string s. We write $s[i\mathinner{..}j]$ for the substring of s beginning at index i and ending at index j, which is empty (λ) if $i > j$, and use $s[i\mathinner{..}]$ as an abbreviation for $s[i\mathinner{..}\#s]$. To simplify notation, we do not distinguish between a symbol and a string of length one, thus we treat $s[i\mathinner{..}i]$ the same as $s[i]$.

input ends with some kind of end-of-input (or end-of-tape) marker, presumably for exactly the same reason — to avoid excessive special cases; our sentinel symbol plays exactly the same rôle. The addition of the sentinel symbol can be considered as a data refinement. In many systems, strings, files, etc. are implemented like this anyway. In other cases, we need to add the sentinel symbol when the actual end of input is seen — in which case we have an explicit data conversion performed "on-the-fly".

With this additional assumption, the specification is now:

$$r : [\, s \in \Sigma_{\mathcal{A}} \;\wedge\; L \subseteq \Sigma_{\mathcal{A}} \,,\, r \equiv s \in L \,]$$

Since the preconditions $s \in \Sigma_{\mathcal{A}}$ and $L \subseteq \Sigma_{\mathcal{A}}$ remain invariant throughout, we will treat them as global invariants and not include them explicitly in subsequent specifications, loop invariants, etc[2].

3 Deriving a generic language recognition algorithm

We will now proceed to derive a generic language recognition algorithm to implement the above specification. As indicated above, we will look at symbols in s one at a time, from left to right. Our generic recognition algorithm will therefore need a loop which iterates through successive symbols of s.

We observe that each symbol of s that the algorithm looks at effectively reduces the part of L in which s could occur. Thus, we can view the algorithm as computing a sequence of sets $L_1, L_2, \cdots$, where L_k is the set of strings in L that have $s[1 \mathinner{..} k-1]$ as a prefix. We note that $L_1 = L$ and $L_{k+1} \subseteq L_k$ for $k = 1, \cdots, \#s$. If $L_k = \{\}$ for any $k \in 1 \mathinner{..} \#s$, then $s \notin L$; and if $\$ \in L_k$ for any $k \in 1 \mathinner{..} \#s$, then $s \in L$.

Rather than computing $L_1, L_2, \cdots$ explicitly, we will construct our generic algorithm so that it computes the sets $C_1, C_2, \cdots$, where C_k is the set of all strings that when contatenated with $s[1 \mathinner{..} k-1]$ give a string in L (and in L_k). Thus, $L_k = \{s[1 \mathinner{..} k-1]\} \frown C_k$. We can think of the set C_k at each step as the *candidate completion set*: s is in L if and only if the remaining input ($s[k \mathinner{..}]$) is in C_k. Thus, we need to maintain the following relationship:

$$s \in L \;\equiv\; s[k \mathinner{..}] \in C_k$$

This relationship can be taken as the basis of a loop invariant; we only need to add a constraint on k. The invariant we will use is:

$$I \mathrel{\widehat{=}} k \in 1 \mathinner{..} \#s \;\wedge\; s \in L \;\equiv\; s[k \mathinner{..}] \in C_k$$

We can express C_k more directly by introducing a new function Rem, such that $Rem(s, L)$ is the set of all strings that can be concatenated to the right of s to form a sentence in L; i.e.

$$Rem(s, L) \mathrel{\widehat{=}} \{\, t \mid s \frown t \in L \,\}$$

We now see that $C_k = Rem(s[1 \mathinner{..} k-1], L)$.

Note that, for any strings u and v, $Rem(u \frown v, L) = Rem(v, Rem(u, L))$. In particular, $C_{k+1} = Rem(s[1 \mathinner{..} k], L) = Rem(s[k], Rem(s[1 \mathinner{..} k-1], L)) = Rem(s[k], C_k)$.

Finally, we observe that when $C_k \neq \{\}$, the loop invariant implies that $s[1 \mathinner{..} k-1] \in Prefixes(L)$, where $Prefixes(L)$ is the set of all prefixes of strings in L; i.e.

$$Prefixes(L) \mathrel{\widehat{=}} \{\, w \in \mathcal{A}^* \mid \exists t \in \mathcal{A}^* \bullet w \frown t \in L \,\}$$

[2]This can be formalised using local invariants, as described in [14].

3.1 Loop design

We can now proceed to design a loop using the loop invariant I given above and the variant $n - k$, and using a single variable C to hold successive values of C_k. We will not attempt to compute r directly in the loop. Instead, the loop will manipulate C into a form from which we can easily compute the required value for r.

There are four steps in constructing the loop:

Initialisation: We can easily establish I by establishing $k = 1 \wedge C = L$, since $C_1 = L$.

Guard: It is not yet clear what the guard should be, so we will just write it as G for now[3].

Progress: We can ensure that the loop makes progress (i.e. decreases the variant) by increasing k. We have already assumed that we will look at one symbol of s at a time, so we will increase k by 1.

Maintain I: Since $C_{k+1} = Rem(s[k], C_k)$, we can maintain I as k increases, by setting C to $Rem(s[k], C)$.

The algorithm at this stage is:

$$
\begin{array}{l}
|[\ \textbf{var}\ k, C\ \bullet \\
\quad k := 1\ \ ||\ \ C := L; \\
\quad \textbf{do}\ G\ \rightarrow \\
\quad\quad k := k + 1\ \ ||\ \ C := Rem(s[k], C) \\
\quad \textbf{od}; \\
\quad r\!:\![\, I \wedge \neg G\, ,\, r \equiv s \in L\,] \\
]|
\end{array}
$$

This algorithm is written using a parallel composition operator[4] ($||$), defined by:

$$w_1\!:\![\, P_1\, ,\, R_1\,]\ \ ||\ \ w_2\!:\![\, P_2\, ,\, R_2\,]\ \equiv\ w_1, w_2\!:\![\, P_1 \wedge P_2\, ,\, R_1 \wedge R_2\,]$$

A parallel assignment $x_1 := e_1 \,||\, x_2 := e_2$ is equivalent to the usual multiple assignment $x_1, x_2 := e_1, e_2$ when $x_1 \neq x_2$. When $x_1 = x_2$, it is equivalent to $x_1 := e_1$ if $e_1 = e_2$, and is miraculous otherwise. We use parallel composition in preference to multiple assignments, since it makes the location and effect of replacements during data refinement more obvious, and simplifies data refinements where an assignment data refines to more complex code. A parallel composition is introduced using the following refinement law[5]:

$$\text{Introduce } || :\ \ \frac{w_1 \cup w_2 \subseteq w\ ,\quad P \Rightarrow P_1 \wedge P_2\ ,\quad R_1 \wedge R_2 \Rightarrow R}{w\!:\![\, P\, ,\, R\,]\ \sqsubseteq\ w_1\!:\![\, P_1\, ,\, R_1\,]\ \ ||\ \ w_2\!:\![\, P_2\, ,\, R_2\,]}$$

[3] We are treating G as a metavariable [18].

[4] This is a very strong form of parallelism that cannot in general be turned into a sequential composition without adding extra variables.

[5] This law only covers the binary case; it can be extended to an n-ary composition in the obvious way.

3.2 Terminating conditions

In order to determine a suitable formula for G, we consider the various conditions under which the loop might terminate. The loop must terminate if s is found to be in L: this can be detected when $s[k] = \$$ and $\$ \in C$. The loop should also terminate if s is found not to be in L: this can be detected when $s[k]$ is not the initial symbol of any string in C[6].

In order to express these conditions concisely, we introduce a function Hds such that, for any set of strings X, $Hds(X)$ is the set of intial symbols of strings in X; i.e.

$$Hds(X) \mathrel{\widehat{=}} \{ a \in \mathcal{A} \mid \exists w \bullet a \frown w \in X \}$$

Noting that $\$ \in C$ is equivalent to $\$ \in Hds(C)$, we now want the loop to terminate when either $s[k] = \$ \ \wedge\ \$ \in Hds(C)$ or $s[k] \notin Hds(C)$ is true. That is, we wish to find a guard G such that $\neg G$ is equivalent to $(s[k] = \$ \ \wedge\ \$ \in Hds(C)) \vee s[k] \notin Hds(C)$.

With a little formula manipulation, we arrive at[7]:

$$G \mathrel{\widehat{=}} s[k] \neq \$ \ \wedge\ s[k] \in Hds(C).$$

We can now calculate r from $I \ \wedge\ \neg G$, since when this holds, s is in L if and only if $s[k] = \$ \ \wedge\ \$ \in Hds(C)$ is true.

The complete abstract algorithm is now:

$$\begin{array}{l}
|[\ \textbf{var}\ k, C\ \bullet \\
\quad k := 1\ \ ||\ \ C := L; \\
\quad \textbf{do}\ s[k] \neq \$ \ \wedge\ s[k] \in Hds(C)\ \rightarrow \\
\qquad k := k + 1\ \ ||\ \ C := Rem(s[k], C) \\
\quad \textbf{od}; \\
\quad r := s[k] = \$ \ \wedge\ \$ \in Hds(C) \\
]|
\end{array}$$

This algorithm is still not executable because it is expressed in terms of the set variables L and C, and the set operations Hds and Rem which cannot be implemented for arbitrary sets. In order to obtain an executable program, we will have to place some restriction on L, and then explore suitable representations for L and C, and corresponding implementations for Hds and Rem. Obviously we won't be able to turn this into a concrete algorithm for all Ls: If L is not decidable, we won't be able to complete the data refinement.

We will now consider various restrictions on L and representations for C that can then be used to obtain recognition algorithms for particular classes of languages. In each case we will need to use the additional assumptions about L in order to justify the data refinement used to represent C. All of the languages will be assumed to be non-empty, since this allows simpler data representations to be used.

4 Specialisation where L is a singleton

For our first specialisation, we will consider the simpest possible case, where L is a singleton, so the recognition algorithm problem (and algorithm) reduces to one of string comparison.

[6] These conditions are significantly simplified by our assumption about sentinel symbols.

[7] An alternative guard is $\lambda \notin C \ \wedge\ C \neq \{\}$. The guard chosen makes subsequent data refinement easier, since it ensures that C is never empty so long as L is not empty.

This will allow us to present the techniques of specialisation and data refinement in a very simple context.

To specialise a program, we introduce additional assumptions about the inputs, and propagate these through the resulting derivation. This process is justified, since we can show that if $w:[P, R] \sqsubseteq w':[P', R']$ then, for any additional assumptions A, we have $w:[P \wedge A, R] \sqsubseteq w':[P' \wedge A, R']$.

When L is a singleton, the additional assumption is $\#L = 1$. Adding this assumption does not affect the generic algorithm we have derived, but does strengthen the loop invariant and various preconditions, allowing us to choose good representations for L and C. We now consider how such representations are to be introduced.

Data refinement involves replacing a program fragment involving certain variables (known as *abstract variables*) by an "equivalent" fragment using different variables (known as *concrete variables*). The required relationship between abstract, concrete and other program variables, is specified in terms of a given *coupling invariant* (or *abstraction invariant*). Our data refinement technique is similar to those of [3] and [6], in that encoding and decoding is done using explicit statements; it is also similar to the auxiliary variable technique [13], in that the encoding and decoding statements are not responsible for variable declarations. A notable feature of our approach is that we allow data refinement of global variables, not just locals. Data refining a global variable imposes an obligation on the surrounding environment to deliver/receive the variable in the data refined form. The surrounding environment may be data refined in the same way, or may require an explicit mapping between the representation it uses and that introduced by the data refinement. If the surrounding environment is a user, then the onus is placed on the user to input or read values in the data refined form. This approach is discussed in more detail in [8]; a similar approach is found in [4].

Since L contains a single string, we can represent L by a string variable, say t; so the coupling invariant will entail $L = \{t\}$. We can now represent C using t and k, since $s[1 \mathinner{..} k-1] \in \mathit{Prefixes}(L)$ gives us $s[1 \mathinner{..} k-1] = t[1 \mathinner{..} k-1]$. Thus, $C = \mathit{Rem}(s[1 \mathinner{..} k-1], \{t\}) = \{t[k \mathinner{..}]\}$.

We can express this relationship between the abstract variables L and C and the concrete variables t and k in terms of the coupling invariant:

$$L = \{t\} \wedge C = \{t[k \mathinner{..}]\} \wedge k \in 1 \mathinner{..} \#t$$

It is easy to see that this representation exists whenever L is a singleton; i.e. for any singleton L, values of t and k can be found that satisfy the above coupling invariant.

With this representation we can implement $\mathit{Hds}(C)$ by $\{t[k]\}$, so $x \in \mathit{Hds}(C)$ becomes $x = t[k]$, and $\mathit{Rem}(s[k], C)$ by $\{t[k+1 \mathinner{..}]\}$, so long as $s[k] = t[k]$, which will be guaranteed by the loop guard.

We can now data refine the generic algorithm using this data representation. The net effect of the data refinement is to delete the declaration of C (we already have a declaration for k, and t is assumed to be global), and replace statements and guards involving L and C by corresponding code involving t and k. The required replacements are determined as follows (where $\sqsubseteq$ indicates algorithmic refinement and $\preceq$ denotes data refinement with respect to the given coupling invariant):

$$
\begin{array}{rcl}
C := L & \equiv & C\!:\![\,C = L\,] \\
& \preceq & k\!:\![\,\{t[k\,..]\} = \{t\}\,] \\
& & (t \text{ can't change, since } L = \{t\} \text{ must be preserved}) \\
& \equiv & k\!:\![\,t[k\,..] = t[1\,..]\,] \\
& \equiv & k\!:\![\,k = 1\,] \\
& \sqsubseteq & k := 1
\end{array}
$$

$$
\begin{array}{rcl}
x \in Hds(C) & \preceq & x \in \{t[k]\} \\
& \equiv & x = t[k] \\
& & (\text{The coupling invariant ensures } k \in 1\,..\,\#t)
\end{array}
$$

$$
\begin{array}{rcl}
C := Rem(s[k], C) & \equiv & C\!:\![\,C = Rem(s[k], C_0)\,] \\
& \preceq & k\!:\![\,\{t[k\,..]\} = Rem(s[k_0], \{t[k_0\,..]\})\,] \\
& & (t \text{ can't change, since } L = \{t\} \text{ must be preserved}) \\
& \equiv & k\!:\![\,\{t[k\,..]\} = \{t[k_0 + 1\,..]\}\,] \\
& & (\text{Since the loop guard gives } s[k_0] = t[k_0]) \\
& \equiv & k\!:\![\,t[k\,..] = t[k_0 + 1\,..]\,] \\
& \equiv & k\!:\![\,k = k_0 + 1\,] \\
& \sqsubseteq & k := k + 1
\end{array}
$$

The specification statements used here have an implicit precondition, which is the weakest precondition to allow the statement to be feasible. Thus, the specification statement $w\!:\![\,R\,]$ is equivalent to $w : [\,\exists w \bullet R\,,\,R\,]$. This form of specification statement is especially useful in performing data refinement since we often need to turn assignments into specification statements in order to data refine them. In practice, we often need to appeal to a stronger precondition, e.g. to take account of a loop invariant or guard, as in the data refinement of $C := Rem(s[k], C)$ above. This can be formalised by using assertions to propagate the required conditions or using miracles [12].

The loop invariant becomes:

$$
\begin{array}{rl}
& 1 \le k \le \#s \;\wedge\; s \in \{t[1\,..]\} \;\equiv\; s[k\,..] \in \{t[k\,..]\} \\
\iff & 1 \le k \le \#s \;\wedge\; s = t[1\,..] \;\equiv\; s[k\,..] = t[k\,..] \\
\iff & 1 \le k \le \#s \;\wedge\; s[1\,..\,k-1] = t[1\,..\,k-1]
\end{array}
$$

The resulting algorithm after performing this data refinement is:

$$
\begin{array}{l}
|[\;\; \mathbf{var}\; k \;\bullet \\
\quad k := 1 \;\;||\;\; k := 1; \\
\quad \mathbf{do}\; s[k] \ne \$ \;\wedge\; s[k] = t[k] \;\rightarrow \\
\qquad k := k + 1 \;\;||\;\; k := k + 1 \\
\quad \mathbf{od}; \\
\quad r := s[k] = \$ \;\wedge\; \$ = t[k] \\
]|
\end{array}
$$

Note that the parallel composition has allowed us to data refine the assignments to C without touching the parallel assignments to k; with multiple assignments, we would have to convert the whole multiple assignment into a single specification statement before performing the data refinement. In this case, the assignments introduced by the data refinement duplicate ones that are already performed by the algorithm. Using the property of parallel assignments mentioned earlier, this algorithm simplifies to:

$$
\begin{array}{l}
|[\ \textbf{var}\ k\ \bullet \\
\quad k := 1; \\
\quad \textbf{do}\ s[k] \neq \$\ \wedge\ s[k] = t[k]\ \rightarrow \\
\qquad k := k + 1 \\
\quad \textbf{od}; \\
\quad r := s[k] = \$\ \wedge\ t[k] = \$ \\
]|
\end{array}
$$

This is an elegant way of writing a string comparison algorithm, which deals with the problem of two stopping conditions rather nicely. It is reassuring to see that specialising a generic algorithm in this trivial case leads to such a simple algorithm.

In this derivation, we have assumed that L is given in terms of the representation used here, i.e. as the string t. The surrounding environment may either be similarly data refined so that it represents L in this way, or if L is obtained in some other way (e.g. it might be a substring of some other string), may require an explicit conversion to get it in this form.

5 Specialisation where L is a finite set of strings

We will now consider specialising the generic algorithm in the case where L is a non-empty finite language, so the additional assumption is $L \neq \{\}\ \wedge\ \#L \in \mathbb{N}$ (where $\mathbb{N}$ denotes the natural numbers). In this case the problem reduces to a table look-up where L is the set of keys in the table and s as the search key. Since we are processing s one symbol at a time, we should represent the table in a way that allows Hds and Rem to be computed efficiently.

If we represented C as a list of strings, we would need some way of keeping track of the remaining candidates, by either deleting or marking the strings that are no longer candidates. This would be cumbersome and inefficient, since on each iteration we would need to compare $s[k]$ with the kth element of every remaining candidate. We can avoid this overhead by using a representation for C in which the set $Hds(C)$ is represented explicitly for every value that C may assume. We can do this by representing C using a *trie* table (see, for example, [1], p163*ff*).

Informally, a trie is a tree in which each edge is labelled with a symbol from $\mathcal{A}$ and the labels on the out-edges of any node are distinct. Each path in the tree represents a string obtained by concatenating all of the symbols on the edges in the path. A trie represents the set of strings represented by all paths from the root to a leaf.

To describe this more formally, we will assume the following operators on trees:

$Root(T)$	The node at the root of tree T.
$Nodes(T)$	The set of all nodes of tree T.
$IsLeaf(p)$	True iff p is a leaf, where $p \in Nodes(T)$.
$OutLabels(p)$	The set of symbols labelling out-edges of p, where $p \in Nodes(T)$.
$Succ(a, p)$	The node at the other end of the out-edge of node p, labelled a, where $p \in Nodes(T)$, p is not a leaf and $a \in OutLabels(p)$.

We define a function Str such that, if p and q are nodes in T and there is a path from p to q, then $Str(p, q)$ is the string obtained by concatenating all of the symbols on the edges in that path; i.e.

$$Str(p,q) \mathrel{\widehat{=}} \begin{cases} \lambda, & \text{if } p = q \\ a \frown t, & \text{if } Str(Succ(p,a),q) = t \end{cases}$$

We now define a function $Strings$ so that, for any node p, $Strings(p)$ is the set of all strings on paths from p to a leaf; i.e.

$$Strings(p) \mathrel{\widehat{=}} \{ w \mid \exists q \in Nodes(T) \bullet IsLeaf(q) \ \wedge\ Str(p,q) = w \}$$

Thus, $Strings(Root(T))$ is the set of strings represented by a trie T.

We will represent L using a trie T, such that $Strings(Root(T)) = L$. There will be one leaf in the trie corresponding to each sentence in L. Our assumption about sentinels implies that each leaf has \$ as its in-edge label, and \$ doesn't occur on any other edges.

We observe that for each $w \in Prefixes(L)$, there is a unique node p in T such that $Str(Root(T),p) = w$ and $Strings(p) = Rem(w,L)$. Thus, we can represent C by a node in T. We also observe that if $Strings(p) = C$, then $Hds(C) = Outlabels(p)$ and $Rem(a,C) = Strings(Succ(a,p))$ for all $a \in Hds(C)$.

Thus, we can represent L by $Root(T)$, and C by a node in T, indicated by a variable p. The coupling invariant is now:

$$L = Strings(Root(T)) \ \wedge\ p \in Nodes(T) \ \wedge\ C = Strings(p)$$

It is easy to see that this representation exists whenever L is finite and non-empty[8]. If L is finite, $Prefixes(L)$ is finite, and so is T. Since T is finite, we can implement it and the required operations in any of the standard ways.

We can now data refine the generic algorithm using this data representation, by making the following replacements:

$$\begin{array}{rcl} C := L & \equiv & C\!:\![\, C = L \,] \\ & \preceq & p\!:\![\, Strings(p) = Strings(Root(T)) \,] \\ & \sqsubseteq & p\!:\![\, p = Root(T) \,] \\ & \sqsubseteq & p := Root(T) \end{array}$$

$$x \in Hds(C) \ \preceq\ x \in OutLabels(p)$$

$$\begin{array}{rcl} C := Rem(s[k],C) & \equiv & C\!:\![\, C = Rem(s[k],C_0) \,] \\ & \preceq & p\!:\![\, Strings(p) = Rem(s[k], Strings(p_0)) \,] \\ & \equiv & p\!:\![\, Strings(p) = Strings(Succ(s[k],p_0)) \,] \\ & & \text{(Since the loop guard gives } s[k] \in OutLabels(p_0)\text{)} \\ & \sqsubseteq & p\!:\![\, p = Succ(s[k],p_0) \,] \\ & \sqsubseteq & p := Succ(s[k],p) \end{array}$$

The loop invariant becomes:

$$\begin{array}{rl} & 1 \le k \le n \ \wedge\ s \in Strings(Root(T)) \ \equiv\ s[k\mathinner{..}] \in Strings(p) \\ \iff & 1 \le k \le n \ \wedge\ s[1\mathinner{..}k-1] = Str(Root(T),p) \end{array}$$

[8] If we had allowed L to be empty, we would also need to allow T to be an empty tree, in which case $Root(T)$ would not exist, so some more complex representation would be required.

The resulting algorithm is:

$$
\begin{array}{l}
[\![\ \textbf{var } k, p \ \bullet \\
\quad k := 1 \ \| \ p := Root(T); \\
\quad \textbf{do } s[k] \neq \$ \ \wedge \ s[k] \in OutLabels(p) \ \rightarrow \\
\quad\quad k := k + 1 \ \| \ p := Succ(s[k], p) \\
\quad \textbf{od}; \\
\quad r := s[k] = \$ \ \wedge \ \$ \in OutLabels(p) \\
]\!]
\end{array}
$$

This algorithm is simply a trie look-up, much as we would find in a data structures text book (e.g. [5], p217) — though it is simpler than most!

The refinement of $C := L$ assumes that we have T already available. We may either assume that L is initially given in this form or construct T from some other representation of L. If L is given, say as a list of strings, the precomputation required to construct the trie is an explicit data mapping from the external representation of L to the representation used here.

6 Specialisation where L is a regular language

Now consider the case where L is a non-empty regular language, described by a regular expression E. Thus, we specialise the initial specification and derivation, with the additional assumption $L \neq \{\} \ \wedge \ L = Lang(E)$, where $Lang(E)$ is the language denoted by the regular expression E. We will assume that the syntax and semantics of regular expressions are defined in the standard way (see, for example, [10] or [11]).

Once again, we want to represent C in a way that allows the Hds and Rem functions to be computed efficiently. Let's again try to represent C in such a way that $Hds(C)$ is represented explicitly using a tree, as we did for finite languages. Unfortunately, since L may be infinite, the resulting tree may be infinite, so we can't store it explicitly. Fortunately, however, we can construct a finite directed graph which contains the same information. We will use the same operations on graphs that we used on trees in Section 5.

Let T be the (possibly infinite) tree constructed as in Section 5. We construct a graph G corresponding to T as follows:

- Nodes in G correspond to equivalence classes of nodes in T; if $Strings(p) = Strings(q)$ for two nodes $p, q \in Nodes(T)$, then p and q map onto the same node in G.
- The root of G corresponds to the equivalence class containing the root of T.
- There is an a-edge from node p to node q in G if and only if there is an a-edge from a node in the equivalence class corresponding to p to a node in the equivalence class corresponding to q in T.

Note that this construction preserves the property that the labels on the out-edges of any node are distinct.

As with a trie, each path represents a string and the graph represents the set of strings represented by all paths from the root to a leaf. In fact, this graph will have exactly one leaf node, since $Strings(p) = \{\lambda\}$ for every leaf p in T. Our assumption about sentinels implies that all edges into this leaf node are labelled with \$, and these are the only edges labelled \$.

Since there may now be more than one path between two nodes, we define a new function STR such that for any nodes p and q in G, $STR(p,q)$ is the *set* of *all* strings on *any* path from p to q; i.e. $STR(p,q)$ is the *smallest* set such that:

(i) if $p = q$, then $\lambda \in STR(p,q)$, and

(ii) if $a \in OutLabels(p)$, then $\{a\} \frown STR(Succ(p,a),q) \subseteq STR(p,q)$.

Note that when G is acyclic, $STR(p,q) = \{Str(p,q)\}$ if there is a path from p to q, and $\{\}$ otherwise.

We again define $Strings$ so that, for any node p, $Strings(p)$ is the set of all strings on paths from p to a leaf; i.e.

$$Strings(p) \mathrel{\widehat{=}} \{\, w \mid \exists q \in Nodes(G) \bullet IsLeaf(q) \ \wedge \ w \in STR(p,q) \,\}$$

Thus, $Strings(Root(G))$ is the set of strings represented by a graph G.

We will represent L using a graph G, such that $Strings(Root(G)) = L$. Observe that for each $w \in Prefixes(L)$, there is a unique node p in G such that $w \in Str(Root(T),p)$ and $Strings(p) = Rem(w,L)$. Thus, we can represent C by a node in G. We also observe that if $Strings(p) = C$, then $Hds(C) = OutLabels(p)$ and $Rem(a,C) = Strings(Succ(a,p))$ for all $a \in Hds(C)$.

Thus, we can represent L by $Root(G)$, and C by a node in G, indicated by a variable p. The coupling invariant is now:

$$L = Strings(Root(G)) \ \wedge \ p \in Nodes(G) \ \wedge \ C = Strings(p)$$

The loop invariant becomes:

$$\begin{array}{ll} & 1 \le k \le n \ \wedge \ s \in Strings(Root(G)) \ \equiv \ s[k\,..\,] \in Strings(p) \\ \iff & 1 \le k \le n \ \wedge \ s[1\,..\,k-1] \in STR(Root(G),p) \end{array}$$

The structure we have just described is, of course, a deterministic finite acceptor: Nodes correspond to states, edges correspond to transitions, the root is the start state, and the leaf node is the single accepting state.

This representation exists whenever L is regular. Because L is regular, the resulting graph has a finite number of nodes. This is a direct consequence of the Myhill-Nerode Theorem (see, for example, [10] or [11]). Since the graph is finite, we can implement it and the required operations in any of the standard ways.

The algorithm resulting from this data refinement is essentially the same as that for finite languages — the only difference is that the data structure may now contain cycles and it uses operations on graphs rather than on trees. This difference affects the loop invariant and the reasoning required to justify the data refinement, but does not affect the resulting code.

It is interesting to see how the DFA has emerged as a way of representing L to provide an efficient computation of Hds and Rem, with no *a priori* idea of state: States emerge by precomputing candidate completion sets. The similarity with the algorithm for finite languages highlights the fact that a trie is just a DFA whose graph is a tree. Applying the techniques of this section to a finite language would give us a directed acyclic graph rather than a tree.

Once again, in initialising C, we have assumed that the required data structure (in this case G) is available. Again, this might be provided, or might be obtained by a precomputation performing an explicit data mapping, e.g. from a regular expression or regular grammar describing L to the required DFA.

7 Specialisation where L is a Context Free Language

Suppose that L is a non-empty Context Free Language, described by a grammar G. As usual, G is a 4-tuple (V_N, V_T, S, P) giving nonterminals, terminals, start symbol and productions (see, for example, [10] or [11]); since $L \subseteq \mathcal{A}^*$, we can assume $V_T = \mathcal{A}$. We specialise the initial specification with the additional assumption $L \neq \{\} \wedge L = Lang(G)$, where $Lang(G)$ is the language defined by G. Knowing that not all context free languages can be recognised deterministically, we should expect to make some further assumptions about L. Rather than making such assumptions now, we will investigate one way of representing C when L is context free, and then consider the class of languages for which it can be used.

Again, consider how we might represent C. As in the regular case, constructing a tree to represent $Hds(C)$ leads (in general) to an infinite tree. This time, however, we can't collapse the tree into a finite graph. Thus, the representation of C requires more information about the form of acceptable completions. One approach is to encode the set of possible completions using a string of terminal and/or nonterminal symbols.

For any string $\sigma \in V^+$, where $V = V_N \cup V_T$, we define $Strings(\sigma)$ to be the set of all terminal strings that can be produced from σ. We can define $Strings(\sigma)$ in terms of the usual "produces" relation ($\Rightarrow_G^*$):

$$Strings(\sigma) \mathrel{\hat{=}} \{\, w \in V_T^+ \mid \sigma \;\Rightarrow_G^*\; w \,\}$$

Alternatively, we can define $Strings(\sigma)$ recursively:

$$Strings(\sigma) \mathrel{\hat{=}} \begin{cases} \{\sigma\} & \text{if } \sigma \in V_T^+ \\ \bigcup_{\sigma' \in \Sigma(\sigma)} Strings(\sigma') & \text{otherwise} \end{cases}$$

where $\Sigma(\sigma)$ is the set of strings (over V^+) that can be obtained by replacing one nonterminal symbol (N) in σ by the right hand side (α) of a rule in P with N as its left hand side; i.e.

$$\begin{aligned} \Sigma(\sigma) = \{\, \sigma' \mid \exists N \in V_N, \alpha, \phi, \psi \in V^* \bullet \\ \sigma = \phi N \psi \;\wedge\; \sigma' = \phi \alpha \psi \;\wedge\; N \;\rightarrow\; \alpha \in P \,\} \end{aligned}$$

Clearly, L can be represented by the start symbol S; we will endeavour to also represent C by some string σ over V. Thus, the coupling invariant will be:

$$L = Strings(S) \;\wedge\; \sigma \in V^+ \;\wedge\; C = Strings(\sigma)$$

The loop invariant becomes:

$$\begin{array}{ll} & 1 \leq k \leq n \;\wedge\; s \in Strings(S) \;\equiv\; s[k\,..] \in Strings(\sigma) \\ \iff & 1 \leq k \leq n \;\wedge\; S \;\Rightarrow_G^*\; s[1\,..\,k-1] \frown \sigma \end{array}$$

The last conjunct says that $s[1\,..\,k-1] \frown \sigma$ is a *sentential form.*

With this representation, we can compute $Hds(C)$ using the function $First$ defined, for any non-empty string $\gamma \in V^+$ as follows:

$$First(\gamma) \mathrel{\hat{=}} \begin{cases} \{\gamma[1]\} & \text{if } \gamma[1] \in V_T \\ \bigcup_{\gamma' \in \Gamma(\gamma[1])} First(\gamma') & \text{if } \gamma[1] \in V_N \end{cases}$$

where, for any nonterminal N, $\Gamma(N)$ is the set of right hand sides of rules in P that have N as their left hand side; i.e.

$$\Gamma(N) \mathrel{\widehat{=}} \{\, \alpha \mid N \rightarrow \alpha \in P \,\}$$

This definition of $First$ does not accommodate rules with empty right hand sides; thus we have an additional assumption about G, that the right hand sides of all rules in P are non-empty (i.e. G has no λ-productions)[9].

Now we have $Hds(C) = First(\sigma)$, which we can simplify to $First(\sigma[1])$. Computing $Rem(s[k], C)$ is not so simple; we will address it shortly.

We can now data refine the generic algorithm using this data representation, as follows:

$$\begin{array}{rcl}
C := L & \equiv & C\!:\![\, C = L \,] \\
& \preceq & \sigma\!:\![\, Strings(\sigma) = Strings(S) \,] \\
& \sqsubseteq & \sigma\!:\![\, \sigma = S \,] \\
& \sqsubseteq & \sigma := S
\end{array}$$

$$x \in Hds(C) \quad \preceq \quad x \in First(\sigma[1])$$

$$\begin{array}{rcl}
C := Rem(s[k], C) & \equiv & C\!:\![\, C = Rem(s[k], C_0) \,] \\
& \preceq & \sigma\!:\![\, Strings(\sigma) = Rem(s[k], Strings(\sigma_0)) \,] \quad (i)
\end{array}$$

Rather than compute $Rem(s[k], Strings(\sigma))$ directly, we will manipulate σ into a form which simplifies the computation: the required form is that the first symbol in σ is a terminal symbol. In modifying σ, we may reduce $Strings(\sigma)$, so long as we still have $s[k] \in Hds(C)$, i.e. $s[k] \in First(\sigma)$, which comes from the loop guard. Thus, we refine (i) as follows:

$$\begin{array}{rcll}
(i) & \sqsubseteq & \sigma : \left[\, s[k] \in First(\sigma) \;,\; \begin{array}{l} s[k] \in First(\sigma) \;\wedge \\ Strings(\sigma) \subseteq \\ \quad Strings(\sigma_0) \;\wedge \\ \sigma[1] \in V_T \end{array} \,\right]; & (ii) \\
& & \sigma : \left[\, \begin{array}{l} s[k] \in First(\sigma) \;\wedge \\ Strings(\sigma) \subseteq \\ \quad Strings(\sigma_0) \;\wedge \\ \sigma[1] \in V_T \end{array} \;,\; \begin{array}{l} Strings(\sigma) = \\ \quad Rem(s[k], Strings(\sigma_0)) \end{array} \,\right] & (iii)
\end{array}$$

If $\sigma[1]$ is not a terminal symbol ($\sigma[1] \notin V_T$), it must be a nonterminal ($\sigma[1] \in V_N$). We will manipulate σ into a form where $\sigma[1]$ is a terminal symbol by successively replacing $\sigma[1]$ by the right-hand side of some rule defining $\sigma[1]$.

$$\begin{array}{rcll}
(ii) & \sqsubseteq & \mathbf{do}\ \sigma[1] \in V_N \ \rightarrow & \\
& & \quad \alpha\!:\![\, \sigma[1] \rightarrow \alpha \in P \,]; & (iv) \\
& & \quad \sigma := \alpha \frown \sigma[2\,..\,] & \\
& & \mathbf{od}; &
\end{array}$$

The loop invariant is $Strings(\sigma) \subseteq Strings(\sigma_0) \;\wedge\; s[k] \in First(\sigma)$, where σ_0 is the value of σ at the beginning of the loop. The variant is the length of the longest derivation of a string beginning with $s[k]$ from σ. We must now show that this variant is reduced by the loop body, and find a way to implement (iv) deterministically.

[9] It is straightforward to relax this restriction — we just need a slightly more complex definition for $First$ — but we prefer to avoid the additional complexity here.

In order to implement (iv) deterministically, we must be able to determine which rule to select, for a given $s[k]$ and $\sigma[1]$. This will be the case precisely when G is an LL(1) grammar. We thus add a further assumption: that L is an LL(1) language and G is an LL(1) grammar. We will implement (iv) using a new function $RightRule$ such that, for any terminal symbol a and nonterminal symbol N, such that $a \in First(N)$, $RightRule(a, N)$ is the right-hand side of the first rule to be used in producing a terminal string starting with a from N; i.e.

$$RightRule(a, N) = \alpha \;\equiv\; N \;\rightarrow\; \alpha \in P \;\wedge\; a \in First(\alpha)$$

This value is unique for an LL(1) grammar, since by definition, whenever $N \rightarrow \alpha_1 \in P$ and $N \rightarrow \alpha_2 \in P$, where $\alpha_1 \neq \alpha_2$, $First(\alpha_1) \cap First(\alpha_2) = \{\}$.

Thus, we have:

$$(iv) \quad \sqsubseteq \quad \alpha := RightRule(s[k], \sigma[1])$$

Assuming that G is an LL(1) grammar ensures that $RightRule(s[k], \sigma[1])$ is well defined when $s[k] \in First(\sigma)$, which is part of the (inner) loop invariant. This assumption also ensures that the variant is reduced, and thus that the loop terminates.

It now remains to implement (iii), which is easy. When $\sigma[1] \in V_T$, $First(\sigma) = \{\sigma[1]\}$. But we also have $s[k] \in First(\sigma)$, so we have we must have $s[k] = \sigma[1]$. Thus, we have:

$$(iii) \quad \sqsubseteq \quad \sigma := \sigma[2\,..\,]$$

The resulting algorithm after performing this data refinement is:

$$\begin{array}{l}
|[\ \mathbf{var}\ k, \sigma \ \bullet \\
\quad k := 1 \ \|\ \sigma := S; \\
\quad \mathbf{do}\ s[k] \neq \$ \ \wedge\ s[k] \in First(\sigma[1]) \ \rightarrow \\
\qquad \mathbf{do}\ \sigma[1] \in V_N \ \rightarrow \\
\qquad\quad \alpha := RightRule(s[k], \sigma[1]); \\
\qquad\quad \sigma := \alpha \frown \sigma[2\,..\,] \\
\qquad \mathbf{od}; \\
\qquad \sigma := \sigma[2\,..\,]; \\
\qquad k := k + 1 \\
\quad \mathbf{od}; \\
\quad r := s[k] = \$ \ \wedge\ \$ \in First(\sigma[1]) \\
]|
\end{array}$$

The algorithm we have arrived at is a table-driven LL(1) parser or "predictive" parser, in which σ acts as a stack: The operation $\sigma := \alpha \frown \sigma[2\,..\,]$ pushes the symbols in α onto the top of the stack, while $\sigma := \sigma[2\,..\,]$ pops the stack. It remains to choose suitable implementations for $First$ and $RightRule$, most likely as tables, and for the stack; but these details need not concern us here.

The main difference between this algorithm and the usual predictive parsing algorithm is the presence of the inner loop. This is required because of the loop invariant and variant we used, which requires k to be increased by one on each iteration. The usual predictive parsing algorithm (e.g. [2], p187) uses a single loop which, on each iteration, either increases k and pops the stack (after checking that the symbol at the top of the stack is $s[k]$) or replaces a nonterminal at the top of the stack. Our version, however, makes the operation of the algorithm clearer and the proof of termination easier: for each input symbol we successively reduce the nonterminal at the top of the stack until we get that terminal at the top, then remove it. Another difference is that the usual version doesn't need to test for $s[k] \in First(\sigma[1])$; if s is not a sentence, it will detect this either when the top of stack is a terminal not equal to $s[k]$ or when $RightRule(s[k], \sigma[1])$ is undefined. We could obtain the standard algorithm by using a different variant for the main loop (though this would destroy the symmetry with our other derivations), or by transforming the above version to combine the two loops.

Again, we have assumed that the language is given in the required form, which in this case means that the tables $First$ and $RightRule$ are already available. Constructing these tables will normally be done as a by-product of data refining the surrounding envorinment; they may be input in the required from or computed from the grammar.

8 Conclusions

We have illustrated the use of data refinement and program specialisation in deriving a family of language recognition algorithms. This development provides derivations for these algorithms which we believe to be more intelligible that would have otherwise been obtained using program refinement. Considerable care was taken over the loop structure in the abstract algorithm, and the final form was not settled until after a couple of specialisations had been worked through, which helped to identify the important concepts in the abstract algorithm. The resulting algorithm then led to elegant loop structures in all the concrete algorithms derived.

The development also highlights some common aspects of these algorithms. In particular, it provides a uniform explanation of use of tries, DFAs and stacks, respectively, in terms of representing the set of candidate completions at each stage. This, in turn, suggests ways in which other related algorithms could be derived. Using an approach more like that in Section 7, we could build a recogniser for regular languages by representing C as a regular expression, and defining suitable functions on regular expressions to implement Hds and Rem; alternatively, we could further specialise the representation used in Section 7 for the case where the grammar is regular — in which case the grammar tables end up being isomorphic to the graph operations of the DFA. If we had chosen a different representation for C in Section 7, we could have obtained an algorithm for some other class of context free languages; for example, it should be just as easy to obtain an LR(1) parser — giving a very different derivation from the one given in [9]. By using a somewhat more elaborate data structure (i.e. one carrying several alternative parses in parallel) we should be able to obtain Earley's algorithm.

The same approach can be applied to obtain a number of other related algorithms, where the language is defined in different ways. We can obtain a recogniser for a language defined by simple patterns consisting of symbols and symbol variables, by adding a binding table to the representation used in Section 4. If we let L be the set of all substrings of a string t, this approach should lead us to the Knuth-Morris-Pratt string searching algorithm.

References

[1] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] R. J. R. Back. *Data Refinement in the Refinement Calculus.* Technical Report Series A, No. 68, Inst. för Informationsbehandling, Åbo Akademi, Turku, Finland, 1988.

[4] Lucy Chubb. *A Data Refinement Model of Interfaces.* University of New South Wales, School of Computer Science and Engineering Report No. 9204, August, 1992.

[5] Rick Decker. *Data Structures.* Prentice-Hall, 1989.

[6] P. H. B. Gardiner and C. C. Morgan. "Data refinement of predicate transformers". *Theoretical Computer Science* **87** (1991), pp143–162.

[7] A. Gravell. "Specialising Abstract Programs". *Proc. 4th Refinement Workshop*, J. M. Morris & R. C. Shaw (Eds), Springer-Verlag, 1991, pp34–50.

[8] Lindsay Groves. "Deriving Sorting Algorithms using Data Refinement". *Proc. 16th Australian Computer Science Conference*, 1993, pp523–534.

[9] Wim H. Hesselink. "LR-parsing derived". Science of Computer Programming **19** (1992), pp171–196.

[10] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[11] John C. Martin. *Introduction to Languages and the Theory of Compuation.* McGraw-Hill, 1991.

[12] C. C. Morgan. "Data refinement using miracles". *Inf. Proc. Lett.* **26**, 5 (Jan. 1988), pp243–246.

[13] Carroll Morgan. "Auxiliary variables in data refinement". *Information Processing Letters* **29** (1988), pp293–296. Also in [16], pp79–85.

[14] Carroll Morgan. "Types and invariants in the refinement calculus". *Mathematics of Program Construction*, J. L. A. van de Snepscheut (Ed.), Lecture Notes in Computer Science, Vol. 375, Springer-Verlag, 1989.

[15] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[16] Carroll Morgan, Ken Robinson and Paul Gardiner. *On the Refinement Calculus*, Technical Monograph PRG-70, Oxfrod University, 1988.

[17] J. M. Morris. "Laws of data refinement". *Acta Informatica* **26** (1989), pp287–308.

[18] Raymond G. Nickson and Lindsay J. Groves. "Metavariables and Conditional Refinements in the Refinement Calculus". *Proc. 6th BCS Refinement Workshop*, 1994.