Chapter 9

Conclusions and Recommendations

In this project, our aim has been to develop software tools and formal notations that facilitate software development in the FORTH programming environment with a particular interest in "Real-Time" and safety critical systems. In this chapter, we provide an overview of the project, linking the various parts, giving comments on the work and making suggestions for possible further work.

9.1 Introduction

Our initial areas of investigation were:

- To provide a method of communicating between multiple FORTH-based RISC systems.
- A version of the FORTH++ compiler to operate with the Harris RTX–2000 stack based RISC processor. This compiler also includes an extensive native-code optimisation technique.
- An interface between FORTH and a local area network. This is to provide a multi-platform development environment. The interface can also be used to pass messages between different (possibly remote) nodes.
- The interfaces between FORTH and other high-level languages, allowing the developer the freedom to interface with supplier propriety code. This was done in such a way that FORTH's interactive user interface is maintained and may be used to enhance the original (non interactive) system.

We have been able to address some of these problems directly. We have described these in various sections: Communicating Novix NC4016s (Appendix A); FORTH++ and the Harris RTX–2000 (Appendix B); Using IBM's NETBIOS from FORTH (Chapter 2); Mixed Language Interface (Chapter 3 and Appendix C).

Many aspects of our investigations proved to be dependent on a more thorough theoretical underpinning of the FORTH language. Thus our attention moved to providing such an underpinning. This work mainly consisted of:

- We looked at how formalisms might be used to define a semantic model of the FORTH language. We have provided a formal base from which the semantic meaning of a program can be derived (Chapter 4).

- In defining our formal base we found a relationship between the stack based virtual machine and a register based target processor. We investigated this relationship, developing a single pass optimisation method (Chapter 5).

- When using our formal definition we found that the parameter stack was a source of uncertainty. The stack is type free, yet to formally define the meaning of a routine we must know the type of the stack items. We therefore investigated the possibility of developing a typed version of FORTH, developing a "type signature algebra" that allows us to type check the FORTH parameter stack (Chapter 6).

  Having defined a type signature algebra we then investigated how this may be implemented and what effect it may have on a program (Chapter 7).

- In order to support the multi-tasking capabilities of FORTH we developed a (formal) theory of concurrent tasks based on state machines that synchronise on events.

  As this system is to be used by people unfamiliar with formal notations we have provided a graphical representation of the theory (Chapter 8).

9.2 Networks

We investigated the IBM NETBIOS system as a standard for interfacing with LANs. The NETBIOS system is designed to operate in parallel with any application. Chapter 2 describes an interface that we have developed to exploit this ability. We have also developed several demonstration programs to show how this system can be used to pass messages from one IBM PC to another.

9.3 Mixed Languages Interface

Our mixed languages interface is an interface between the FORTH programming environment and another programming language. The system described in Chapter 3 is designed to interface with code written in the C language, however, the method is largely language independent thus allowing the developer to interface with code written in any language. Although the system was written with the Microsoft C compiler in mind, it was developed using Borland's Turbo C compiler. We have tested the "portability" of this system by compiling the same code under a number of different compilers.

This interface has shown how we can tap into the ever growing pool of application libraries available for C (and other languages), thus reducing the maintenance and manufacturing costs of our system while providing more functionality.

We have seen one implementation of the general method of integrating these language features. We can see how the general methods can be used to extend the FORTH system into areas that were out of reach. This system would allow us to indulge in hiding of information that the FORTH system has no requirement to access in the C system, such as the floating point stack in Appendix C.

There are several ways in which this work could be advanced. It should be possible to provide a FORTH to C interface on a remote target system or a Pascal or even an Ada interface based on these ideas. It would also be possible to apply this interface to other FORTH systems with relative ease.

9.4 Formal FORTH

Several aspects of our investigations appear to be dependent on a more thorough theoretical underpinning of the language. Thus we moved our attentions to providing such support. Due to the nature of the FORTH abstract machine it is possible to provide a set of tools that will aid a designer/programmer

in ascertaining whether a FORTH program meets its (formal) specification. Chapter 4 shows some preliminary investigations in to how formal descriptions can be applied to parts of the FORTH programming environment.

9.5 Stack Optimisation

We investigated the relationship between the FORTH virtual stack machine and a register based host processor. In developing our formal support (Chapter 4) we modelled the stack as a sequence of untyped elements. This has lead to the development of a compiler optimisation technique based on this concept. Chapter 5 reviews the more common optimisation techniques and develop our own "stack registers" technique. This technique keeps the top three elements of the stack in internal registers. Unlike traditional systems that keep the items in specific registers, our system allocates the internal registers *dynamically*. We use a relationship between the top elements of the stack and their allocated register to produce a *stack image*. Using this technique it is possible to obtain 100% optimisation on certain stack manipulation operations (such as `SWAP`, `ROT` and `DROP`).

By combining the most common existing techniques (*inline compilation*, and *peep-hole optimisation*) with our *stack registers* we can provide a very powerful optimising compiler.

9.6 Type Algebra

Traditionally the FORTH parameter stack has always been typeless. This allows the programmer to "cast" data items without recourse to inefficient and unnecessary subroutines (as in C). The majority of FORTH programmers use this ability (either directly or indirectly) in almost all applications. Yet it carries with it the problem that the programmer must keep track of the logical type of all the items he is using. It would be possible to fetch an *integer* from a variable, thinking that it was an *execution-token*, causing the system to crash when this *pseudo execution-token* was executed.

In developing our formal support we found that we had to leave the stack as a sequence of untyped elements. Although we could reason about the logic of a program, we need to know the stack types to be able to check if a program is "correct" or not. For this we would need to provide some form of typing mechanism.

Jaanus Pöial of Tartu University has developed a "Stack Type Algebra" (Pöial 1990) that provides the capability of checking the type of stack elements. Chapter 6 presents a new type algebra based on these ideas. The algebra includes the capability of handling items of variable type in addition to catering for program structures and conditional execution.

Using our type algebra, it would be possible to say that a program handles its stack correctly. It is not possible to say that the program will perform as required. By combining the type algebra with the formal toolset, given in Chapter 4, it should be possible to formally prove that a FORTH program has a given property. In this way, we can prove that the FORTH program has the same properties as the specification and is thus a true implementation of that specification.

9.7 FORTH Type Checker

Having developed our "type algebra" we investigated the possibility of implementing a FORTH type checker based on these ideas. Chapter 7 presents an initial specification for such a program, referred to as "FLINT". The specification also allows the program to check for a number of additional errors (such as stack underflow) and various software metrics in addition to its type checking rôle. Such a system will stop the abuse of the typeless stack whilst maintaining the flexibility of a type free stack.

Although we have specified a possible type checker and some thought to its initial design has been incorporated in this specification, we have not, as yet, developed the system. A commercial vendor has shown an interest in this work and may develop an implementation.

9.8 The Event Calculus

The "Event Calculus" is a diagrammatic notation that provides an easily used means of formally specifying the behaviour of concurrent systems. It can describe synchronous and asynchronous communications, data flow modelling and function application, in addition to temporal constraints. It also abbreviates the description of complex state changes such as data base updates via the use of formal notations. This gives the designer a good level of control over the levels of abstraction used.

Chapter 8 introduces the Calculus in a tutorial like manner, initially introducing the notation then building up its functionality one step at a time. The formal definition of the Calculus is given alongside an informal introduction to the notation.

We have found that the diagrammatic nature of the Calculus makes it relatively easy for non specialist to use when compared to other (event based) process algebras such as CSP, CCS and LOTOS. As a specification produced using the Calculus has an underlying formal specification, it is possible to use this specification for deriving proofs of the system begin modelled.

The Calculus is used to not only represent the state transitions in one system but also the interaction between different state machines. There is a very simple correlation between a state machine as represented in the Calculus and a FORTH task. Indeed, the Calculus appears to be an effective mechanism for breaking down a complex problem into a multi-tasking implementation.

9.9 Future Directions

In this section we describe how the work presented in this document could be extended. We have identified three areas of interest that we feel worthy of further development.

9.9.1 Type Algebra

The type algebra can be developed further:

- The rules for handling variable types (rules 5–8) are probably not required, they can be derived from the matching rules (rules 1–3) and the reduction rule (rule 4).
- The "Multiple Signature" (+) and "Alternative Type" (|) capabilities are the only way subtypes can be expressed at current. The provision of a types hierarchy should be investigated.
- The implementation of the algebra in some form of automatic tool, such as FLINT, or by incorporating the algebra into an existing compiler.

9.9.2 Formal FORTH

It is our intention to provide a secure FORTH programming environment. We intend to produce such a system on a Windows based workstation, such as a Sun (Sparc station) or an IBM PC under Windows. This system should be a formally specified standard FORTH implementation[1]. The formal specification, probably written in Z, will be based on the preliminary "Formal FORTH" work.

We would like to include type checking facilities based on the type algebra, similar to those prescribed for the FLINT program. In time we would like to extend this system to include a formalised variant of the stack registers optimisation technique.

[1] Probably the ANSI Standard, when it is released.

9.9.3 Event Calculus

The interface between an event and a Z Schema is perhaps more complicated than is necessary. This interface should be investigated, a new less complex one should be provided if possible.

The Event Calculus should be used to generate some example specifications. Taking a number of case studies from initial specification, through to final implementation (in FORTH), complete with corresponding proofs. This will also provide us with a less formal introduction to the Calculus.

Such examples could be used to introduce the FORTH community to formal methods whilst the Event Calculus could be used to introduce the formal methods community to the FORTH language.

9.10 References

Pöial, J. (1990). The algebraic specification of stack effects for FORTH programs. In *Proc. FORML Conf., Proc. EuroFORML Conf.*, San Carlos, CA. FORTH Interest Group.