Towards testability in smart card operating system design

Pieter H. Hartel * Eduard K. de Jong Frz †

Abstract

The operating system of a smart card is a safety critical system. Distributed in millions, smart cards with their small 8-bit CPU support applications where transferred values are only protected by the strength of a cryptographic protocol. This strength goes no further than the implementation of the software in the card and terminal allows. Because of its complexity, to guarantee absolute reliability of the smart card software is prohibitively expensive. Obtaining a high level of confidence in the implementation of a smart card application is essential for their widespread acceptance. A highly structured design of the smart card operating system gives the designer control over the complexity of the system.

A functional language has been used to prototype a smart card operating system. The prototype has the same structure as the real operating system and it offers most of the functionality of the real system. The well defined semantics of pure functional languages and their compositionality in particular are instrumental to the structuring of the prototype. With the functional language implementation as reference, the reliability of the implementation can be assessed in detail.

Key words: prototyping, smart card operating system, defensive data types, design for testability, functional programming.

1 Introduction

A smart card is a complete computer housed in a piece of plastic the same size as a credit card [17]. In most systems, the computer communicates with a smart card terminal via half a dozen contacts on the surface of the card. Some of these contacts supply the power to the computer.

A smart card computer has to be small to reduce the risk of mechanical problems. Because of these mechanical constraints, as well as aspects of cost, the current generation of smart cards typically contains only a small 8-bit micro processor, a few hundred bytes of RAM, a few Kbytes of ROM and a few Kbytes of EEPROM. This small size constrains the freedom in the design of the software that has to be run on a smart card processor, and als places restrictions on the development methods that can be used.

Smart cards are applied in many different areas including transport, banking, retail, health-care, logistics, telecommunication, identification and security.

A smart card operating system provides the processing and storage facilities that enable an application provider to implement specific card applications with flexibility and ease. Current smart card operating systems are modelled on DOS-like file systems, with files and directories [11, 14, 18]. A set of commands interpreted by the operating system provides for basic data access operations. Security is supported by access conditions associated with files and directories. Cryptographic protocols [5, 8] are used to satisfy access conditions. Smart card operating systems based on the file system model require lengthy sequences of commands to retrieve or modify a data element on the card. The smart card operating system described here follows different principles. Our operating system presents the smart card as a processing element that communicates with a terminal in client-server fashion. Using this operating system, applications are designed as objects containing persistent data, access methods and security procedures [12]. Our system offers a powerful and secure instruction set that requires a few, short commands to implement a transaction [6].

In a typical application, a smart card is entrusted with powers of considerable economic value. To protect these economic values, the software on the card uses secure cryptographic protocols. In an

*Department of Computer Systems, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands, Email: pieter@fwi.uva.nl

†QC Consultancy v.o.f., Ankersmidplein 63, 1506 CK Zaandam, The Netherlands, Email: eduard@q2c.nl

ideal situation the implementation should be proven correct so that the security of the system cannot be compromised. A structured formal approach to assessing the correctness of an implementation is feasible for smart card applications which are designed from scratch, including the design of any dedicated operating system functions needed. Correctness proofs for such implementations are expensive and require considerable time. There are a number of reasons for this. Firstly, the predominant use of assembly language to achieve a compact encoding of the system does not lend itself to formal reasoning. Secondly, the communication patterns between card and terminal, which are essentially asynchronous devices, may be arbitrarily complex. Thirdly, the different applications on a multi-application card may interact in complex ways. Fourthly, all formal methods for reasoning about the correctness of implementations require assumptions to be made about the behaviour of the underlying hardware. As a result, we follow a more pragmatic approach and strive to reason about an implementation *with a high level of confidence.*

To achieve a high level of confidence, while meeting the physical constraints, the designer of a smart card operating system has to find recourse to a carefully considered hierarchical design. At each layer in the hierarchy, functionality is added. This functionality builds on that provided by lower layers of the hierarchy. The strict hierarchy makes it possible to reason in relatively simple terms about the behaviour of lower layers, while trying to manage the complexity of the current layer. The behaviour of the system as a whole requires extensive testing, so support for the generation of testing procedures must be provided as part of the system design.

A layered design is often built at the expense of execution speed. For smart card systems this is not a problem, as for most smart card applications a modest overall execution speed is acceptable. A road pricing system, where transactions must be handled while a fast moving vehicle is within range of a radio link is perhaps an exception.

Computer architects and software engineers have built layered systems for many years [15]. The experience gained with this method applies to smart card systems as to any other system. What makes smart card systems different from general purpose computer system is the pervasive demand for secure operations. It is our belief that security has to be considered at all layers of the system and that security has to be taken into account right from the start of the design process.

In the next section we discuss the implementation strategy that has been followed to achieve our goals. Section 3 introduces the overall structure of the prototyped layered smart card operating system. The "secure instruction interpreter" level, which describes the operational behaviour of the smart card is presented in some detail in Section 4. Conclusions and future work are discussed in the last section. In the appendix a brief introduction is presented to the functional programming language Miranda [1] [16] used for prototyping.

2 Implementation strategy

Having discussed some of the fundamental issues in smart card operating design, we now turn to the method that we have used to realise the design.

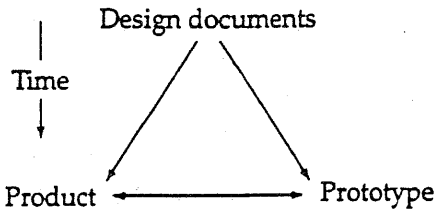

Figure 1: Simultaneous prototype and product development strategy. The nodes represent elements of the strategy. The arrows indicate the main direction of information flow.

Our approach differs from the conventional approach, in that the construction of the prototype and the construction of the product were started *at the same time.* The construction of the prototype was completed well before that of the product. The motivation for this approach stems partly from the need to develop a comprehensive testing strategy for the finished product [1, 13]. Some researchers point

[1] Miranda is a trademark of Research Software Ltd.

out that building prototypes even after completing a product may be useful [2] to construct testing procedures.

Our strategy is schematically represented in Figure 1, where the elements of the strategy are shown as *Design documents*, *Product* and *Prototype*. The arrows indicate the main direction of information flow from one element to another. The prototype and the product implement essentially the same functionality, using the same set of design documents. The two systems are different for a number of reasons. Firstly, the systems were built using different paradigms. Secondly, the systems are targeting very different machines. Thirdly, the systems were built by different people. Fourthly, the systems were built in different ways: the prototype was built top-down, and the product was built using a mixture of detailed top-down and bottom-up design steps. Because of these differences, the product and the prototype are unlikely to contain the same mistakes – unless the mistakes were also in the design documents.

The interactions between the developers of the product on the one hand and of the prototype on the other proved fruitful. Details left unspecified in the design were discussed in a number of sessions with the designer and the two developers present. Some inconsistencies in the design were found early in the prototype. Valuable development time was saved because of this. These advantages have to be balanced against the cost of building the prototype.

At first the prototyping, and later the product development showed no major problems in the basic design. So it seems that the prototyping only helped to avoid some relatively minor problems. If on the other hand the design had turned out to have major problems, the prototyping would have discovered these well before the product development (The prototype was completed well before the product). Since prototyping is cheaper than product development, rebuilding the prototype from scratch after a redesign would have been cheaper than redeveloping the product.

Throughout our experiment, we found that the top-down development style enforced by the use of a purely functional language [10] made it essential to understand fully the structure of the design from the first. This structure is clearly visible in the prototype as a hierarchy of interrelated data structures. Building the smart card operating system around these data structures allows for a highly modular development.

Having outlined our development methodology we now return to the design of the smart card operating system proper.

3 A layered smart card operating system

In the general hierarchical machine model [15], several identifiable levels are present. Moving from a lower to a higher level in the hierarchy means that more and more powerful (and specialised) instructions will be encountered, taking arguments with an increasing amount of structure. The encoding of the instructions also becomes more dense as we move up in the hierarchy.

A multi-application smart card system with a good security mechanism is of a sufficient complexity to warrant the use of a hierarchical construction method. Four different levels in interpretation are distinguished in our design, as shown in Figure 2. Working from the top downwards, we have:

Application command interpreter This is the top layer. It is logically part of a smart card operating system, but its implementation is physically separated from the smart card: the application command interpreter normally resides in the smart card terminal. Through a dialogue with the smart card owner, the application command interpreter issues instructions to the secure instruction interpreter on the smart card itself.

Secure instruction interpreter The smart card offers a small repertoire of instructions, mainly aimed at reading, writing and updating information held by the smart card [7]. The major purpose of providing this particular level in the smart card operating system is to provide an interface for high level security features, such as authentication of the terminal and the owner and encryption and decryption of information. The secure instruction interpreter is implemented as a section of "compact codes", which resides partly in ROM, partly in EEPROM.

Compact code interpreter The ROM code of a smart card processor contains a small program, which interprets a section of compact codes. The interpreter implements a specialised instruction set, which provides basic smart card operations as well as basic security features. These include read or write protection on blocks of storage. Storage at this level is organised into small blocks of

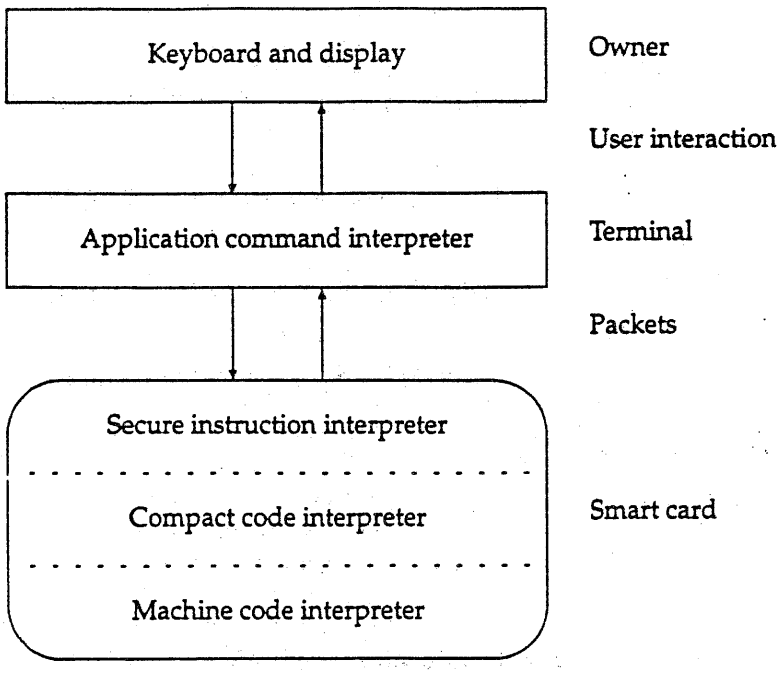

Figure 2: The layered smart card operating system. The oval represents the smart card, the boxes represents the terminal and the keyboard and display. The arrows represent the exchange of messages and the dotted lines represent interfaces between software layers.

memory, with an associated security status. The compact code interpreter is located in ROM, so it cannot be changed.

Machine code interpreter This is the hardware of the smart card, including the RAM, ROM and EEPROM. The hardware does not provide security features, so any routine operating at this level has unrestricted access to all information. The only routines written directly in machine code are the compact code interpreter and a few cryptographic routines, which would be too slow if implemented at the compact code interpreter level.

In the remainder of the paper, an overview is presented of the secure instruction interpreter.

4 Secure instruction interpreter level

The secure instruction interpreter of the multi-application smart card operating system is itself organised as a hierarchy that controls access to the information stored on the card. A schematic representation of this hierarchy is presented in Figure 3. It shows that a multi-application card carries a number of Card System Applications (CSAs), which in turn hold a number of Card Terminal Applications (CTAs).

For example, the same card may be used to carry a travel ticket as well as an electronic purse to pay for the ticket. The electronic purse application may be issued by a bank, while a public transport company will issue the travel ticket application. The two CSAs involved can be considered as encapsulations of information and suitable access routines.

For each different use of a CSA, a specialised CTA is required. The electronic purse application, requires a CTA to be used by the bank (to load the purse), and a different CTA to be used by the owner (for making payments using a point of sale terminal). The CTA used by the bank is not accessible to the card owner, who would otherwise be able to create money. A CTA is a description of security requirements, permitted operations on data and associated access rights.

In the following sections, we will discuss the working of a multi-application smart card prototype implementation in terms of:

1. the permanent, internal data structures as maintained by the card;

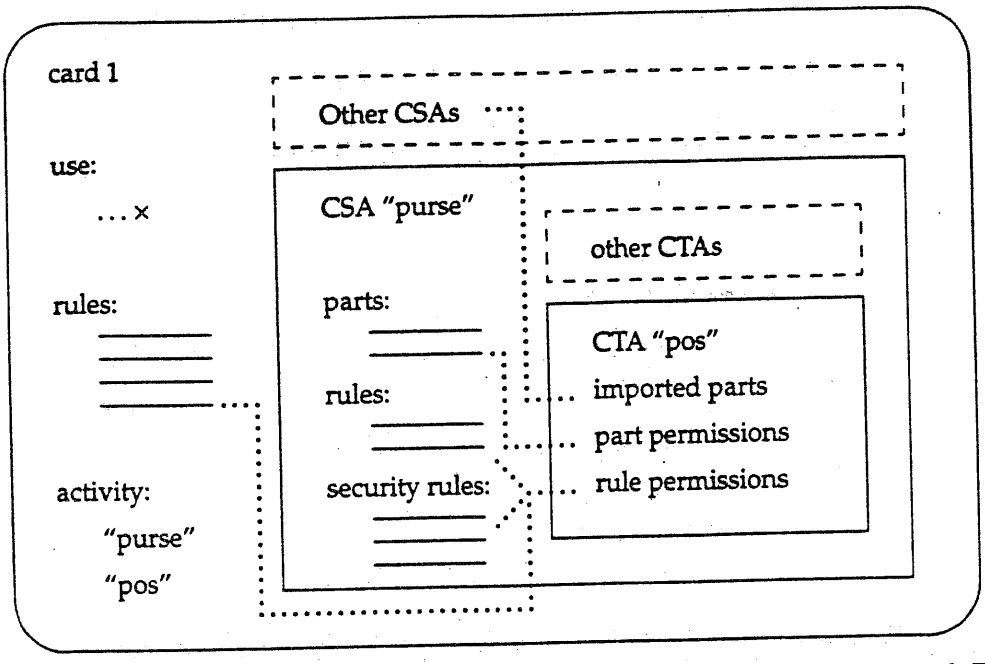

Figure 3: The structure of the secure instruction interpreter of a multi-application smart card. This shows the status of the interpreter when activated by a point of sale terminal (pos). The dotted lines are references from the CTA data structures to parts and rules contained in the CSA data structure and references to the rules of the card in which the CTA data structure is embedded.

2. the transient, input and output data as exchanged between the card and the terminal;
3. the secure instruction interpreter;
4. a dialogue between a smart card and a terminal.

The discussion will be concluded with an investigation of the liveness property [3] of the prototype implementation.

4.1 Internal data

The various substructures present in figure 3 will now be introduced in a top-down fashion. In the next section, the security data types `card`, `csa` and `cta` are described. These data types represent layers in the design hierarchy, which provide security and an operational framework. They do not provide storage of data or operations. These aspects are provided by two separate structures: `part` and `rule`.

Defensive data types: overall structure

The structure represented informally in figure 3 can be defined in a precise manner using the data type declarations of Miranda (see the appendix for a brief overview of Miranda). The data definition below specifies that the data type `card` has one of two possible forms:

```
> card ::=  No_Card
>         |  Card card_num card_use card_rule card_csa card_activity
```

The form `Card` ... represents correct information. This will be described below in more detail. The form, `No_Card`, represents a totally corrupted card. Providing this `No_`... form for the data type makes it possible to distinguish explicitly between a completely corrupted structure, and a correct structure. All data structures of the smart card prototype are defined in the same *defensive* way, so that at each level, correct as well as corrupt information can be represented. The security aspects of voluntary and involuntary corrupted information on the smart card can thus be studied in detail.

The form `Card ...` states that a proper `card` data structure has five components: an identification number of type `card_num`, a usage count of type `card_use`, a list of rules of type `card_rule`, a list of CSA descriptions of type `card_csa` and finally a description of the status of the card of type `card_activity`.

In the prototype, all data structures carry an identification number. For efficiency reasons, the production implementation of the operating system omits identifications for some data structure.

The definitions below reveal some of the structure of the five components of the constructor `Card`:

```
> card_num      ::= No_Card_Num
>                 | Card_Num string
> card_use      ::= No_Card_Age
>                 | Card_Use int
> card_rule     ::= No_Card_Rule
>                 | Card_Rule [rule]
> card_csa      ::= No_Card_CSA
>                 | Card_CSA [csa]
> card_activity ::= No_Card_Activity
>                 | Card_Off
>                 | Card_On
>                 | Card_Active csa_num cta_num
```

Some of the basic types will not be further elaborated. These include the data type `string`, which is a string of characters and `int`, which is an integral number.

The data type `card_activity` describes the status of the card. The status is `Card_Off`, when the card is not inserted into a reader. The status becomes `Card_On` as soon as the card has been inserted into the reader. This causes an application to be activated. The reference to the opened application is encoded in the form of a data structure `Card_Active`, which holds a pair of numbers, describing the CSA (a value of type `csa_num`) and the CTA (a value of type `cta_num`).

Defensive data types: an example of use

Having defined a number of defensive data structures let us now turn to the problem of accessing such data structures in a disciplined manner. Each data structure may possibly be corrupt, as represented by the `No_...` constructor. Therefore, when accessing such a data structure each component of interest must be analysed carefully. The pattern matching facilities of Miranda are ideal for this purpose, as pattern matching allows for a case analysis of a data structure.

Consider as an example the task of accessing the identification number of a smart card. The function `card_to_num` as shown below explicitly distinguishes between a valid card with a valid identification number on the one hand, and on the other hand a card that does not have a valid identification number, or a card that is completely corrupted. An appropriate error message is issued if anything is wrong.

```
> card_to_num :: card -> string
> card_to_num (No_Card                          ) = abort "corrupted card"
> card_to_num (Card (No_Card_Num) cu cc cs ca) = abort "corrupted card number"
> card_to_num (Card (Card_Num cn) cu cc cs ca) = cn
```

Since all data structures of the smart card prototype are built in the same defensive way, the functions operating on the data structures must take the "defensive state" of each structure into account explicitly. This guarantees that at all times, the health of the data structures is monitored and appropriate action can be taken as soon as a problem occurs.

The Miranda compiler performs strong, static type checking to ensure that the argument to `card_to_num` is of type `card`. The compiler cannot enforce by static type checking that the actual argument always has the form `Card ...`, since this form and the form `No_Card` are both of the same (static) type `card`. As the distinction between these two forms is made at runtime, defensive data types can be viewed as a form of dynamic type checking. This is something that one would not expect to encounter in a language with strong type checking. Yet we have shown that dynamic type checking is useful. Furthermore, with defensive data types dynamic and static type checking are completely orthogonal.

Defensive data types: application structures

A multi-application smart card must be capable of storing the information pertinent to each application in such a way that applications cannot interfere with one another. To achieve this, each card system application is represented by a data structure of type `csa`. The constructor `CSA` shown below lists five components. All `No_...` variants in subsequent definitions of data structures will be omitted, to make the presentation succinct.

```
> csa ::= CSA csa_num csa_part csa_rule (sec csa_rule) csa_cta
```

The first component of a CSA, which is of type `csa_num`, is a number used to identify the application to the smart card terminal. The second component of type `csa_part` defines the operational data of the application. The component `csa_rule` specifies which rules are available to the CSA, in addition to the rules available to all CSAs, as specified in the `card_rule` data type. The fourth component is another set of rules, which implement security functions: `(sec csa_rule)`. The last component of type `csa_cta` lists the CTAs available with the current CSA. The data types `part`, and also `rule` and `cta`, which describe the components of the respective lists, will be defined later.

```
> csa_num  ::= CSA_Num string
> csa_part ::= CSA_Part [part]
> csa_rule ::= CSA_Rule [rule]
> csa_cta  ::= CSA_CTA [cta]
```

A CSA offers a different CTA for each different kind of terminal that can be used with the CSA. The main purpose of the `cta` data structure is to describe the access rights associated with the terminal. The CTA must therefore describe both the data parts that are accessible, and the rules that can be applied to the data. Access to data parts may be restricted to certain modes (i.e. read and/or write). The rules may be associated with specific security parameters and security rules.

```
> cta ::= CTA cta_num cta_import cta_part_perm cta_rule_perm
```

The `CTA` constructor has four fields. The first is an identification number, which is of type `cta_num`. The second field of type `cta_import` specifies data that is imported from other CSAs on the card. This feature is typically used to store the name of the card owner in one CSA only, and make it available to other CSAs without having to duplicate the data. Only data with unrestricted access can be imported, to maintain strict security requirements. The last two fields of the `CTA` constructor specify access rights to parts (`cta_part_perm`) and rules (`cta_rule_perm`). These data types are not further elaborated here (See [9]).

The presentation of the three main data structures `card`, `csa` and `cta` is now complete. Before we can begin to define the secure instructions and the secure instruction interpreter, we must look at the two operational data structures `part` and `rule` that remain to be defined.

Defensive data types: operational data

The main data types `card`, `csa` and `cta` are layers in the design hierarchy, which provide security and an operational framework. The storage of data and operations is are provided by two separate structures: `part` and `rule`.

The `part` data type defines an individually addressable part of the total available storage on the card. Associated with each "part" is an identification number of type `part_num`, a set of permissions of type `part_perm`, an actual value of type `part_value` and a backup value of type `part_backup`. The defensive data type mechanism gives us a direct way to represent corrupt data storage.

```
> part ::= Part part_num part_perm part_value part_backup
```

The four components of the `Part` constructor are defined below. A part identification number may be either one of the five special values `Part_Arg` till `Part_Unique` or the name may be given by an arbitrary string. A part may have any combination of two sets of permissions, which are defined as `readable=Rd_Ok | Rd_Nok` and `writable=Wr_Ok | Wr_Nok`. This simple permission system can be extended to include more complex access conditions. A permission for a secret key, which may be replaced, and used by encryption routines, but which may never be read is an example.

The data type `data` allows various kinds of basic data types (e.g. `int` or `string`) to be stored in a part. The data type `part_backup` has the same structure as the `part_value`.

```
> part_num    ::= Part_Arg
>                 |  Part_Sec_Arg
>                 |  Part_Result
>                 |  Part_Sec_Result
>                 |  Part_Unique
>                 |  Part_Num string
> part_perm   ::= Part_Perm readable writable
> part_value  ::= Part_Value [data]
> part_backup ::= Part_Backup [data]
```

The data type `rule` binds a function of type `rule_action` to an identification number, of type `rule_num`.

```
> rule        ::= Rule rule_num rule_action
> rule_num    ::= Rule_Num string
> rule_action == ip_oper -> [part] -> part -> part
```

As shown in figure 3, and also in the definition of the `csa` data type, access to a `part` is provided only from within a particular CSA. This means that unauthorised access from one CSA to the parts of another CSA is prohibited.

4.2 Input and output data

Having completed the description of the five main data types that represent the permanent, internal data of the smart card operating system, we now turn to the data structures representing the input and output packets as they are exchanged by the smart card system and the terminal. These transient data structures are shown as packets in Figure 3.

An instruction from the secure instruction set is implemented as the data type `ip` as shown below. Instructions are issued exclusively by the smart card reader, in the form of input packets:

```
> ip ::= IP ip_num ip_command ip_oper ip_data (sec ip_data)
```

Each input packet bears an identification number of type `ip_num` as its first component. The second component specifies what kind of instruction the present packet encodes. The remaining fields specify operands of the present instruction. The `ip_oper` field gives the names (addresses) of any operands that are stored on the card. Operational data generated by the smart card reader is supplied as the fourth component. Data used for secure operations is supplied as the fifth component.

```
> ip_num      ::= IP_Num int
> ip_command  ::= IP_Reset
>                 |  IP_Open_App csa_num cta_num
>                 |  IP_Command rule_num
>                 |  IP_Close_App
> ip_oper     ::= IP_Oper [part_num]
> ip_data     ::= IP_Data [data]
```

The secure instruction set has three special instructions to manipulate the general state of the card and one regular instruction. The first special instruction, `IP_Reset`, is issued when a smart card is inserted in the card reader. In the smart card prototype the `IP_Reset` instruction assumes the role of the hardware reset. The two other special instructions serve to open and close an application. These instructions are called `IP_Open` and `IP_Close` respectively. The `IP_Open` command is parameterised with the names of the CSA and the CTA to be opened. Once opened, a smart card application will accept any of a number of regular instructions, until the application is closed. The regular instruction (called `IP_Command`) is parameterised with respect to the required functionality. During a complete card/terminal dialogue the following commands are issues: reset, open application, zero or more regular commands and a close application.

The result of processing an instruction, whether special or regular, is gathered in an output packet of type `op`. An output packet bears an identification number and the result proper of performing the command:

```
> op ::= OP op_num op_answer
```

All instructions except reset generate an answer that carries operational data of type op_data, secure data (sec op_data) and a result state. The result state is defined as `state = Ok | Nok string`, so that a meaningful message can be attached to problem situations. The answer generated for the reset instruction consists of a list of the applications available on the card. In addition a unique transaction number is delivered by the card. This number can be used in authentication protocols and other security processing.

```
> op_num         ::= OP_Num int
> op_answer      ::= OP_Answer op_data (sec op_data) state
>                  | OP_Answer_To_Reset op_csa_num op_transaction
> op_data        ::= OP_Data [data]
> op_csa_num     ::= OP_CSA_Num [csa_num]
> op_transaction ::= OP_Transaction int
```

This concludes the presentation of the defensive data structures for permanent and transient data as they are used by the prototype of the smart card operating system.

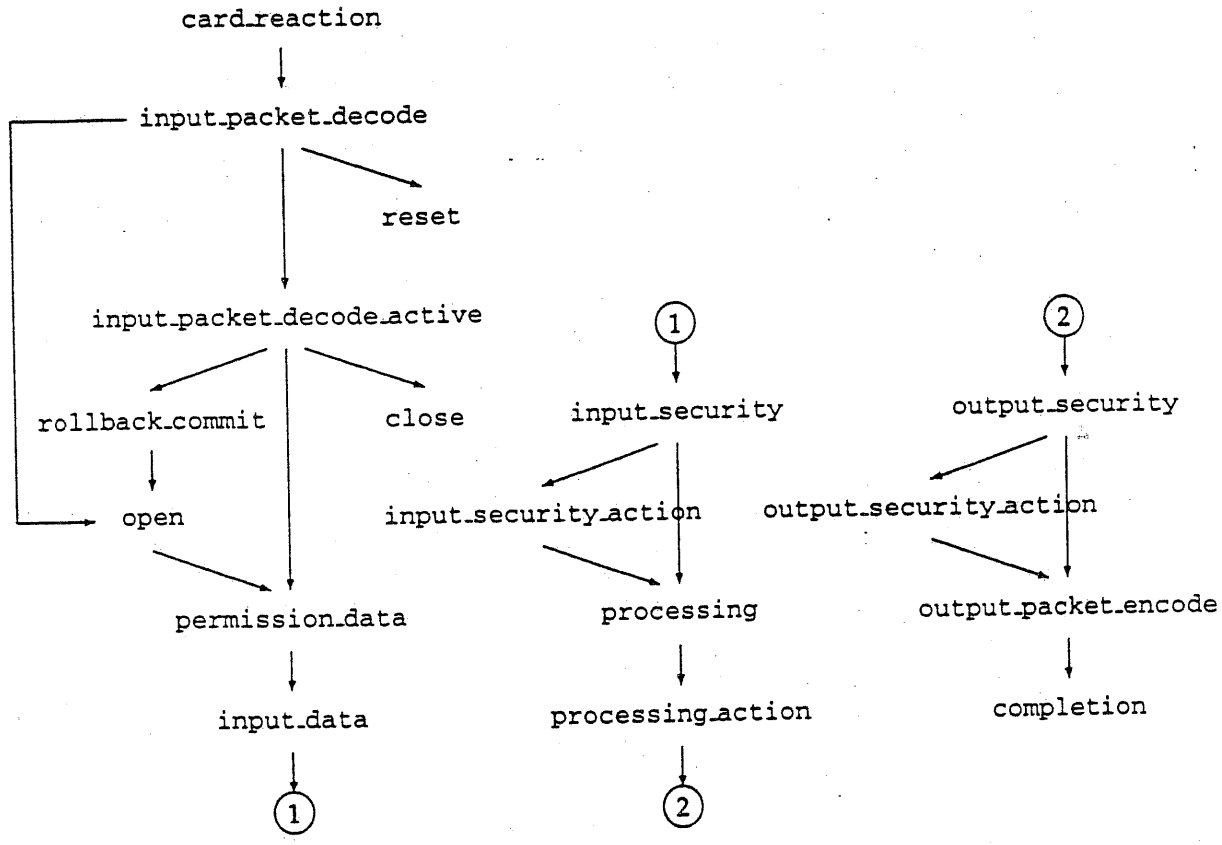

Figure 4: Finite state machine representing the secure instruction interpreter. The arrows represent state transitions. The states are shown as the nodes in the graph. The initial node is card_reaction and the completion, reset and close nodes are the final nodes. All error transitions have been omitted.

4.3 The secure instruction interpreter

The multi-application smart card and the card reader communicate through the exchange of packets. The packets received by the card are interpreted by the secure instruction interpreter. This interpreter is

implemented as a function that takes an input packet and a card as arguments, and produces an output packet. The output packet is the reaction on the input packet. A complete dialogue between card and terminal consists of one or more pairs of input/output packets. The terminal has the initiative. All packets are processed synchronously, which means that while the card is busy producing a response to an input packet, the terminal waits and vice versa.

The first step: reaction of the card to an input packet

The secure instruction interpreter is implemented as a finite state machine. The state diagram is shown in figure 4. The initial node is called `card_reaction`, because it represents the first step in the reaction of the card to an input packet. There are several final nodes, of which `completion` represents an orderly successful completion of ordinary instruction processing. The node `reset` represents successful completion of the `IP_Reset` command and `close` represents normal completion of the `IP_Close` command.

The state diagram of Figure 4 reflects the implementation of the secure instruction interpreter. A node represents a function and an edge represents a function application. The diagram does not show the actual arguments to the functions, because some functions take so many arguments that the diagram would become cluttered.

The function `card_reaction` as shown below corresponds to the initial node of the state transition diagram. The type declaration of the function shows that its first argument is of type `card` and that its second argument is an input packet of type `ip`. The output of the function is a tuple with an output packet of type `op` and a card as a result. It would not be sufficient for the instruction interpreter to deliver only an output packet, as the instructions must be able to change information held by the card. The use of a functional programming style forces this aspect to be made explicit. This makes it possible to reason about questions such as: "When does the data change?" and "Who makes the changes?".

```
> card_reaction :: card -> ip -> (card,op)
> card_reaction (Card cn (Card_Use cu) cr cs ca) i
>                   = input_packet_decode c' ca i
>                     where
>                     c' = (Card cn (Card_Use (cu+1)) cr cs ca)
> card_reaction c i = (c,answer_error c "card_reaction c")
```

The initial step in reacting to an input packet is to increment the use count on the card. Therefore, the `card_reaction` function specifies that its first argument should be a valid smart card, whose second component must specify a valid use count. There are no specific requirements placed on the input packet (`i`), except that there should be one. If all is well, the function `input_packet_decode` is called, with three arguments: the first is a new card, which is identical to the old card except that the use count has been incremented. The second argument of `input_packet_decode` is the card activity component taken from the card, and the last arguments is the input packet. The chain of function calls starting with the call to `input_packet_decode` should eventually lead to the production of a tuple consisting of the new card and an output packet. This output packet is produced by `completion`.

If the first argument to `card_reaction` does not exhibit the structure as specified in the first clause by the pattern of the formal argument, the use of defensive data structures causes the second clause of the definition to take force (Note that in Miranda the defining clauses of a function are tried in order). Again, the function result is a tuple consisting of the card and an output packet. This tuple is formed by the auxiliary function `error`. In the state transition diagram of Figure 4 no error transitions are shown, because there are several error transitions from every single node.

The second step: decoding an input packet

The normal sequence of events causes `card_reaction` to transfer control to `input_packet_decode`. The latter initiates the execution of the command encoded in the input packet. The state transition diagram of Figure 4 shows that the node labelled `input_packet_decode` has three outgoing edges. These edges appear in the functional program as three defining clauses of the function. The first defining clause, which corresponds to the edge that connects to the node `reset` is shown below. The other defining clauses will be described shortly.

```
> input_packet_decode :: card -> card_activity -> ip -> (card,op)
> input_packet_decode c Card_Off (IP in (IP_Reset) io id sec_id)
>                 = reset c
```

The result of obeying a reset instruction when the card is in state "off" is to switch the state to "on". This is taken care of by the function `reset`. The output packet that is manufactured by `reset` will be returned to the card terminal.

When compared with `card_reaction`, the function `input_packet_decode` takes an extra parameter of type `card_activity`. In all other respects its parameters are identical with those of `card_reaction`. Providing the `card_activity` information separately, in addition to its being part of the `card` argument, serves two purposes. Firstly, it establishes the fact that the data structure holding the information has been found in a fit state, so there is no need to check that data again. Secondly, providing information separately focuses on data that is of particular importance in the current function.

The patterns specified in the formal arguments of the first clause of the definition of `input_packet_decode` match if the card is currently in the state "off", and if the command is a reset command. This combination is one of a number of possible combinations of the state in which a card may find itself and the commands that it receives. As discussed in Section 4.1, a smart card may be in any of three states (`Card_Off`, `Card_On` or `Card_Active` ...). In each of the three states, the smart card may encounter one of the three special instructions (`IP_Reset`, `IP_Open`, `IP_Close`) or the general instruction `IP_Command`. The function `input_packet_decode` handles all possible combinations, but only a few are really interesting.

The second combination of card state and command that we will discuss corresponds to the situation where the card has just been reset (it is in the state "on") and an "open application" command is issued. In this case control is passed to the function open, which is responsible for properly opening the application as identified by the CSA number `sn` and the CTA number `tn`. We will not discuss this further, but note that opening an application sets the state of the card to `Card_Active`. The terminal receives an output packet with the results of the open application command.

```
> input_packet_decode c Card_On (IP in (IP_Open_App sn tn) io id sec_id)
>                 = open c sn tn io id sec_id
```

We will not discuss the case where an "active" card receives an ordinary command, which in the state diagram brings us to the node labelled `input_packet_decode_active`. Instead we now look at some irregular situations. Consider the case below, in which the card is already in the state "on", when a reset command is issued:

```
> input_packet_decode c Card_On (IP in (IP_Reset) io id sec_id)
>                 = (c,answer_reset c)
```

This is an error, because the card should have been in state "off", as discussed earlier. This problem occurs when a smart card user removes the card from the reader before a dialogue is complete. The smart card operating system cannot properly wind up the current dialogue. The operating system has to take all error situations into account by providing a mechanism, which effectively jettisons all changes made by the incomplete dialogue. In the present case there is no cleanup to be done, because the reset command only changes the card state and no other data. The action taken therefore is merely to produce the answer to reset output packet.

The last defining clause of `input_packet_decode` lumps all remaining combinations of card state and command type into one error packet:

```
> input_packet_decode c ca i
>                 = (c,answer_error c "input_packet_decode i")
```

The description of these cases shows that the pattern matching facility of a functional language provides support for case analysis. It is not difficult to enumerate all cases, study the implications of each and every one of them, and thereby arrive at a program that is prepared for the worst, without in fact cluttering the code that handles the regular cases.

Further steps: general remarks about the transition diagram

It would be useful to present the entire implementation of the state diagram here. This would show that a complete implementation of a smart card operating system prototype can be made in a functional language, such that the intended behaviour of the smart card is fully described. However, such a presentation would take up too much space, so we will have to content ourselves with the more abstract view as provided by the state diagram of Figure 4. In essence each state transition causes some changes to be made to the internal state value of the card. This value is explicitly carried from one state to the next. At the terminal nodes of the diagram, the state and the output packet are made available. What remains to be investigated is how the state transition machine can be embedded in a framework for conducting a dialogue between the smart card and the terminal. This corresponds to the top layer in Figure 3, and this subject is discussed in the following section.

4.4 A dialogue between smart card and terminal

The state diagram of Figure 4 represents the entire course of action that ensues after receipt of an input packet by the smart card, up to and including the production of an output packet. This is sufficient to describe a single smart card transaction, but to describe a dialogue (a sequence of terminal actions and card reactions), a model implementation of a smart card terminal is also required. The function `dialogue` below represents such an implementation. The type `some` represents the state value, as maintained by the terminal. This value is part of the user interaction with the card terminal. It contains information pertinent to the application as needed by the card terminal to drive the dialogue trough a sequence of transactions with the smart card. The function `dialogue` is presented with an initial smart card of type `card`, an initial state of the terminal of type `some` and an initial (dummy) output packet of type `op`. The dialogue then consists of a synchronised sequence of calls to `some_action` and `card_reaction`. The dialogue ends as soon as the auxiliary function `ok_ip` returns false. This shows that the smart card operates as a slave/server to the terminal. (In the definition of the function `dialogue` below, the definitions of `s'`, `i'`, `c'` and `o'` as introduced under the Miranda keyword `where` are in scope throughout the entire function body):

```
> dialogue :: card -> some -> op -> card
> dialogue c s o  = c, if ~ ok_ip i'
>                 = dialogue c' s' o', otherwise
>                   where
>                   (s',i') = some_action s o
>                   (c',o') = card_reaction c i'
```

The function `dialogue` maintains the state of the card and the state of the terminal simultaneously, and also arranges for the exchange of the input and output packets. When at some point the terminal decides that the dialogue has finished, the final state of the card is returned.

The security of the combined smart card and terminal implementations crucially depends on the correctness of the `dialogue` function, because it is the only function that has access to both the card state and the terminal state. However, it is easy to see that the functions called by `dialogue` do not have access to each others state. The `some_action` function only sees its own state `s`. Similarly the `card_reaction` function only sees its own state `c`. An implication is that the secrecy of information used by cryptographic protocols in either the card or the terminal is not compromised.

4.5 The liveness of the prototype

The material introduced in the preceding sections is sufficient to discuss a property of the prototype implementation in an informal way. The reasoning could be formalised if the complete implementation were introduced. The present discussion serves to demonstrate the appropriateness of the prototype for formalising reasoning about its properties. Such formalisations contribute to achieving a sufficient level of confidence about the reliability of the prototype.

From the state diagram of Figure 4 it can be seen that there are no back edges, but only forward edges. If the amount of work performed by each node is strictly limited, the state diagram will always be traversed in an amount of time that is bounded by some constant. The prototype implementation is thus guaranteed to respond within a certain amount of time to any input packet.

To proof the amount of work performed by each node as limited requires a detailed analysis of the complete code, so this is beyond the scope of the paper. However, the line of reasoning is not difficult to give:

1. The only recursive data structure used in the implementation is the list. This can be verified by inspecting all the data type definitions in the previous sections.
2. All lists stored on the card are finite because the card has a finite (rather small) storage capacity.
3. The implementation creates no circular lists. Because there is no assignment in functional languages, an existing list cannot be changed to become circular.
4. The number of functions called by each node of the state machine is bounded by the sum of the length of all lists that are stored on the card, assuming a worst case scenario in which all lists are visited (searched and/or copied) once. In reality, only a few nodes cause some lists to be visited, so the upper bound is rather coarse.

Each node thus performs a finite amount of list manipulation, thereafter control is passed to the appropriate successor node. The conclusion is that the implementation will produce a reaction to any input packet within a finite amount of time. This establishes the liveness of the prototype.

In the informal proof above, it has been assumed that all the data structures involved are constructed according to the specifications. In particular no circular lists are constructed. This can be guaranteed only if the compiler and the hardware function correctly. So in fact, correctness proofs as sketched above should include proofs relating to the compiler and the hardware, which are difficult to give. To cope with malfunctioning compilers and hardware, in the smart card prototype we use the alternative `No_...` forms of the data structures. If malfunctioning smart card hardware causes a pointer to be destroyed, so that a list becomes circular, then we wish to represent this by interpreting the corrupted list as `No_...`.

The explicit representation of corrupt data structures provides a way of identifying tests that might be incorporated in the production implementation of a smart card operating system. A general way to do that would be to modify the compiler so that it would associate a check sum with each pointer, which could be verified when the pointer is dereferenced. A more specific test would be to associate a counter with each routine responsible of traversing a list. As soon as the counter exceeds a predetermined number of steps, the list is declared circular. Because of space requirements, smart card systems are partly written in assembler. Therefore programming errors giving rise to corrupt data structures are a fact of life, that must be dealt with. Defensive data types help to identify testing strategies that deal with such eventualities.

The liveness of the secure instruction interpreter has been established, without precise information about exactly how long it will take for each input packet to be processed. This depends on the implementation of the nodes in the state machine. Even if these implementations are taken into account, then we still cannot be accurate. The reason is that only one layer of the smart card operating system has been prototyped. In this layer the only notion of time available is the equivalent of one function call say, or one visit to an element of a list. To be more accurate, one would have to implement the compact code interpreter, and the machine code interpreter as well. Then statements involving timing can be made in terms of a number of machine instructions.

5 Conclusions and future work

We have built a prototype implementation of a smart card operating system. The system provides full functionality, including support for security based on RSA and on DES. The prototype has the form of a Miranda program, which can be executed on a work station. The prototype can be viewed as an executable specification of the system, which gives confidence in the design of the system.

The prototype has been built at the same time as a production implementation of the system. Both implementations were built using the same set of design documents. Building the two implementations simultaneously provided opportunities for the respective implementors to discuss their implementations as they were built. Because of the different paradigms (imperative versus functional) and approaches to building the implementations (top-down versus a mixture of top-down and bottom-up), the implementors were thinking about their programs in rather different ways. Mistakes the implementors made were also rather different. Because of the different views the implementors held for their programs, the

mistakes made by one implementor would be identified as such by the other without difficulty. This has helped to produce a good implementation of the smart card operating system.

The implementation of a prototype of a smart card operating system in a functional language has made it possible investigate formal claims about the implementation, such as its liveness. The present implementation takes into account only a one layer of the smart card operating system, so that the properties of the implementation that can be reasoned about are approximate.

Using a functional language to build the prototype allows us to make the following two observations concerning the behaviour of the prototype. The first is that state manipulations are made explicitly. Every piece of information that is passed on from one function to the next can be identified. Information that is not passed on is simply not accessible. Secondly, no information can be changed; it is only possible to build new data structures from old structures whilst leaving the old structures untouched. These two observations are entirely consistent with the design documents. Therefore they also hold for the design itself, which gives us confidence in the design.

The use of defensive data structures proved useful, both to be able to reason about problem situations and to identify tests for coping with such situations. As part of future work we plan to refine the defensive data type system. As well as modelling correct and corrupt information one could model other information attributes, such as secret and public, permanent and temporary etc. Offering more data attributes would increase the level of confidence that we may have in the system.

Much work remains to be done to take further advantage of the parallel development process of prototype and production system. In a first step the structure of the prototype can be brought into conformity with the more refined hierarchical organisation of the production system. Such detailed conformance is expected to allow more detailed reasoning with a high level of confidence about the production implementation. In a second step the detailed prototype implementation could be embedded in the form of comments and/or assertions in the production system. By expedient use of software tools, prototyping information can be transferred to the testing phase of the production system, thus further improving testability of smart card operating systems.

Acknowledgements

We thank Marcel Beemster and Hugh McEvoy for their comments on a draft version of the paper. Andy Gravell has made a number of useful suggestions for which we are grateful.

References

[1] M. Archer, J. Bock, D. Frincke, and K. Levitt. Effectiveness of operating system prototyping from a template: application to MINIX. In N. Kanapoulos, editor, *2nd Rapid system prototyping*, pages 55–66, Research Triangle Park, North Carolina, Jun 1991. IEEE Computer Society Press, Los Alamitos, California.

[2] M. Archer, D. Frincke, and K. Levitt. A template for rapid prototyping of operating systems. In S. Winkler, editor, *1st Rapid system prototyping*, pages 119–127, Research Triangle Park, North Carolina, Jun 1990. IEEE Computer Society Press, Los Alamitos, California.

[3] M. Ben-Ari. *Principles of concurrent programming*. Prentice Hall, New York, 1982.

[4] R. S. Bird and P. L. Wadler. *Introduction to functional programming*. Prentice Hall, New York, 1988.

[5] D. Chaum. Privacy protected payments: unconditional payer and/or payee untraceability. In D. Chaum and I. Schaumüller-Bichl, editors, *Smartcard 2000: the future of IC cards*, pages 69–93, Laxenburg, Austria, Oct 1987. North Holland.

[6] E. K. de Jong Frz. *Smart card instruction set, European Patent Application*. QC consultancy, Zaandam, 1994.

[7] E. K. de Jong Frz and J. N. E. Bos. An application tool kit for smart cards: Security "as you like it". In *Smart Card 1994, Day one: Market overview of Leisure, finance and security*, pages 76–81, London, England, Mar 1994. Lowndes Exhibition Organisers, Peterborough, England.

[8] S. Even. Secure off-line electronic fund transfer between nontrusting parties. In D. Chaum and I. Schaumüller-Bichl, editors, *Smartcard 2000: the future of IC cards*, pages 57–66, Laxenburg, Austria, Oct 1987. North Holland.

[9] P. H. Hartel and E. K. de Jong. Prototyping a smart card architecture in a lazy functional language. Technical report CS-94-08, Dept. of Comp. Sys, Univ. of Amsterdam, May 1994.

[10] R. J. M. Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, Apr 1989.

[11] ISO. *Draft international standard 7816-4.2 "Inter industry command set"*. International Standards Organization, 1993.

[12] P. Paradinas and J-J. Vandewalle. How to integrate smart cards in standard software without writing specific code. In *Cardtech/Securtech*, pages 69–85, Arlington, Virginia, Apr 1994. Cardtech/Securtech Inc. Rockville, Maryland.

[13] M. Reston. Testing of the rapidly developed prototypes. In S. Winkler, editor, *1st Rapid system prototyping*, pages 139–143, Research Triangle Park, North Carolina, Jun 1990. IEEE Computer Society Press, Los Alamitos, California.

[14] B. Struif. Das smartcard anwendungspakket STARCOS. *Der GMD Spiegel*, 22(1):29–34, Mar 1992.

[15] A. S. Tanenbaum. *Structured computer organisation*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1984.

[16] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 1–16, Nancy, France, Sep 1985. Springer-Verlag, Berlin.

[17] M. Ugon. Smart card - present and future. In D. Chaum and I. Schaumüller-Bichl, editors, *Smartcard 2000: the future of IC cards*, pages 3–18, Laxenburg, Austria, Oct 1987. North Holland.

[18] M. Ugon. Future of smartcard operating systems. In *Cardtech/Securtech*, pages 245–255, Arlington, Virginia, Apr 1994. Cardtech/Securtech Inc. Rockville, Maryland.

Appendix – A brief introduction to functional programming in Miranda

In a functional programming language such as Miranda, functions are first class citizens. This means that a function may be passed as an argument to another function, a function may be delivered as the result of a computation and a function may be stored as part of a data structure.

A Miranda program consists of a number of data type and function definitions, and an expression to be evaluated. The formal arguments of a function definition may specify patterns involving arbitrary data structures. Consider the program below as an example. Each line of program text is preceded by a special symbol (>) using the literate programming convention:

```
> num_tree ::= Num_Branch num_tree num_tree | Num_Leaf num

> walk_add :: num_tree -> num
> walk_add (Num_Leaf data)         = data
> walk_add (Num_Branch left right) = (walk_add left) + (walk_add right)

> walk_main :: num
> walk_main = walk_add (Num_Branch (Num_Leaf 1)
>                      (Num_Branch (Num_Leaf 2) (Num_Leaf 3)))
```

The first line defines a binary tree of numbers as the data type `num_tree`. The identifier `Num_Branch` is a data constructor, which in the functional programming paradigm can be viewed as a constant function of two arguments: a left sub tree and a right sub tree. Both subtrees are values of type `num_tree`. The identifier `Num_Leaf` is again a data constructor, this time taking one argument; a numeric value.

The next three lines of the example program define a function `walk_add`, which traverses the tree, adding the numbers stored in the leafs. The last three lines define a (parameterless) function `walk_main`, which applies `walk_add` to a small binary tree. In Miranda, parentheses serve to build expressions out of groups of symbols, and not to delineate function arguments. The first line of each function definition specifies the type of that function. A type specification is preceded by the symbol `::`. The type of `walk_add` specifies that the function has one argument with values of type `num_tree`, and that the function result is of the type `num`. The "function" `walk_main` has no arguments, so that the type specification of `walk_main` only mentions its result type: `num`.

The definition of `walk_add` has two clauses: the first clause applies when a leaf node is encountered. In this case the data stored in the leaf node is returned. Interior nodes of the tree do not contain numeric data, so the function result returned by the second clause is the sum of the results from the left and right branch.

The `walk_add` function is a combination of two things: it embodies a tree traversal algorithm, and it encodes a particular operation (addition) over the tree. These two issues can be separated, by introducing a polymorphic data type `tree`, and a pure tree walk function `walk`, as shown below. The `tree` data type has a type parameter, indicated by the asterisk, for which any concrete type may be substituted. Similarly, the `walk` function carries a parameter, which is the binary operator to be applied when results of the left and right branch are to be combined. The function `walk_main'` (in Miranda the apostrophe can be used as part of an identifier) applies the general `walk` function to a concrete binary tree which has numbers at its leaf nodes. Here `(+)` is the notation for the addition function on numbers, where we have used the convention that any infix binary operator can be turned into a prefix function by enclosing the operator in parentheses.

```
> tree * ::= Branch (tree *) (tree *) | Leaf *

> walk :: (*->*->*) -> tree * -> *
> walk op (Leaf data)          = data
> walk op (Branch left right) = op (walk op left) (walk op right)

> walk_main' :: num
> walk_main' = walk (+) (Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)))
```

The following identity relates the two functions that have been defined thus far: `walk (+) = walk_add`.

This concludes the brief introduction into functional programming using Miranda. The interested reader is referred to Bird and Wadler [4] for a thorough treatment.