Evaluation of expressions in a multiparadigm framework[1]

Xavier Burgués, Xavier Franch
{diafebus, franch}@lsi.upc.es
Dept. Llenguatges i Sistemes Informàtics (LSI)
Universitat Politècnica de Catalunya (UPC)
Pau Gargallo 5, 08028 Barcelona, Catalunya (Spain)
FAX: 34-3-4017014. Phone: 34-3-4017006

Abstract

A framework that combines imperative programs with algebraic specifications is presented. We focus on an execution mechanism that allows the arbitrary combination of programs and equations, which is based on the evaluation of expressions by means of the transformation of values from data structures to terms and vice versa, using the abstraction function of the implementation. The correctness of the execution procedure follows from a set of basic transformation rules.

1. Introduction

Software development with abstract data types is usually done by first specifying the types and next implementing them. Specifications may be formal (equational) or not, and in the first case they may be executable or not. In this framework, two main questions arise: how can we prove the correctness of an implementation with respect to its specification? and, is there any help to transform a specification in an implementation?

Roughly speaking, we may distinguish between formal verification and testing for the first question. Formal verification is better than testing because it assures the correctness of programs; however, it may be difficult to carry out, even with the help of appropriate tools, and this is why testing is still in use. Concerning the second question, specifications and implementations may be kept separated in different stages, interfering with software evolution, or else they may be joined together with a multiparadigm (wide-spectrum) language (which may be really a single language or else the combination of two or more existing languages into a single framework); in the second case, evolution may be performed with the help of some transformation rules or not.

In this paper, we propose a multiparadigm language combining equational specifications and imperative programs into a single framework. Even more, we require specifications to be operational (executable), and thus we adopt a testing approach to implementation correctness. To provide programmers with a great flexibility, we build an execution procedure that permits execution of software that combines specifications and imperative programs in an arbitrary manner; as a result, we may adopt a prototyping software development model in which a program is obtained by applying a sequence of refinement steps starting from a formal, possibly incomplete specification of the problem, being able to prototype the product at every stage of development by combining a term-rewriting system and an interpreter.

We remark that our prototyping strategy does not prevent the use of a transformational approach. On the other hand, this kind of testing may be sometimes employed as a complement of formal verification to show the execution behaviour of a piece of software (for instance, to study interfaces with the user). Also, we remark the flexibility of the strategy, that does not oblige neither the specification of a data type before its implementation (we may think in those procedural aspects which specification really looks as an implementation) nor any fixed

[1]This work is partially supported by the spanish research programme PRONTIC under contract TIC92-0667.

"implementation rate" (implementations may be built and tested at once, or function by function); in any case, execution is always possible.

The rest of the paper is organised as follows. In section 2, we present the multiparadigm notation used in the paper. In section 3, it is shown how may we customise the execution environment. In section 4, the execution procedure is presented from some demonstrably correct transformation rules, and an example is shown in section 5. Finally, section 6 remarks some undesirable issues of the work, and section 7 gives the conclusions and future work.

2. A multiparadigm language

The ideas we present in this paper are intended not to be tied to any concrete programming language: they may be applied to a specification language as OBJ [FGMJ85] or ACT ONE [EM85], linked with any imperative programming language as Ada, Modula or O.-O. ones. Also, they may be applied to any equational enrichment of an imperative programming language (Ada with equations, as Anna [LH85]), or (with a few modifications) even to logic programming combined with imperative programming (as in [DDL88, Rad90]).

To be as general as possible, we use in the paper an *ad hoc* notation based on the concept of abstract data type as design unit. It allows: on the one hand, formal, possibly incomplete specification of types as a previous step to their implementation; on the other hand, gradual implementation of those types supported by a programming environment, which is able to incrementally execute programs, even if they do combine specified types with implemented ones. Data type encapsulation mechanisms are called **universes**; they may define new types (probably parameterised), or enrichments of types by adding new operations on some existing ones; depending on their contents, we talk about **specification universes** or **implementation universes**.

Specification universes present some constructs to define type signatures (that is, types' names -called sorts- and operations) and properties. Properties are stated in equational style; equations may be conditional, the conditions in turn being equations. The logic to express equations allows to associate **initial semantics** to specifications [GTW78] and thus it is possible to use a term-rewriting system to manipulate terms during execution of programs. Implementation universes' notation presents Ada-like programming language constructs. Each implementation universe must be referred to a specification one (which presents at least the signature of the type) and it may include data structures and/or functions with no side-effects. In many contexts, prototyping can take place only if there exists an abstraction function [Hoa72] to find out which value of a type is represented by a particular state of the implementation. Thus, implementation universes should provide this function in order to be able to prototype the execution in any intermediate state; in spite of being defined in implementation universes, the abstraction function may be stated either in equational or imperative style.

In both kinds of universes some structuring mechanisms can be employed. The basic one is the use of specifications, which makes all the symbols exported by a specification universe available to its users. There is also a parameterisation facility to define generic universes, coupled with a parameter passing mechanism (instantiation); formal parameters are defined in the so-called **characterisation universes**. Last, there are some constructs to control the scope of symbols in specifications: symbols may be renamed, hidden or introduced as private from the very beginning.

As an example, we give the highlights of the specification of a generic type for directed, unlabelled graphs and its implementation with adjacency matrix. In the specification, the parameters are the node's type *elem* and an equality operator *eq* (to specify membership), both of them defined in the characterisation universe *ELEM_=* (the first one as a sort and the second one as an operation which properties are stated). *empty* and *add* are the so-called *generating operations*, which "generate" all the values of the type and that serve as the basis for the specification of the other operations in the initial semantics framework. Note the use of specifications of booleans, the definition of auxiliary functions, the instantiation and renaming of a generic type for lists in order to define the signature, the specification of erroneous situations and the use of conditional equations. The abstraction function is given in imperative style; when this is the case, a variable of a generic type *term* is used to hold the result. Also, a representation invariant is provided, which may be used during execution.

```
universe DIGRAPH (ELEM_=) defines
    uses BOOL
    type digraph
    instantiates LIST(ELEM)
        where elem is ELEM_=.elem
        renames list by list_of_nodes
    ops
        empty: → graph
        add: graph elem elem → graph
        private remove: graph elem elem → graph
        …
        succ: graph elem → list_of_nodes
    eqns g: graph; v, w, x, y: elem
        add(add(g, v, w), v, w) = add(g, v, w)
        add(add(g, v, w), x, y) = add(add(g, x, y), v, w)
        …
        succ(empty, v) = LIST.empty
        succ(add(g, v, w), v) =
            LIST.put(succ(remove(g, v, w), v), w)
        [eq(v, x) = false] ⇒ succ(add(g, v, w), x) = succ(g, x)
end universe
```

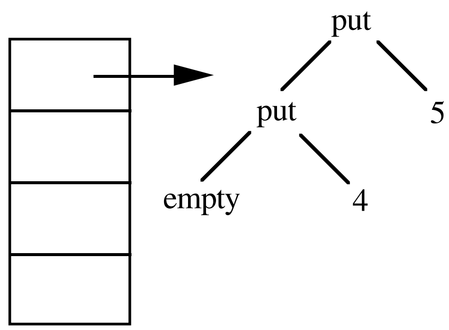
```
universe ADJ_MATRIX(ELEM_=)
    implements DIGRAPH(ELEM)
    uses BOOL
        repr
            type digraph is
                array [elem, elem] of bool
            abstr (g: digraph) → t: term
                t := empty
                for all v, w in elem do
                    if g[v, w] then
                        t := add(t, v, w)
            inv (g: digraph)
                for all v in elem: ¬ g[v, v]
    function empty →  g: digraph is
    var v, w: elem
        for all v, w in elem do
            g[v, w] := false
        end for all
    end function
    …
end universe
```

```
universe ELEM_= characterizes
    uses BOOL
    type elem
    ops eq: elem elem → bool
    eqns v, v1, v2, v3: elem
        eq(v, v) = true
        [eq(v1, v2) = true] ⇒ eq(v2, v1) = true
        [eq(v1, v2) ∧ eq(v2, v3) = true] ⇒ eq(v1, v3) = true
end universe
```

In the general case, there may be many levels of implementations; for instance, graphs may be also implemented by (adjacency) lists, which may implemented by arrays in turn. Anyway, the bottom level consists of predefined types like arrays and integers.

Last, let's make clear some language concepts from the execution point of view. First of all, there is no main program in applications; instead, we may pick up any function to be the subject of execution at any time. As a result, we may assimilate the notion of execution to the idea of evaluating a term, that is, the application of a function over its parameters. Throughout the paper, a term is called an **expression** when it is used in a context that requires its evaluation; note that an expression may contain many proper sub expressions. Inside expressions, objects do appear which will be evaluated in the appropriate moments (we denote by the word **object** any of the following entities: variables, parameters, constants -predefined or user-defined- and function results). As a key point during execution, objects' value may be either a data structure or a term; in the first case, we talk about **implemented objects**, while we call **specified**

objects the second ones. Note that objects may be intertwined due to the existence of many levels of implementation; that is, an object of type *graph* may be implemented with an array, being their components of type *list* which may be specified (if there not exist an implementation for it), and elements inside lists (for instance, integers) may be implemented in turn. In this case we talk about **hybrid objects**. The figure below illustrates this situation.



3. Customisation of execution

As far as we have adopted a prototyping development, the aim of the execution system is to offer the possibility of testing the current state. So we mainly pay attention to this goal and not to efficiency issues, which will be of interest once prototyping has finished and a fully implemented program has been obtained.

In order to guide program execution, we should set the value of two attributes related to each function:

- Unit under test (UUT). This attribute has three possible values: *specification*, *implementation* and *none*. In the first two cases, we say that the function is **under test**.
- Method of execution (ME). This attribute takes effect only if the function is not under test. Possible values for this attribute are *rewriting* and *interpretation*.

Execution of an expression *f(...)* with UUT(*f*) = *specification* means rewriting of *f(...)* using some rewriting rules. Execution of *f(...)* with UUT(*f*) = *implementation* means interpretation of its code. Execution of *f(...)* with UUT(*f*) = *none* means execution of *f(...)* depending on ME(*f*).

By default, a function *f* has UUT(*f*) = *specification* until an implementation for that function is built. At this moment, UUT(*f*) becomes *implementation* until we declare that *f* is not under test by assigning *none* to UUT(*f*). Then, ME(*f*) becomes equal to *interpretation*. This default behaviour may be tuned with the help of the programming environment. Predefined functions of the language, such as manipulation of data structures, are not under test and they are executed using their built-in implementation.

With these attributes, the most relevant facts of the execution process are:

- All appearances of a function under test will be executed using their UUT value, even if their execution is not strictly necessary, because we are focusing on testing the correct behaviour of the unit. For example, let's suppose that we are going to test a stack of integers with operations *empty*, *push*, *pop* and *top* such that we have the specification and an implementation of *empty* and *push*, together with a representation of the stack. If the attributes of execution have their default values, the evaluation of the expression *pop(push(empty, 1))* will be carried out on these steps:

    1.- interpretation of *empty* obtaining the appropriate value of the representation.

    2.- interpretation of *push*, obtaining another value of the representation.

3.- abstraction of the result, obtaining the term *push(empty, 1)*.

4.- rewriting of the term *pop(push(empty, 1))* to *empty*.

5.- interpretation of *empty* obtaining the appropriate value of the representation.

Note that steps 1, 2, 3 have no computational effect. In terms of efficiency, they should be avoided, but in terms of testing they are interesting. Such an inefficient behaviour of the execution system is intentionally enabled in order to perform testing as far as possible during execution. Note also that step 5 gives no new results from the testing point of view and so it should be avoided (as we propose in section 4).

Some other efficiency aspects (such as those performed by an optimiser) could be added to the system, but we do not address to them at this stage of the project.

- Functions not under test will possibly be simplified with rewriting and, hence, not executed. With the same example as before, but assuming that *empty* and *push* are not under test, the evaluation of the expression will simply perform step 4 and, depending on the context, step 5.

4. Execution procedure

Execution begins when we choose a function to start with, which yields to the evaluation of an expression. That is, we want to compute the value corresponding to its result. But, in fact, after substitution of its variables, the expression becomes a term denoting a value (its result) and, then, execution will apply a sequence of transformations on this term to obtain another, simpler one, denoting the same value. When we do not want to (or we cannot) apply more transformations to a term we say that it is in its **final form**[2].

These transformations will be carried out using specifications and implementations altogether. We ensure the correctness of the process in the framework of the ideas of [Hoa72, EKMP82]. For each user-defined type T (the initial model associated to its specification E), there is a type T_i (the initial model corresponding to its implementation I). If the implementation is correct, for each function f on T there is a function f_i on T_i (with the same signature as f substituting specification sorts by the corresponding implementation ones), together with an abstraction function $abstr_i$ and a representation invariant inv_i. Moreover, there is an isomorphism between T and T_i (to be more precise, the restriction of T_i to those values satisfying the invariant *modulo* the equivalence induced by the abstraction function) such that $f(x) = abstr_i(f_i(concr_i(x)))$ holds, where $concr_i$ stands for the inverse of $abstr_i$ and the symbol = stands for the equational calculus. If the implementation is partial (not all functions are implemented) the relation stated above is still valid for the implemented functions.

The transformations allowed by the theoretical framework we have adopted are the following:

- **T1a**. If the type of f is defined in the same universe as f, substitute $f(...)$ by $abstr_i(f_i(...concr_i(...)...))$,

  T1b. If the type of f is not defined in the same universe as f, substitute $f(...)$ by $f_i(...concr_i(...)...)$

  In both cases, $concr_i$ has to be applied to those parameters whose type is defined in the same universe as f.

- **T2**. Substitute $f(...)$ by $g(...)$ using a rewrite rule.
- **T3**. Substitute $f_i(...)$ by the result of its evaluation with the interpreter. If it is necessary, the result will be abstracted.

[2] Which may be equal to its normal form or not.

- **T4**. Substitute *$abstr_i(x)$* by its result (a term).
- **T5**. Substitute *$concr_i(abstr_i(x))$* by *x*.

These transformations will be correct if the implementation is correct. They may be applied to a term of type *T* in different ways that could lead to different final terms denoting the same value. Having the goal of testing in mind, we choose the following strategy:

1) Given the expression *f(...)* of type *T*, built up with objects and with operation symbols of this type and other ones, we first substitute its objects with the following rule: the value of an object declared of type *T* will be of type T_i if the object is inside the implementation *I*; otherwise, the value will be of type *T*.
2) Next, we execute all functions under test, by means of a post-order traversal of the term. Although this step and the preceding one may be performed intertwined, we left them separate to be more clear.
3) Once testing has been carried out, we use rewriting and interpretation as a result of the application of the transformations until we get a term in final form. Concerning rewriting, we have considered bottom-up and top-down strategies; with bottom-up rewriting, each time a rule is applied it should be necessary to restart the process from the bottom of the resulting term; so, we have chosen the top-down approach. On the other hand, interpretation of a function is performed only after evaluating all of its parameters. Usually, a function will manipulate all of its parameters; so, one may think that they would be evaluated more than once during interpretation if they were not evaluated before.

During evaluation, subterms reaching their final form will hold a tag (we call it **final tag**) in all their nodes. This tag will remain on the term when assigned to a variable and, thus, it will be present in the terms obtained by further substitution of the variable. This tag will be useful to cut branches on term traversals. Another tag (called **non-test tag**) will be held by subterms that do not contain non-executed operations under test, for similar reasons.

We present below an algorithm of execution based on this strategy. First, it is necessary to state the condition for a term to be in final form: a term *t* is considered to be in final form if one of the following holds:

a) *t* comes from the abstraction of a term that was in final form. Although *t* may contain executable operations not executed before, this does not happen very often. See section 6 for an explanation.
b) *t* is a value of a predefined type.
c) the three following conditions hold altogether: the root *f* of *t* has UUT(*f*) = *specification* or ME(*f*) = *rewriting*; there is no rewriting rule applicable for *t*; and the children of *t* are in final form.

It is easy to see that a term in final form only may contain non-executed operations under test if it comes from an abstraction.

The algorithm is correct because the final term holds a final tag and final tags are incorporated to a subterm when it meets the condition to be in final form. The termination of the algorithm (provided the confluence and finite termination of the specification and the termination of the imperative code) is assured because recursive calls involve smaller terms that lead to a final case.

In the algorithm, functions in uppercase stand for primitive operations on terms and on an environment that contains all information about universes and functions. Their meaning is often clear; if not, comments have been added.

Evaluate: evaluation of a term *t* in a given environment *e* applying the steps explained above.

```
function evaluate (t: term; e: environment) → t: term
{ t is not empty }
    t := var_subst(t, e)   // Step 1
{ t has no variables }
    t := test_ev(t, e)    // Step 2
{ all functions under test of t have been executed }
    t := final_ev(t, e)    // Step 3
{ t is in final form and it holds the corresponding  final tag }
end function
```

Var_subst: substitution of all the variables of a term *t* with their values in the environment *e*

```
var_subst (t: term; e: environment) → f: term
{ t is not empty }
    f := MKTERM(ROOT(t))   // f is a term containing a node with the root of t
    if IS_VAR?(ROOT(f)) then f := OBTAIN(ROOT(f), e)
            // the whole expression was simply a variable which is now
                    substituted by its value, a term with a final tag
    else for all g in CHILDREN(t) do  // recursive variable substitution
              if IS_VAR?(ROOT(g))
                    then f := ADD_CHILD(f, subst_abstr(f, ROOT(g), e))
                    else f := ADD_CHILD(f, var_subst(g, e))
              endif
         end for all
    endif
{ f is the result of proper substitution of variables of t }
end function
```

Subst_abstr: substitution of a single variable *g* with its value in the environment *e* inside a term *f* which root is an operation symbol

```
subst_abstr(f: term; g: var_symbol; e: environment) → r: term
if DEFINED_IN_THE_SAME_UNIVERSE(ROOT(f), TYPE(g), e) ∧
     DECLARED_IN_THE_IMPLEMENTATION_OF(ROOT(f), g, e)
    then r := ADD_CHILD(MKTERM("abstr"), OBTAIN(g, e))
     // the type of g is defined in the same universe U as ROOT(f), and g
             appears on the universe implementing U. According to the
             former rules, the type of g is T_i and it should be abstracted
    else r:= OBTAIN(g, e)
endif
end function
```

Final_ev: execution of functions not under test

```
final_ev(t: term; e: environment) → t: term
{ all operations under test in f have been executed and f has no variables }
while not HAS_FINAL_TAG(t) do
        if ME(ROOT(t), e) = interpretation then t := interpretation(t, e) // no more loops
        else // ME(ROOT(t), e) = rewriting
                t := rewriting(t, e)    // termination depends on the specification
        endif
        { all operations under test in f have been executed and f has no variables }
end while
{ t is in final form and it holds a final tag }
end function
```

Test_ev: execution of functions under test

```
test_ev(t: term; e: environment) → f: term
{t is not empty and it does not contain variables }
f := MKTERM(ROOT(t))
if HAS_FINAL_TAG(f) then Ø   // the term is in final form
else  // not final tag
        if HAS_NON_TEST_FLAG(f) then Ø // not under test; if executed, at step 3
        else  // the term (or parts of it) may be under test
                for all g in CHILDREN(f) do f := ADD_CHILD(f, test_ev(g, e)) end for
                     // post-order traversal to perform testing
                     // f is a transformation of t and all operations under test in f's children
                        have been executed and all children of f hold a non-test tag or a final tag
                if IS_UNDER_TEST(ROOT(f), e) then
                        if UUT(ROOT(f), e) = implementation then
                                f := interpretation(f, e)
                                                // f is in final form and it holds a final tag
                                                // it will not call test_ev anymore
                        else // UUT(ROOT(f), e) = specification
                                f := rewriting(t, e)
                                        // f does not contain any non-executed
                                        operation under test and it holds a
                                        non-test tag or a final tag
                                        // it may call test_ev after application of
                                        transformation T2.
                        endif
                else f := SET_NON_TEST_FLAG(f)   // the whole term has been tested
                endif
        endif
endif
{ all operations under test in f have been executed and  f has no variables }
{ f holds a non-test tag or a final tag }
end function
```

Interpretation: control procedure to call the interpreter to execute an expression with a given environment. During execution, transformations T1a, T1b, T3 and T5 are applied

```
interpretation(f: term; e: environment) → h: term
{ Operations under test of all proper subterms of f have been executed and f contains no variables.
It holds that UUT(ROOT(f)) = implementation or else that UUT(ROOT(f)) = none and
ME(ROOT(f)) = interpretation }
t := MKTERM(ROOT(f))
for all g in CHILDREN(f) do
        tmp := final_ev(g, e)
        if DEFINED_IN_THE_SAME_UNIVERSE(ROOT(f), TYPE(ROOT(g)), e) then
                // the type of g is some type defined in the same universe as f. f
                  accepts a parameter of some type T but the implementation
                  of f accepts a parameter of type Ti
                if ROOT(tmp) = "abstr" then t := ADD_CHILD(t, CHILD(tmp))
                                                    // "abstr" has an only child
                else // term that is not an abstraction
                        if IMPLEMENTATION_AVAILABLE(tmp, e) then
                                t := ADD_CHILD(t, CHILD(interpretation(tmp, e)))
                        else --> FAIL
                                // Execution can't go further because we are not able
                                  to compute concr for the parameter of f as
                                  required by transformations T1.a and T1.b
                        endif
                endif
        else  t := ADD_CHILD(t, tmp)
        endif
end for
        // now, the real call to the interpreter tool, INTERPRETER
if DEFINED_IN_THE_SAME_UNIVERSE(ROOT(f), TYPE_OF_RESULT(ROOT(f)), e)
        then h := ADD_CHILD(MKTERM("abstr"), INTERPRETER(t, e))
        else h := INTERPRETER(t, e)
endif
{ h is in final form and holds a final tag }
end function
```

Rewriting: control procedure to call the term rewriting system to execute an expression with a given environment. During execution, transformations T2 and T4 are applied

```
rewriting (t: term; e: environment) → f: term
{ Operations under test of all proper subterms of f have been executed and f contains no variables.
It holds that UUT(ROOT(f)) = specification or else that UUT(ROOT(f)) = none and ME(ROOT(f))
= rewriting }
if ¬ MATCHING_WITH_ANY_RULE(t, e) then
        if (for all g in CHILD(t): HAS_FINAL_TAG(g)) then
                f := PUT_FINAL_TAG(t)
        else    f := MKTERM(ROOT(t))
                for all g in CHILDREN(t) do
                        f := ADD_CHILD(f, final_ev(g, e))
                end for
                { all children of f are in final form and they hold a final tag }
                f := rewriting(f, e) // so, this rewriting will not enter this branch again
        endif
else   // a term rewriting step may be applied
        f := test_ev(ONE_REWRITING(t, e), e)
        // ONE_REWRITING possibly has abstracted one or more times. In this case,
        the abstraction holds a final tag. This will prevent redundant
        evaluation of terms during the execution of test_ev.
endif
{ all operations under test in f have been executed and f holds a non-test tag or a final tag }
end function
```

5. Example

Let's suppose we need an abstract data type for sets, and let's suppose that we decide to implement it with another data type for lists, which in turn is implemented with a sequential array. Specifications and implementations for both types are presented below.

```
universe SET defines
    uses INT, BOOL
    type set
    ops
        empty: → set
        put: set int → set
        single: int → set
        in?: set int → bool
    eqns
      // generating ops.: empty, put
        put(put(s, x), x) = put(s, x)
        put(put(s, x), y) = put(put(s, y), x)
        single(i) = put(empty, i)
        in?(empty, i) = false
        in?(put(s, j), i) = equal(i, j) or in?(s, i)
end universe
```

```
universe LIST defines
    uses INT, BOOL
    type list
    ops
        init: → list
        add: list int → list
        reset, advance: list → list
        current: list → int
        end?: list → bool
    eqns
        …
end universe
```

```
universe SET_IMPL implements SET
    uses LIST, INT, BOOL
    repr
        type set is list

        abstr (s: set)
            abstr(init) = empty
            abstr(add(s, v)) = put(abstr(s), v)
            abstr(reset(s)) = abstr(s)
        inv (s: set)
            for all v, w in s: v ≠ w
    function empty → set
    returns init
    function put (s: set; i: int) → s: set
        if ¬ in(s, i) then s := add(s, i)
                     else Ø
        endif
    end function
    function single (i: int) → set
    returns put(empty, i)[4]
    function in? (s: set; i: int) → r: bool
    var r is bool
        r := false; s := reset(s)
        while (¬ r) and (¬ end(s)) do
            if equal(current(s), i) then r := true
            else s := advance(s)
            endif
        end while
    end function
end universe
```

```
universe LIST_IMPL implements LIST
    uses INT, BOOL
    repr
        type list is record
            A: array [1..N] of int; n, cur: int
        abstr (l: list) → t: term
            t := init
            for all i in l.cur..l.n-1 do
                t := add(t, l.A[i])
            end for all
            for all i in 1..l.cur-1 do
                t := add(t, l.A[i])
            end for all
        inv (l: list):
            (l.n ≠ 0 ⇒ l.cur ≤ l.n) ∧ l.n ≤ N

    function init → list
    returns <A, 0, 1>[3]
    function add (l: list; i: int) → l: list
        l.n := l.n+1; l.A[l.n] := i
    end function
    function reset (l: list) → list
    returns <l.A, l.n, 1>
    function current (l: list) → int
    returns l.A[l.cur]
    function end (l: list) → bool
    returns l.cur > l.n
    function advance (l: list) → list
    returns <l.A, l.n, l.cur+1>
end universe
```

Let's suppose that, in a first stage, just a specification for sets exists. Then, any testing process will involve term-rewriting only, as it is the case if we evaluate the expression *in?(single(1), 1)*, which yields *true* after four rewriting steps.

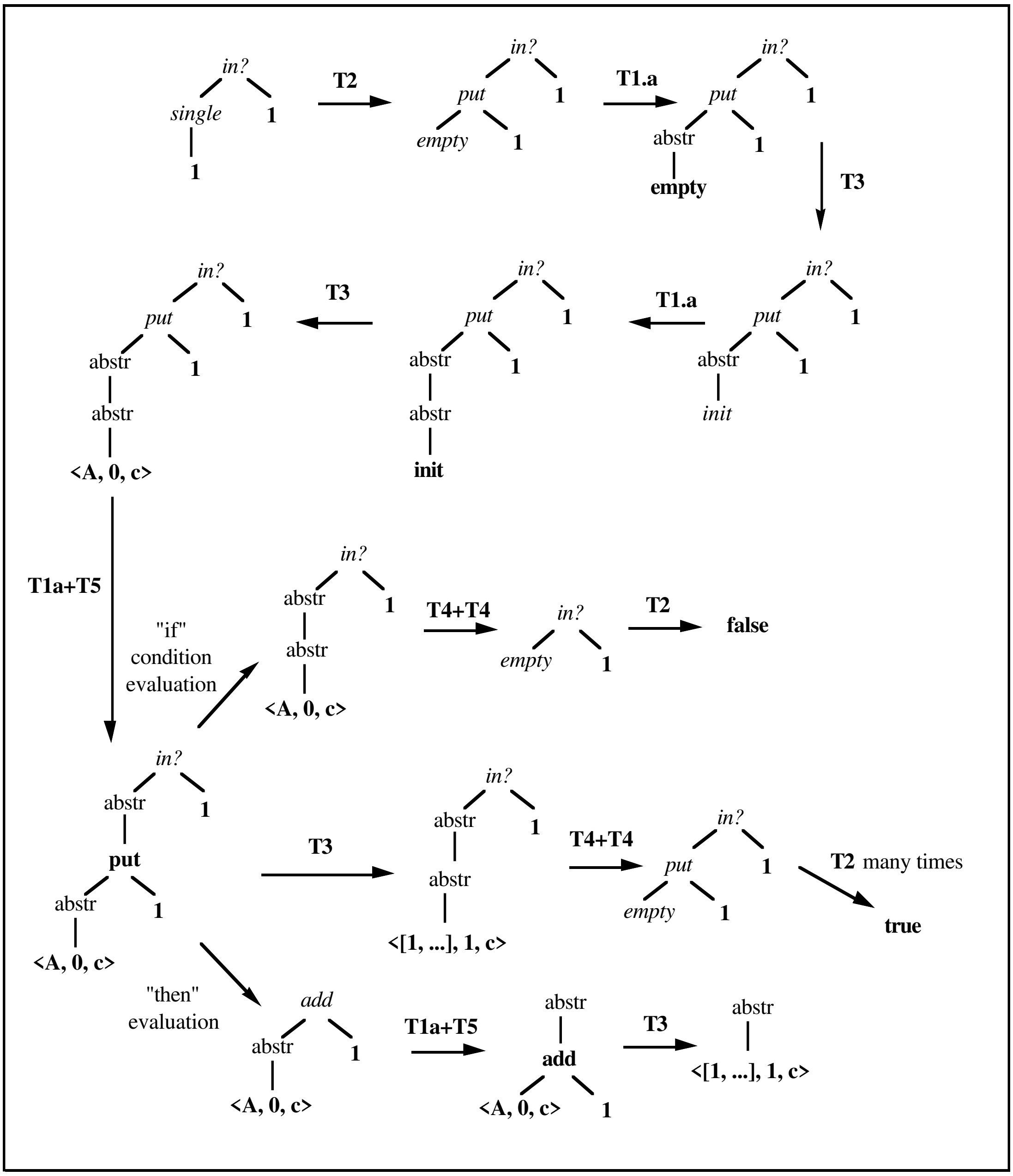
Now, we decide to represent type *set* and that we implement *empty*, *put* and the abstraction function over this representation; also, let's suppose we reuse the specification and the implementation for lists. In order to perform testing, we decide to customise the execution system by assigning UUT(*empty*) = UUT(*put*) = *implementation*. Then, interpretation of both operations is done if possible, even if it is not necessary. In the next figure we show again the execution of *in?(single(1), 1)*, which yields the same result but with a different execution evolution. Transition from an expression to another are done with the rules explained at section 4 under the algorithm of the execution procedure. Boldface means implemented objects and functions, while italics stand for specified ones. The two branches in the third row are auxiliary evaluation processes set by the main one; in the first branch ("if" condition evaluation), an

---

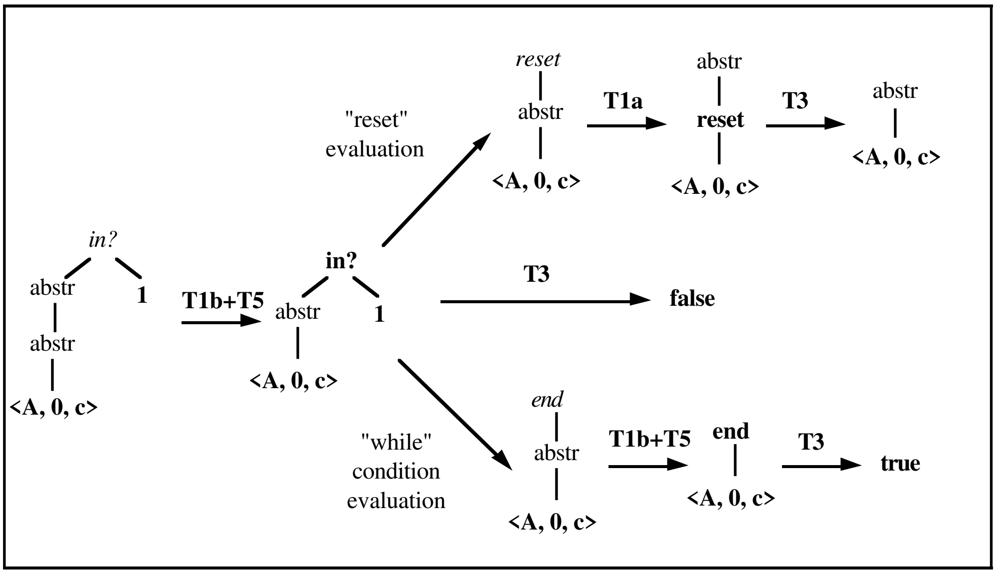
[3] In general, <x, y, z> stands for a record containing three elements of the appropriate types. <A, 0, 1> stands for a record containing any array, 0 and 1.

[4] As a remark, note that this implementation of *single* equals its specification; so, in our execution context, it could be avoided.

abstraction appears because the variable *s* declared in *put* is a parameter of *in?* of a type defined in the the same universe as *in?*. We remark that the second expression reappears after a lot of work is done, because of the testing option.



Next, we implement *in?*, and we set UUT(*in?*) = *implementation*. We show below how the "if" evaluation started at the third row above is now performed, using the implementation for *in?*. Again, two auxiliary evaluation processes are set.



Once the testing of sets finishes, we may customise again the execution system. If we decide to set ME to *rewriting*, the execution of *in?(single(1), 1)* is done again with term-rewriting only, with the same sequence of steps we use before implementing the type. Otherwise, interpretation is chosen as execution strategy for sets.

6. Avoiding undesirable results

The algorithm presented in section 4 may terminate with two annoying results:

a) Failure. It occurs when the execution algorithm decides to evaluate a subterm *f(...)* interpreting the implementation of f, f_i, and there is a formal parameter of type T_i (the implementation of a type T) for which the actual parameter is of type T and there is no way of computing the concretion[5]. This may happen when we provide the implementation of f before implementing the functions that may appear in the result of a rewriting process (the term that we need to concrete). We can avoid this situation if we write down specifications which yield normal forms composed by generating operations and if we start type implementations coding these operations (all at once) [BBF88]. This seems not to be a strong restriction, and the resulting prototyping strategy is still flexible enough (see [FB93] for an example).

b) The result of the evaluation contains operations that could be executed but that they actually have not been executed. This situation arises when the abstraction function yields a term which may contain operations that have not been executed in the generation of the abstracted value. If we follow the development strategy stated in a), we may design the abstraction function such that it returns terms composed only by generating operations; in fact, this situation arises naturally in the general case. Seldom, it could still be possible for the final result to contain some non-executed generating operations; if we want the scheme to work out even in this case, it suffices with having the generating operations not under test before implementing the other ones.

---

[5] In fact, if f_i does not manipulate the representation of the parameter, concretion is not necessary to proceed, but the algorithm presented here does not handle this possibility.

7. Conclusions and future work

A framework for software development by means of prototyping has been presented. We use an abstract data type oriented language that combines equations and imperative programs, and thus software refinement consists of gradually transforming the specified types into implemented ones, being able to prototype the application at any stage of development, provided that specifications are operational and that an abstraction function and a representation invariant do exist for every implementation.

Being evaluation of expressions the main issue of execution, we have identified many rules to transform an expression assuring that its value is preserved; the crucial fact is how specified and implemented functions and values coexist. Then, we have formulated an execution procedure which gives preference to testing over efficiency and which involves a term rewriting system to evaluate specified functions and an interpreter for imperative ones.

Other projects do exist that support execution of equations mixed with imperative programs. The closest to ours is the execution system of Asspegique [CK90], which combines PLUSS specifications with Ada implementations. There is a basic difference, however: mixed execution relies on the compilation of equations into Ada and in the generation of packages for the involved modules. We think that, in the testing framework, compilation prevents the easy interaction between the user and the execution system, as well as quick and interactive modifications of the rewrite rules and customisation of the environment. This compilation scheme has been adopted by many other teams [DDL88, Rad90].

The idea of mixed evaluation may also be found in other contexts. For instance, [Wal90] uses it to improve execution speed of term rewriting systems, considering basic data types (integers, sets, etc.) as implemented, although using them with a set of virtual equations. Other approaches combine imperative and logic programs, or they enrich imperative programs with Horn clauses [DDL88, Rad90, TC88].

About the future work, we want to allow mixed execution even if more than one implementation for the same type is used in different contexts of the same program. This issue has been addressed in [Fra94] and the integration into the current work seems to be natural: no new constructions are required over the language (the abstraction function is enough) and initial semantics still provides a valid formal framework. On the other hand, efficiency of computations should be improved. Mainly, we are studying the interest of sharing of subterms and the advantages of maintaining a twofold value in objects (the result of an abstraction and the value before abstraction). We are also considering the possibility of a lazy implementation of the final evaluation, which would be possible due to the sharing of subterms.

References

[BBF88] P. Botella, X. Burgués, X. Franch. "Combining the imperative and the equational programming paradigms in the Excalibur system". In *Proceedings of the 1st International Workshop on Software Engineering and its Applications*, Toulouse (France), 1988.

[CK90] C. Choppy, S. Kaplan. "Mixing abstract and concrete modules: specification, development and prototyping". In *Proceedings of 12th ICSE*, Nice (France), 1990.

[DDL88] L. Dewez, J.-M. Douin, P. Lucas. "The MODULOG Modula workstation". In *Proceedings of the 1st International Workshop on Software Engineering and its Applications*, Toulouse (France), 1988.

[EKMP82] H. Ehrig, H.-J. Kreowski, B. Mahr, P. Padawitz. "Algebraic implementation of abstract data types". Theoretical Computer Science 20, 1982.

[EM85] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*. EATCS Monographs on Computer Science, Vol. 6, 1985.

[FB93] X. Franch, X. Burgués. "A case study on prototyping with specifications". In *Proceedings of ERCIM Workshop on Development and Transformation of Programs*, Nancy (France), 1993.

[FGMJ85] K. Futatsugi, J. Goguen, J.-P. Jouannaud, J. Meseguer. "Principles of OBJ2". In *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, New Orleans (USA), 1985.

[Fra94] X. Franch. "Combining Different Implementations of Types in a Program". In *Proceedings Joint of Modular Languages Conference*, Ulm (Germany), 1994.

[GTW78] J.A. Goguen, J.W. Thatcher, E.G. Wagner. "An initial algebra approach to the specification, correctness and implementation of abstract data types". A *Current Trends in Programming Methodology*, Vol. IV, Prentice-Hall, 1978.

[Hoa72] C.A.R. Hoare. "Proof of Correctness of Data Representations". In *Programming Methodology*, Springer-Verlag, 1972.

[LH85] D.C. Luckham, F.W. von Henke. "An overview of Anna, a specification language for Ada". IEEE Software, March 1985.

[Rad90] A. Radensky. "Towards integration of the imperative and logic programming paradigms: Horn-Clause programming in the Pascal environment". ACM SIGPLAN Notices, 25(2), 1990.

[TC88] R.B. Terwilliger, R.H. Campbell. "An early report on ENCOMPASS". In *Proceedings of 10th ICSE*, Singapur, IEEE Computer Science Press, 1988.

[Wal90] H.R. Walters. "Hybrid implementations of algebraic specifications". In *Proceedings of 2nd ICALP*, Nancy (France), 1990.