ELSEVIER

The Journal of Systems and Software 41 (1998) 161–173

The Journal of Systems and Software

A practical language and toolkit for high-integrity tools

I. Toyn [*], D.M. Cattrall, J.A. McDermid, J.L. Jacob

Department of Computer Science, University of York, Heslington, York YO1 5DD, UK

Received 26 February 1996; received in revised form 7 June 1996; accepted 10 October 1996

Abstract

This paper shows how a safe interface to heap storage, based on garbage collection as provided in implementations of pure functional languages, can be combined with imperative languages. It also shows how expressive notation from functional languages, such as algebraic data types and equational definition of functions with pattern matching, can be adopted. The paper argues that the resulting combination is appropriate for the construction of high-integrity tools, based on an assessment against the same criteria as have been used for assessing the suitability of imperative languages for producing high-integrity software. © 1998 Elsevier Science Inc. All rights reserved.

Keywords: Functional languages; Imperative languages; Heap storage; Algebraic data types

1. Introduction

This paper is concerned with notation and tools that assist in the development of correct software. It is couched in the context of high-integrity software where correctness is the primary concern, but the ideas are more widely applicable than that. After clarifying what is meant by the term high-integrity software, the paper reviews the requirements on languages for high-integrity software. These requirements form the basis for subsequent assessments of the suitability for high-integrity software of imperative languages, functional languages and finally a combination of notation from functional languages with an imperative language. A toolkit supporting a combination of functional notation with the imperative language C is presented, and some uses of the toolkit are discussed.

2. What is high-integrity software?

Computers are increasingly being used in systems whose failure could lead to financial loss, damage to the environment, personal injury, or even loss of life. The software component of these systems must exhibit a low rate of failure and good failure behaviour: any failures must have only minor consequences (Anderson, 1989). Software that is suitable for such use is known as high-integrity software.

For there to be confidence in the dependability of high-integrity software, the intended behaviour of the software must be carefully specified and the compliance of programs with these specifications must be argued (Sennett, 1989). Ideally, the specifications should be written in formal (mathematical) language so that they have a precise meaning, and the programs should be shown to comply with the specifications by a process based on formal reasoning. However, formal development is relatively uneconomic, and the mechanical tools needed to assist it are not yet mature enough for widespread application. In current practice, confidence in most high-integrity software is gained by a mixture of systematic construction and testing.

It is important to realise that it is the binary compiled code of high-integrity software that must be dependable. This code results from a development process involving tools for specification analysis, refinement, proof, program analysis, compilation, assembly and linking. A fault in a compiler can directly affect the integrity of the software, e.g. by introducing a flaw, whereas a fault in an analysis tool can result in an incorrect assessment of the software and hence affect integrity only indirectly, e.g. by failing to detect a flaw in the code. There are characteristic differences between these *high-integrity tools* and *high-integrity application software*: tools have no hard real-time constraints; and only application software might have severe limitations on memory.

[*] Corresponding author. Tel.: +44 1904 433386; e-mail: ian@cs.york.ac.uk.

High-integrity tools are usually too large and complex to be realisable by fully formal development – although this situation may change as tools evolve.

The uncertainty about the degree of dependability that is both necessary and economically achievable is reflected by the different views adopted by standards. IDS OO-55/56 (MoD, 1991a, b, 1993) require certain tools, e.g. the compiler, to be produced to the same level of rigour as application software (fully formal development). In contrast the civil standard DO-178B (RTCA, 1992) has much less stringent requirements, based on the approach used by Airbus Industries on the Airbus family, in its SAO (Specification Aidé per Ordinateur) tools.

3. What are the requirements on languages for high-integrity software?

Some high-level requirements on languages for high-integrity software have been given by Carré (1989). The following list uses Carré's names for the requirements, with our own abbreviated explanations.

Logical soundness: The language must be logically coherent and unambiguous, with formally defined semantics, otherwise formal reasoning is impossible.

Complexity of formal language definition: The formally defined semantics must not be too complex, otherwise formal reasoning is impractical.

Expressive power: The programming language should aim to approach the conciseness of expression of a specification language, otherwise formal reasoning, such as involved in showing the compliance of a program with its specification, is more difficult. [1]

Security: High-integrity software must not fail at run-time: the language definition should ideally permit static detection of all misuses of the language. Potential problems include uninitialized variables and array indices out of range. Static detection of such problems is discussed in Garnsworthy et al. (1993), which illustrates what can be achieved by a combination of language restrictions and proof obligations discharged by a theorem prover. Although that work addresses all of the exceptions that can be raised during the execution of a SPARK program, it relies on informal annotations as well as the formal program text. These annotations typically reflect properties of the possible inputs to the program, and thus are assumptions on which the static analysis depends.

Verifiability: The formally defined semantics must be such that it is possible to show that a program is a correct refinement of its specification.

Bounded space and time requirements: It must be possible to show that any constraints on memory and time usage (as occur in hard real-time high-integrity application software, though rarely in high-integrity tools) are not exceeded.

Some of these requirements conflict: e.g. a language that has greater expressive power or greater verifiability than another language may be less able to satisfy constraints on, and be less amenable to calculation of, space and time usage.

4. Are imperative languages suitable for high-integrity software?

The report of Cullyer et al. (1991) assesses the suitability of various existing imperative programming languages to high-integrity software development. It presents assessments of a typical structured assembler, C, CORAL 66, Pascal, Modula-2, and Ada. As these languages were not designed to meet the high expectations represented by Carré's requirements, it is not surprising that they are found to be unsuitable. One approach to this problem is to design a new language to satisfy the requirements, but it is difficult for a new language to gain acceptance. A more pragmatic approach is to find safe subsets: use the existing languages but avoid problematic features, i.e. those features that are not sound, have complex formal definitions, give rise to insecurities, are difficult to verify, or hard to cost. The assessments of an Ada subset and C of Cullyer et al. are now reviewed.

SPARK (Carré et al., 1988) is intended to be a safe subset of Ada. Its semantics has been formally specified (PVL, 1994) in 530 pages of Z (Spivey, 1992). Being a subset, it is less expressive than full Ada. Much static analysis of SPARK programs is possible, with avoidance of run-time failure being guaranteed, subject to the discharge of proof obligations. An approach to the formal refinement of Z specifications to SPARK programs based on Morgan's refinement calculus (Morgan, 1994) has been prototyped (Jordan et al., 1994). SPARK programs do not use heap storage; global and stack storage can be accounted for statically, as can execution times (Chapman et al., 1994).

C (Kernighan and Ritchie, 1988) is a widely used and well understood language. Its implementation on many different computers preceded the definition of its formal semantics. In order to admit these different implementations, its formal semantics admits variations, e.g. the sign of the result of the integer remainder operator is not specified. [2] Such ambiguities make formal verifica-

[1] Note that making the specification language approach the expressiveness of a programming language, i.e. become executable, does not help, as it loses the expressiveness needed to write abstract specifications.

[2] A new library function with defined behaviour is provided.

tion very difficult. There are few restrictions on the use of pointers, making it possible to overwrite arbitrary locations in memory. This insecurity is so deep-seated in the language that there seems to be no worthwhile safe syntactic subset of C. As Carré writes (Carré, 1989, Section 5.3.2, p. 109), "if one removed its dangerous features, little would remain".

The safe subsets that have been defined for various languages invariably exclude heap storage. Heap storage is useful when the quantity of space needed cannot be determined at compile time, or when the data stored in it is required to outlive the subroutine that creates it but without surviving forever as it would in static storage. High-integrity tools usually make great use of heap storage: for instance a compiler would otherwise be forced into making assumptions about the size of program to be compiled, and would have to reserve large quantities of static storage that would rarely be used. The exclusion of heap storage from safe subsets results in a loss of expressive power, so why is it excluded? Consider the problems that can arise with heap storage in C.

In C, heap storage is usually managed by the standard library allocation routine `malloc( )` and reclamation routine `free( )`.

```
extern void *malloc(unsigned size);
extern free (void *ptr);
```

The `malloc( )` routine takes the number of bytes of storage required, and returns a pointer to a suitable area in the heap. The area can subsequently be reclaimed by passing that pointer to `free( )`. This interface to heap memory is open to several kinds of misuse.

- The number of bytes requested of `malloc( )` might be too many or too few.
- The call to `free( )` might be omitted.
- `free( )` might be called prematurely.
- The same pointer might be given to `free( )` more than once.
- The pointer given to `free( )` might not have come from `malloc( )`.

The consequences of these forms of misuse range from omitting to recycle space to possible misinterpretation of the contents of memory. In the worst cases, the immediate consequence is merely the corruption of a few bytes of memory, and it is only when those bytes are used later that the mistake can be detected. Such delayed consequences are a programmer's nightmare. Whether corrupted bytes are ever reused (or whether the program terminates before they are reused) depends on the implementation of `malloc( )` and `free( )`; a program that works with one implementation of these routines may fail with another, as might be used when the program is ported to another machine. A further problem is fragmentation of free memory space: although the total quantity of free space may exceed that requested of `malloc( )`, if it does not occur in a sufficiently large contiguous block then the heap will have to grow or the request must be denied.

Not all interfaces to heap storage share all of these insecurities, but any non-empty subset of these problems justifies the exclusion of heap storage from safe subsets. A quite different interface to heap storage is used within functional programming systems. In Section 5, pure functional languages are described and then assessed against Carré's requirements.

5. Are functional languages suitable for high-integrity software?

We are not aware of any existing assessments of pure functional languages against the requirements listed by Carré, so we present a short review of functional languages followed by our own assessment. The review introduces examples of the notation which will be used in subsequent sections.

5.1. A review of functional languages

A characteristic difference between pure functional languages and imperative languages is in the use of variables. In imperative languages, variables denote storage locations whose values vary with time. In pure functional languages, variables denote values, as in mathematics: there are no assignment statements, and hence no side-effects.

Suppose a type is needed to represent logical propositions, e.g. $A \vee B \wedge \neg C$. Propositions can be literals or other propositions combined by binary disjunction, binary conjunction and unary negation. A functional notation known as an algebraic type definition can be used to define the type of propositions as follows. This definition assumes that `sym` is the type of literals.

```
prop: Lit sym
     | Or prop prop
     | And prop prop
     | Not prop
```

A transformation that one might wish to apply to a proposition is to rearrange it into literal normal form (Bundy, 1983), i.e. a form in which all (if any) negations apply to literals. The rules for this transformation are shown in Fig. 1. Any part of the proposition that matches the left-hand side of one of these rules should be re-

$$\begin{aligned} \neg(\neg A) &\Rightarrow A \\ \neg(A \vee B) &\Rightarrow \neg A \wedge \neg B \\ \neg(A \wedge B) &\Rightarrow \neg A \vee \neg B \end{aligned}$$

Fig. 1. Transformation rules for literal normal form.

placed by something of the form of the right-hand side of that transformation rule. The rules must be applied repeatedly until no more can be applied (termination being guaranteed because the operands of $\neg$ in the right-hand sides are all smaller than those in the left-hand sides). One way of expressing the literal normal form transformation in a functional language is shown in Fig. 2. This defines a recursive function, named `lnf`, by a sequence of equations. When `lnf` is applied to a proposition, the earliest equation in the sequence whose left-hand side pattern matches the whole of that proposition is determined and the appropriate instantiation of its right-hand side is taken to be the value of the application. The first two equations assist in finding propositions to be transformed. The next three equations correspond directly to the transformation rules. The last equation is matched by propositions that are already in literal normal form. This ordering is important: if the last equation were written first, then `lnf` would behave as an identity function.

Functional programs may be type-checked to ensure things such as that applications of `Lit`, `Or` and `lnf` are to expressions of the appropriate types.

As another example, consider the following type definition for binary trees.

```
tree A  : Node (tree A) (tree A)
        | Leaf A
```

In this type, `A` is a parameter standing for an arbitrary type. The data items appearing in the leaves of a tree must all be of the same type, while those of another tree can be of a different type. This is ensured by a polymorphic type-checker (Cardelli, 1987). If there are propositions in the leaves of a tree, then `A` would be `prop` and hence the type of the tree would be `tree prop`.

Suppose a function is needed to compute the sum of the numbers in a tree containing a number in each leaf. It could be written in a functional language like this.

```
treesum (Node t1 t2)
= treesum t1 + treesum t2
treesum (Leaf n) = n
```

In this function, the ordering of the two equations does not matter, as they cover disjoint cases. In the first, both recursive applications of `treesum` must be fully evaluated (to numbers) before the sum can be computed. The same result is computed whichever application is evaluated first. Indeed, the evaluations could be intermingled and the result would not be affected. This independence from evaluation order is a characteristic difference from imperative languages, where evaluation of one operand of + might have side-effects that could affect the value of the other operand.

```
lnf (Or p q) = Or (lnf p) (lnf q)

lnf (And p q) = And (lnf p) (lnf q)

lnf (Not (Not r)) = lnf r

lnf (Not (Or r s)) = And (lnf (Not r)) (lnf (Not s))

lnf (Not (And r s)) = Or (lnf (Not r)) (lnf (Not s))

lnf p = p
```

Fig. 2. Functional program for literal normal form.

Input and output can be treated in a functional language as unbounded streams of data items. In this context, evaluation order matters: a program must avoid consuming all the input it can get without doing any useful computation or producing any output. Lazy evaluation ensures that as much output is produced as possible before any more input is consumed.

The algebraic style of type definition, illustrated by the above examples, subsumes enumerations, record and union types, and is similar to notation used in specification languages, e.g. free types in Z.

The `lnf` function defined in Fig. 2 is not the most efficient functional program for the literal normal form transformation: the matching of `Not` patterns by three separate equations is redundant. Another `lnf` program can be formally synthesized from this one by applying a sequence of equivalence-preserving transformations. The first transformation completes the set of `Not` patterns by replacing the last equation by two separate equations for the patterns that it covers

```
lnf (Or p q) = Or (lnf p) (lnf q)
lnf (And p q) = And (lnf p) (lnf q)
lnf (Not (Not r)) = lnf r
lnf (Not (Or r s)) = And (lnf (Not r))
(lnf (Not s))
lnf(Not (And r s)) = Or (lnf (Not r)) (lnf
(Not s))
lnf (Not (Lit p)) = Not (Lit p)
lnf (Lit p) = Lit p
```

Next, introduce a new function to deal with the four `Not` cases, which assumes that the outermost `Not` has already been matched.

```
notlnf (Not r) = lnf r
notlnf (Or r s) = And (lnf (Not r)) (lnf
(Not s))
notlnf (And r s) = Or (lnf (Not r)) (lnf
(Not s))
notlnf (Lit p) = Not (Lit p)
```

This `notlnf` definition can then be folded in `lnf`'s equations for `Not` patterns.

```
lnf (Or p q) = Or (lnf p) (lnf q)
lnf (And p q) = And (lnf p) (lnf q)
lnf (Not p) = notlnf p
lnf (Lit p) = Lit p
```

Lastly, the third of these equations can be folded in the `notlnf` function.

```
notlnf (Not r) = lnf r
notlnf (Or r s) = And (notlnf r) (notlnf s)
notlnf (And r s) = Or (notlnf r) (notlnf s)
notlnf (Lit p) = Not (Lip p)
```

The last two function definitions are the new literal normal form program.

The definition of literal normal form provided in Fig. 1 is an operational one. An alternative structural definition is given in Fig. 3. This `islnf` definition is in functional notation (assuming appropriate definitions of `True`, `False` and `and`). Given this intuitively correct definition, we can gain more confidence in the correctness of the new literal normal form program by considering the truth of the expression `islnf (lnf p)`, by exhaustive cases of `p`.

```
islnf (lnf(Lit s))
  ⇒ islnf (Lit s)
  ⇒ True

islnf (lnf(Or p q))
  ⇒ islnf (Or (lnf p) (lnf q))
  ⇒ islnf (lnf p) and islnf (lnf q)
  ⇒ True, by induction

islnf (lnf (And p q))
  ⇒ islnf (And (lnf p)(lnf q))
  ⇒ islnf (lnf p) and islnf (lnf q)
  ⇒ True, by induction

islnf (lnf(Not (Lit s)))
  ⇒ islnf (notlnf (Lit s))
  ⇒ islnf (Not (Lit s))
  ⇒ True

islnf (lnf (Not (Or r s))) ⇒ islnf (notlnf
(Or r s))

  ⇒ islnf (And (notlnf r) (notlnf s))
  ⇒ islnf (notlnf r) and islnf (notlnf s)
  ⇒ True, by induction

islnf (lnf (Not (And r s)))
  ⇒ islnf (notlnf (And r s))
  ⇒ islnf (Or (notlnf r) (notlnf s))
  ⇒ islnf (notlnf r) and islnf (notlnf s)
  ⇒ True, by induction

islnf (lnf (Not (Not r)))
  ⇒ islnf (notlnf (Not r))
  ⇒ islnf (lnf r)
  ⇒ True, by induction
```

If there were another auxiliary function `eqv` for testing the equivalence of two propositions, then it would be possible to synthesize `lnf` from the following specification,

```
islnf (lnf p) and eqv p (lnf p)
```

```
islnf (Lit s) = True

islnf (Not (Lit s)) = True

islnf (Or p q) = islnf p and islnf q

islnf (And p q) = islnf p and islnf q

islnf p = False
```

Fig. 3. Test for literal normal form.

Many examples of program synthesis may be found in the literature (Bird and Wadler, 1988; Paulson, 1992). They are based on the method of fold/unfold transformation (Burstall and Darlington, 1977). There are tools that can check the correctness of each step in a formal program synthesis (Runciman et al., 1993). More details of functional languages may be found in the textbooks, such as those by Bird and Wadler (1988), Reade (1989) and Field and Harrison (1988). They all stress the simplicity, logical coherence and ease of formal reasoning in functional languages.

5.2. Assessment of functional languages

The suitability of functional notation for high-integrity software can be assessed relative to Carré's requirements. Functional notation is known to be logically sound with simple semantics. There is considerable expressive power in the algebraic definition of types and equational definition of functions. This notation avoids the many insecurities associated with imperative languages, and type-checking ensures a high degree of security. Verification is relatively straightforward, as illustrated by the example of program synthesis and transformation. However, functional notation has major difficulties with regard to calculation of space and time requirements.

As functional programs are devoid of updatable variables, every value computed during execution must be stored separately, with that storage being reclaimed only when those values are no longer of any use. Consequently, it is difficult to predict space usage. The control flow during the execution of a functional program is not explicit but determined from the evaluation strategy. If a lazy evaluation strategy is used, as is necessary for the functional stream view of input and output, then it is difficult to predict space or time usage. The best that can be done is to observe the space and time costs using profiling techniques (Sansom and Jones, 1995; Runciman and Wakeling, 1993), and then to revise the software accordingly. This is not a predictive approach, and so functional languages are unsuitable for high-integrity application software.

In addition to the problems of space and time, functional languages suffer from lack of acceptance in industry. This is due partly to the lack of quality implementations for industry-standard computers, and partly to their novelty: programming in a functional language can require a different mind-set to programming in an imperative language – idioms that work in an imperative language and upon which a programmer has come to depend do not necessarily transfer to functional languages. Moreover, the space and time costs cannot usually be reduced to those of equivalent imperative programs.

The differences between high-integrity tools and high-integrity application software identified earlier suggest that space and time are less of a problem for high-integrity tools. The expressive data abstraction notation of functional languages inevitably requires use of heap storage in executing programs–the very feature that is excluded from safe subsets of existing languages and needed for high-integrity tools. Yet none of the problems listed for C's interface to heap storage are suffered by functional languages. This observation forms the basis for another approach.

6. Is a combined notation suitable for high-integrity software?

6.1. An approach to high-integrity software development

There is a third approach besides those of defining a new language or a safe subset of an existing language: take an existing language and extend it with notations that satisfy the requirements for high-integrity software. This proposal is in the spirit of Carré's remark (Carré, 1989, Section 5.3.1, p. 109) that "new languages point the way; adaptations of standard languages persuade people to follow".

By introducing to an imperative language an interface to heap storage similar to that used in functional languages, it becomes possible to extend the imperative language with functional notation. The extensions are intended to be used wherever applicable, so that the advantages of the functional notation can be exploited, and the disadvantages of the imperative language avoided. This is a less safe approach than the other two, because use of the unsafe features of the imperative language is not precluded. Nevertheless, it has an important role to play, as argued below.

Our toolkit provides the extensions to C in the same way as the original version of C++ provided its extensions: a preprocessor to translate the new notation to C accompanied by a library of auxiliary routines. The library of auxiliary routines is named *Compost*. There is one preprocessor for algebraic type definitions named *Peat*, and a separate preprocessor for function definitions named *Coir*. Fig. 4 shows how these tools are used in program development. The algebraic type definitions are prepared in separate files and translated by Peat to abstract data types comprising interface specifications (C header files) and operations, which are then compiled into a library. The user program is processed by Coir before being compiled, so as to translate equational definitions of functions to C routines. The compiled code is finally linked with the Compost library to form the runnable program.

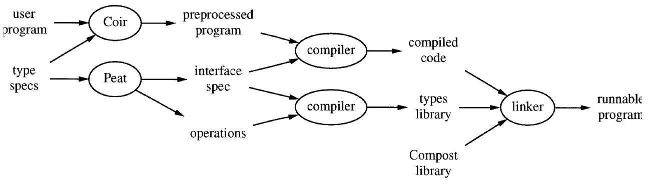

Fig. 4. How the toolkit fits together.

6.2. Compost – The interface to heap storage

Heap storage is managed by the allocation routine `mkcell( )` and garbage collection routine `gc( )`.

```
typedef void *celltype;
typedef unsigned short ord;
extern celltype mkcell (ord cons);
extern void gc(void (*rootsfn)());
extern void collroot (celltype *root);
```

The `mkcell( )` routine takes an ordinal number denoting the value-to-be-stored's constructor (for example, for the type `prop`, this is one of `Lit`, `Or`, `And` and `Not`), and returns a pointer to a suitable area in the heap – a *cell*. The `gc( )` routine reclaims all cells whose values can no longer be used. It determines which values can still be used (and hence those that cannot) by tracing all cells that can be reached from those referred to directly by *root pointers* – the pointers stored in program variables. The routine supplied as the `rootsfn( )` argument to `gc( )` must identify every root pointer by passing the address of each to `collroot( )`. The address is passed rather than the value so as not to preclude the use of a copying garbage collection algorithm. The constructor given when a cell is allocated allows `mkcell( )` to determine the size of the cell and `gc( )` to determine where within the cell are pointers to other cells. Both of these mappings are generated by Peat.

This interface to heap storage avoids all the problems listed for `malloc( )` and `free( )`, so long as the value stored in the cell respects the given constructor. The memory fragmentation problem can be solved by using a garbage collector that compacts the reachable cells (and hence the free space). The details of the garbage collection algorithm are irrelevant to the approach, indeed there can be several alternatives (Compost currently offers two). One new problem is introduced: that of omitting to identify a root pointer. This can cause storage to be reclaimed prematurely. It is avoidable with a functional language, where the run-time system is able to keep track of all root pointers. It cannot be avoided with the combined notation, but there are coding practices that ease its detection.

The ability to traverse data structures of arbitrary types enables several other polymorphic operations. The Compost library provides operations: to preserve the contents of the entire heap in a file, to restore the contents of the entire heap from a file, to write (eagerly or lazily) the binary representation of a heap data structure (Toyn and Dix, 1994), to read the binary representation of a heap data structure, to write an ASCII representation of a heap data structure, and to compare two heap data structures for equality of representation. Note that these are all library routines: they work whatever type of data is in the cells, given suitable descriptions of the data formats as generated by Peat. Compost additionally provides an operation to determine the constructor that was given when a cell was allocated.

```
extern ord cellcons(celltype cell);
```

Further details of Compost may be found in its user manual (Toyn, 1994a).

6.3. Peat – The algebraic type definition preprocessor

Peat translates each algebraic type to an abstract data type comprising a C type representation and a collection of operations. These operations are named by composing constructor name, attribute name and type name in various combinations.

Consider again the `prop` type of propositions, here written in notation accepted by the Peat preprocessor.

```
prop  : Lit sym
      | Or prop1:prop prop2:prop
      | And prop1 prop2
      | Not prop
```

The two disjuncts in the `Or` construct are shown with distinct names to distinguish them. The two conjuncts in the `And` construct illustrate a permitted abbreviated notation: the form as written is taken as the name, and trailing digits are stripped to determine the type.

The abstract data type that Peat generates has operations that are named according to the rules given in Fig. 5. Other naming conventions could work just as well, for instance some users might prefer more underscore characters.

For the `prop` example, the type of pointers to the representation is named `proptype`. There are allocators named `mklitprop`, `mkorprop`, `mkandprop`

Type name:	`<type>type`
Allocators:	`mk<constructor><type>`
Constructor constants:	`n_<constructor><type>`
Constructor projector:	`<type>cons`
Constructor predicates:	`is<constructor><type>`
Attribute projectors:	`<constructor><type><attribute>`
Attribute updators:	`set<constructor><type><attribute>`

Fig. 5. Templates for names generated by Peat.

and `mknotprop`. These take parameters in the order given in the algebraic type definition, and return a value of type `proptype`. Constructors are denoted by the constants named `n_litprop`, `n_orprop`, `n_andprop` and `n_notprop`. These constants are passed in calls to `mkcell( )` by the allocators. Projectors are defined to extract attributes, named `litpropsym`, `orpropprop1`, `orpropprop2`, `andpropprop1`, `andpropprop2` and `notpropprop`. Predicates are defined for testing for particular constructs named `islitprop`, `isorprop`, `isandprop` and `isnotprop`, but more useful is the operation `propcons` which returns one of the constructor constants and so can be used in a multi-way branch statement. Also defined are updators for overriding the values of particular attributes, named `setlitpropsym`, `setorpropprop1`, `setorpropprop2`, `setandpropprop1`, `setandpropprop2` and `setnotpropprop`.

The implementation of the above abstraction in C generated by the Peat preprocessor is shown in Figs. 6–8. Not shown is the code generated by Peat to assist Compost in traversing data structures. All this

```
typedef union propunion *proptype;
union propunion {
        struct {
                symtype sym;
        } litprop;
        struct {
                proptype prop1;
                proptype prop2;
        } orprop;
        struct {
                proptype prop1;
                proptype prop2;
        } andprop;
        struct {
                proptype prop;
        } notprop;
};
```

Fig. 6. Representation type generated by Peat.

```
#define n_litprop       (ord)6
#define n_orprop        (ord)7
#define n_andprop       (ord)8
#define n_notprop       (ord)9

#define propcons(x)     cellcons(x)

#define islitprop(x)    (cellcons(x) == n_litprop)
#define isorprop(x)     (cellcons(x) == n_orprop)
#define isandprop(x)    (cellcons(x) == n_andprop)
#define isnotprop(x)    (cellcons(x) == n_notprop)

#define setlitpropsym(x,y)      (x)->litprop.sym = y
#define setorpropprop1(x,y)     (x)->orprop.prop1 = y
#define setorpropprop2(x,y)     (x)->orprop.prop2 = y
#define setandpropprop1(x,y)    (x)->andprop.prop1 = y
#define setandpropprop2(x,y)    (x)->andprop.prop2 = y
#define setnotpropprop(x,y)     (x)->notprop.prop = y

#define litpropsym(x)   (x)->litprop.sym
#define orpropprop1(x)  (x)->orprop.prop1
#define orpropprop2(x)  (x)->orprop.prop2
#define andpropprop1(x) (x)->andprop.prop1
#define andpropprop2(x) (x)->andprop.prop2
#define notpropprop(x)  (x)->notprop.prop
```

Fig. 7. Operations defined by macros generated by Peat.

generated code is compiled into a library for linking into programs that use the types. The quantity of code generated by Peat reflects the expressive power of the functional notation.

The generated `typedef` allows the C compiler to check that these operations are used in a type-correct manner, and hence helps to ensure that "the value stored in the cell respects the given constructor" as required by Compost. The C compiler cannot detect all misuses of the abstractions where polymorphic types are concerned; this problem is avoided by Coir, as explained in Section 6.4.

Some other interesting features of the Peat notation that are not illustrated by the `prop` example are worth mentioning.

Common attributes are attributes needed by every construct in an algebraic type. Peat provides an abbreviated notation for defining them, and special operations in the abstraction so that they can be projected and updated without naming a particular construct.

Initial values may be specified for attributes in algebraic types. Values for such attributes are not passed as arguments to the allocators. Instead, the allocators set the attributes to the initial values. These values can be overridden using the updator operations later. This notation is convenient with data types representing the abstract syntax of a language: attributes whose values are deduced by a parser are passed to the allocator, while other derived attributes such as type and binding information are set later once they have been computed.

```
proptype
mklitprop(symtype sym)
{
        register proptype z;

        z = mkcell(n_litprop);
        setlitpropsym(z, sym);
        return (z);
}
proptype
mkorprop(proptype prop1, proptype prop2)
{
        register proptype z;

        z = mkcell(n_orprop);
        setorpropprop1(z, prop1);
        setorpropprop2(z, prop2);
        return (z);
}
proptype
mkandprop(proptype prop1, proptype prop2)
{
        register proptype z;

        z = mkcell(n_andprop);
        setandpropprop1(z, prop1);
        setandpropprop2(z, prop2);
        return (z);
}
proptype
mknotprop(proptype prop)
{
        register proptype z;

        z = mkcell(n_notprop);
        setnotpropprop(z, prop);
        return (z);
}
```

Fig. 8. Allocator functions generated by Peat.

A *repeated attribute* provides array-like functionality within an algebraic type. The projector and updator operations have an extra index argument, and the attribute must have an initial value which is assigned to every element in the array.

The scheme used by Peat for generating operation names has no trouble coping with the same name being used for different constructors, so long as they are for different types. This *constructor overloading* is an extension to the usual functional notation that is sometimes very convenient.

Peat recognizes certain type names as denoting primitive C types, e.g. `nat` denotes `unsigned long`. Attributes having such types have their values stored in so-called unboxed representation, i.e. immediately within a cell rather than via a pointer to a separate cell. The details of this representation can be hidden by the abstraction.

A difficulty with tools such as Peat that generate large abstractions comprising many operations is how to organize the code for efficient compilation. If the interface specification for each type is generated in a separate file, then the programmer's source files will be cluttered with many inclusions of interface specifications. On the other hand, if a single file is generated, every source file will take a long time to compile, and any symbolic information retained to assist run-time debugging will be enormous. Peat permits the programmer to engineer a compromise: the `typedef` declarations all go in a single file for inclusion first so as to allow mutually recursive types, but the rest of the interface specifications can be clustered however the programmer chooses.

Further details of Peat may be found in its user manual (Toyn, 1994b).

6.4. Coir – The equation preprocessor

Coir looks for so-called *rewrite functions* in C source files and translates them to C routines. The following example shows the first definition of `lnf` coded as a rewrite function for Coir.

```
rewrite lnf: prop -> prop
{    Or p q => Or {lnf p} {lnf q}
\/    And p q => And {lnf p} {lnf q}
\/    Not (Not r) => {lnf r}
\/    Not (Or r s) => And {lnf (Not r)}
{lnf (Not s)}
\/    Not (And r s) => Or {lnf (Not r)}
{lnf (Not s)}
\/    p => p
}
```

The `rewrite` keyword marks the start of a rewrite function. It is followed by type declarations for all static (top-level) identifiers used within the body of the rewrite function. Note that braces are used to distinguish applications of static identifiers from applications of constructors.

Coir's translation to C is based on the work of Augustsson (1985). The C code generated makes use of the operations defined by Peat. For the `lnf` function, the code generated by Coir is a little verbose, so to save space here a simpler equivalent is shown in Fig. 9. The expressive power of the functional notation is reflected in the relative closeness to the specification in Fig. 1 of the rewrite function given to Coir compared with that of the generated C code.

As another example, consider the `treesum` function defined earlier. It is coded as a rewrite function for Coir in Fig. 10. The initial design for Coir hoped to intermingle C notation and rewrite functions rather more than the code in that figure, so that the separate `add( )` routine would not be needed. However, this proved to be incompatible with the wish to perform polymorphic type-checking of rewrite functions. Cattrall's algorithm (Cattrall and Runciman, 1992, 1993) is used for type-checking in Coir. It is an extension of Milner's (Milner, 1978) (necessary to cope with constructor overloading), and has been proven to be complete and sound (Cattrall, 1993). This achieves the same degree of security as a type-checker for a functional language.

```
proptype lnf(proptype x)
{
   switch(propcons(x)) {
   case n_orprop: {
      proptype p = orpropprop1(x);
      proptype q = orpropprop2(x);
      return (mkorprop(lnf(p), lnf(q)));
   }
   case n_andprop: {
      proptype p = andpropprop1(x);
      proptype q = andpropprop2(x);
      return (mkandprop(lnf(p), lnf(q)));
   }
   case n_notprop: {
      proptype q = notpropprop(x);
      switch (propcons(q)) {
      case n_notprop: {
         proptype r = notpropprop(q);
         return (lnf(r));
      }
      case n_andprop: {
         proptype r = andpropprop1(q);
         proptype s = andpropprop2(q);
         return (mkorprop(lnf(mknotprop(r)), lnf(mknotprop(s))));
      }
      case n_orprop: {
         proptype r = orpropprop1(q);
         proptype s = orpropprop2(q);
         return (mkandprop(lnf(mknotprop(r)), lnf(mknotprop(s))));
      }
      }
   }
   default: return x;
   }
}
```

Fig. 9. Literal normal form transformation using Peat.

```
rewrite treesum : (tree nat) -> nat
fun add : <nat,nat> -> nat
{
        Leaf (Num n) => n
\/      Node x y => {add {treesum x} {treesum y}}
}

nat add(nat x, nat y)
{
        return (x + y);
}
```

Fig. 10. Summing the leaves of a tree using Coir.

Some other interesting features of the Coir notation that are not illustrated by the above examples are worth mentioning.

As patterns are a common functional language notation for expressing that an equation is applicable only if the pattern matches despite the fact that the transformation performed by the equation on the matching value is merely an identity. From the implementation perspective, an as pattern eases the retention of the original value, rather than rebuilding a new value from matched pieces.

Wildcard or anonymous patterns are another common notation. They match any value without introducing a name for the value, which is useful where an equation does not incorporate that value in its result.

Guards are also a common notation. They allow a computational condition to be associated with an equation in addition to the structural condition imposed by pattern matching. Coir guards must be Boolean-valued applications of named C routines.

Constructor-polymorphism is a less conventional Coir notation for expressing equations over an explicit set of different types. It is especially useful in expressing commutativity and associativity transformations, where it allows a single equation to deal with many operators.

6.5. Assessment of the combined notation

The combined notation supports the algebraic definition of types and the specification of functions by equations, based on a much safer interface to heap storage than C's standard interface. The functional notation imposes an effective discipline on the use of C's heap storage, type coercions and pointers. It is slightly less secure in this combination than it is in a functional language: there is the root pointer identification problem with Compost, and Coir's assumption that type declarations for static identifiers match the types of those identifiers, definitions (checking this would require Coir to parse all of C, rather than starting to parse at the `rewrite` keyword). The combined notation does, however, offer a more expressive and more secure style of programming than C alone, while clearly maintaining the space and time efficiency advantages of C.

The implementation of the functional notation by preprocessors is not the best approach from the point of view of debugging, which has to be done in terms of preprocessed source rather than original source. However, debugging is nevertheless feasible, thanks to the mnemonic names generated by Peat.

None of the languages that have been assessed as suitable for high-integrity software provide heap storage. Yet for high-integrity tools, heap storage is a necessity. The combination of a safer interface to heap storage with an efficient systems programming language is an appropriate choice for developing high-integrity tools.

This pragmatic approach, exemplified by the combination of functional notation with C, should be applicable to combinations of functional notation with other imperative languages.

7. Some applications of the combined notation

7.1. The CADiZ *tools for Z specifications*

CADiZ (Toyn and McDermid, 1995) is a set of tools for manipulating Z specifications. CADiZ includes parsers, a type-checker, typesetters, a browser and a proof tool to assist in reasoning about properties of the specifications. Compost and Peat were developed in tandem with CADiZ, and hence there is great dependence from CADiZ on Compost and Peat. Coir, however, was begun much later, after most of CADiZ had been written, and so is used little. CADiZ has been very successful, receiving one of the three British Computer Society (BCS) Awards in 1992.

Almost all of CADiZ's data structures are managed by Compost and Peat, most notably the abstract syntax tree representing the user's Z specification. A functional style of programming is used throughout the CADiZ tools, but with imperative style used where the benefits (convenience or efficiency) outweigh the insecurities.

Coir was expected to be suitable for expressing transformations in CADiZ's proof assistant tool, but this use was disappointing. The problem was more to do with the functional notation than with Coir, it being that CADiZ represents conjectures, goals, theorems, laws, etc. by sequent constructs having about 13 attributes: patterns matching such large constructs are simply not readable, nor are they easy to maintain when the type is changed. One solution would be to alter the data structures to ease use of Coir, but there is no great motivation to change: the Peat notation is quite readable and very easy to maintain.

7.2. The Ten15 Distribution Format installer

Ten15 Distribution Format (DRA, 1993) (TDF) is a porting technology and hence is part of a shrink-wrapping, distribution and installation technology. TDF has been chosen by the Open Software Foundation as the basis of its Architecture Neutral Distribution Format. It was developed by the United Kingdom's Defence Research Agency (DRA). DRA are working with Unix System Laboratories to commercialise the TDF technology. TDF is not UNIX specific, although most of the implementation has been done on UNIX.

Software vendors, when they port their programs to several platforms, usually wish to take advantage of particular features of each platform. That is, they wish versions of their programs on each platform to be functionally equivalent, but not necessarily algorithmically identical. TDF is intended for porting in this sense. It is designed so that a program in its TDF form can be systematically modified when it arrives at the target platform to achieve the intended functionality and to use the algorithms and data structures that are appropriate and efficient for the target machine. A fully efficient program, specialised to each target, is a necessity if independent software vendors are to adopt a porting strategy.

The TDF compiler (or *installer* as it is called) is coded in C and one of its components is an optimiser that performs TDF-to-TDF transformations. Coir was developed for, and used to re-implement, this component of the installer.

Since the TDF installer was an existing program, with its own memory management code and data type representations, Compost and Peat could not be used. However, the abstraction defined by Peat and used by Coir proved to be a suitable interface between Coir and the TDF installer. By hand-coding an implementation of this interface in terms of the TDF installer's existing data representations, Coir was made usable in the absence of Compost and Peat.

The use of Coir to recode TDF transformations has made the code clearer, more readable and easier to modify. Confidence in integrity has been increased: one major transformation has been derived from an inductive proof. The cost of executing the TDF installer has risen by an acceptable amount: for quite large programs (5000–8000 lines), execution time increased by between one-quarter and one-third.

8. Related work

Compost, Peat and Coir are based on technology that is widely used amongst implementors of functional programming systems (Peyton Jones, 1987; Augustsson, 1985).

There have been many tools that take data type specifications and generate programming language implementations. Perhaps the best known is Interface Definition Language (IDL) (Lamb, 1987). As its name suggests, the translator for IDL specifications generates not just representations for data types but also readers and writers for communicating data structures. The designer provides IDL with both abstract descriptions of data types and also representation specifications for use in generating representations tailored to the needs of specific programs. This contrasts with Compost and Peat, where a single representation is generated automatically from a more abstract specification by Peat, and single reader and writer operations in Compost cope with all data types.

Coir's use for TDF transformations is related to various tools for transforming syntax trees. Estra (Grosch and Emmelmann, 1990) and OPTRAN (Lipps et al., 1988) are similar to Coir in that they express transformations by rules. Each rule consists of a pattern describing a tree fragment, a condition or predicate to restrict the applicability of the rule (like a Coir guard), and an action or output description. Estra and OPTRAN differ from Coir in that they use a search procedure to find trees to which rules are applicable. Estra offers a choice of two pattern matchers: a dynamic programming algorithm and a table-driven pattern matcher. OPTRAN allows user-defined search procedures. Coir leaves the search to the programmer, who must decide which rule to apply. A major difference between Coir and the Estra and OPTRAN tools is that Coir is a pattern matching *compiler* – the appropriate rule is found by the control flow passing through C `switch` statements – which obviously aids efficiency.

9. Conclusions

This paper has shown how functional notation can be combined with C, based on a different interface to heap storage and abstract data types for manipulating values of algebraically defined types. The approach can be supported by a toolkit comprising a library of routines offering relatively secure heap storage, and separate preprocessors for algebraic definition of types and equational definition of functions. Two large real-world applications of the toolkit have been discussed, revealing benefits from use of only subsets of the toolkit.

The combination of notations offers a practical compromise between the integrity of functional languages and the efficiency of C. The functional notation is logically sound, has a simple semantics, considerable expressive power, is relatively secure and verifiable. C has a direct implementation on conventional von Neumann hardware thus offering efficient execution. The combined notation's basis on heap storage makes it more appropriate for the construction of high-integrity tools than recognised languages for high-integrity software.

This pragmatic approach of introducing functional notation into C should help to persuade programmers toward the more reliable style of programming offered by functional languages. We believe that the same pragmatic approach should be applicable to combinations of functional notation with other imperative languages.

Acknowledgements

Philip Core and Michael Foster of DRA encouraged the development of Coir. Comments from Jonathan Moffett, Jon Hall and Stephen Wilson on previous drafts led to this improved presentation. DRA and SERC SCHEMA project (GR/G49531) provided funding.

References

Anderson, T. (Ed.), 1989. Dependability of Resilient Computers. Blackwell Professional Books, Oxford.

Augustsson, L., 1985. Compiling pattern matching. In: Jouannaud, J.-P. (Ed.), Functional Programming Languages and Computer Architecture, Nancy, France. Lecture Notes in Computer Science, vol. 201. Springer, Berlin, pp. 368–380.

Bird, R., Wadler, P., 1988. Introduction to Functional Programming. Prentice-Hall, Englewood Cliffs, NJ.

Bundy, A., 1983. The Computer Modelling of Mathematical Reasoning. Academic Press, New York.

Burstall, R., Darlington, J., 1977. A transformation system for developing recursive programs. Journal of the ACM 24 (1), 44–67.

Cardelli, L., 1987. Basic polymorphic typechecking. Science of Computer Programming 8 (2), 147–172.

Carré, B., 1989. Reliable programming in standard languages. In: C. Sennett (Ed.), High-integrity Software, Computer Systems Series, Ch. 5. Pitman, London, Program Validation Ltd.

Carré, B., Jennings, T., Maclennan, F., Farrow, P., Garnsworthy, J., 1988. SPARK – The SPADE Ada Kernel, 3.1 ed, Program Validation Ltd.

Cattrall, D., 1993. The design and implementation of a relational programming system. D.Phil. Thesis YCST 93/01, Department of Computer Science, University of York, England.

Cattrall, D., Runciman, C., 1992. A relational programming system with inferred representations. In: Bruynooghe, M., Wirsing, M. (Eds.), Fourth International Symposium, Programming Language Implementation and Logic Programming, Leuven, Belgium. Lecture Notes in Computer Science, vol. 631. Springer, Berlin, pp. 475–476.

Cattrall, D., Runciman, C., 1993. Widening the representation bottleneck: A functional implementation of relational programming. In: Williams, J. (Ed.), FPCA'93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark. ACM, New York, pp. 191–200.

Chapman, R., Burns, A., Wellings, A., 1994. Integrated program proof and worst-case timing analysis of SPARK Ada. In: Pugh, W. (Ed.), PLDI'94: ACM workshop on language, compiler and tool support for real-time systems. ACM, New York.

Cullyer, W., Goodenough, S., Wichmann, B., 1991. The choice of computer languages for use in safety-critical systems. Software Engineering Journal 6 (2), 51–58.

DRA, 1993. TDF Specification. St Andrews Rd, Malvern, Worcestershire: Defence Research Agency. Issue 2.1.0.

Field, A., Harrison, P., 1988. Functional Programming. Addison-Wesley, Reading, MA.

Garnsworthy, J., O'Neil, I., Carré, B., 1993. Automatic proof of the absence of run-time errors. In: Collingbourne, L. (Ed.), Ada: Towards Maturity. IOS Press, Amsterdam, pp. 108–122.

Grosch, J., Emmelmann, H., 1990. A tool box for compiler construction. In: Hammer, D. (Ed.), Compiler Compilers, Third International Workshop, CC'90, Schwerin, FRG. Lecture Notes in Computer Science vol. 477. Springer, Berlin, pp. 106–116.

Jordan, D., Locke, C., McDermid, J., Parker, C., Sharp, B., Toyn, I., 1994. Literate formal development of Ada from Z for safety critical applications. In: Proceedings of SAFECOMP'94, Instrument Society of America, pp. 1–10.

Kernighan, B., Ritchie, D., 1988. The C Programming Language, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ.

Lamb, D., 1987. IDL: Sharing intermediate representations. ACM Transactions on Programming Languages and Systems 9 (3), 297–318.

Lipps, P., Möncke, U., Wilhelm, R., 1988. OPTRAN – a language/system for the specification of program transformations: System overview and experiences. In: Compiler Compilers and High Speed Compilation. 2nd CCHSC Workshop, Berlin. Lecture Notes in Computer Science, vol. 371. Springer, Berlin, pp. 52–65.

Milner, R., 1978. A theory of type polymorphism in programming. Journal of Computer and System Sciences 17 (3), 348–375.

MoD, 1991a. The procurement of safety critical software in defence equipment. INTERIM Defence Standard 00-55 (PART 1: REQUIREMENTS)/Issue 1.

MoD, 1991b. The procurement of safety critical software in defence equipment. INTERIM Defence Standard 00-55 (PART 2: GUIDANCE)/Issue 1.

MoD, 1993. Safety management requirements for defence systems containing programmable electronics. Draft Defence Standard 00-56.

Morgan, C., 1994. Programming from Specifications, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ.

Paulson, L., 1992. ML for the Working Programmer, Cambridge University Press, Cambridge.

Jones, S.P., 1987. The Implementation of Functional Programming Languages. Prentice-Hall, Englewood Cliffs, NJ.

PVL, 1994. Formal Semantics of SPARK. Program Validation Limited (2 volumes).

Reade, C., 1989. Elements of Functional Programming. Addison-Wesley, Reading, MA.

RTCA, 1992. Software considerations in airborne systems and equipment certification. Prepared by RTCA SC-167/EUROCAE WG-12.

Runciman, C., Toyn, I., Firth, M.A., 1993. Incremental, exploratory and transformational environment for lazy functional programming. Journal of Functional Programming 3 (1), 93–115.

Runciman, C., Wakeling, D., 1993. Heap profiling of lazy functional programs. Journal of Functional Programming 3 (2), 217–245.

Sansom, P., Jones, S.P., 1995. Time and space profiling for non-strict higher-order functional languages. In: Proceedings of the 22nd Annual ACM Symposium on the Principles of Programming Languages.

Sennett, C. (Ed.), 1989. High-integrity Software. Computer Systems Series. Pitman, London.

Spivey, J., 1992. The Z Notation: A Reference Manual, 2nd ed. Prentice-Hall, Englewood Cliffs, NJ.

Toyn, I., 1994. Compost – The BIO Heap. Department of Computer Science, University of York.

Toyn, I., 1994b. Peat – A Program to Elaborate Algebraic Types. Department of Computer Science, University of York.

Toyn, I., Dix, A.J., 1994. Efficient binary transfer of pointer structures. Software – Practice and Experience 24 (11), 1001–1023.

Toyn, I., McDermid, J., 1995. CADiZ: An architecture for Z tools and its implementation. Software – Practice and Experience 25 (3), 305–330.

Ian Toyn is a Research Fellow in the High-Integrity Systems Engineering group. He has B.Sc. and D.Phil., degrees in Computer Science from the University of York. Having worked previously in the areas of functional programming and human–computer interaction, he has since 1989 worked on the CADiZ tools for manipulating Z specifications, and is a member of the BSI/ISO panel that is preparing the Z standard.

Dave M. Cattrall gained a B.Sc. in Computer Science from Lancaster University in 1989 and a D.Phil., in Computer Science from the University of York in 1993. He then worked for two years in the Software and Systems Engineering Division of Northern Telecom. Currently he works for Logica UK. He is the author of eight papers on topic as diverse as functional and logic programming, high-integrity systems engineering, feature interactions in telecommunication networks, and ATM telecommunication networks.

John A. McDermid is Professor of Software Engineering at the University of York, where he runs the High-Integrity Systems Engineering group. His main interests are in software engineering for safety critical systems, with application in aerospace; he directs the BAe Dependable Computing Systems Centre (DCSC), and the Rolls-Royce University Technology Centre (UTC) in Systems and Software Engineering. He is also a Director of York Software Engineering Ltd., which provides tools and consultancy in the area of high-integrity systems.

Jeremy L. Jacob has a B.Sc. in Mathematics from Hull, and an M.Sc. and a D.Phil. in Computation from Oxford. He has worked mostly in the field of formal methods, particularly in the domain of security. He believes that with appropriate tool support formal methods have much to offer the development of safety critical systems.