FUNCTIONAL AND STRUCTURAL RECURSION IN SPREADSHEET LANGUAGES

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
The Department of Computer Science

by

Jérémie Allard

B.S., Université d'Orléans, France, 2001

August 2002

ACKNOWLEDGMENT

During the preparation of this thesis, I was honored to work with Dr. Markus Montigel and Damon Hanchey. I thank them for their cooperation, help and comments.

I also thank Dr. Golden G. Richard III and Dr. Eduardo Kortright for the time they devoted to me and for accepting to be on my committee.

I would also like to thank my friends here at UNO: Cyrille, Fred, Capu, Dennis, Lisa, Paul, Minoo, and the others…

TABLE OF CONTENTS

TABLE OF FIGURES

ABSTRACT

Without any programming knowledge, many users are able to use spreadsheet languages, which combine a simple cellular based model with a visual interface. However these languages suffer from several limitations: lack of loop or recursion support, limited structure (two dimensional matrix), no debugging support. This thesis presents several solutions to overcome these limitations. Wizcell, a spreadsheet language currently in development, is used for the concrete implementation.

While the focus of the thesis in on functional and structural recursion, several other crucial concepts are considered first. To support memory constructs, loops and time-based computations, a time model is added, and circular references are introduced. Array support adds the possibility of manipulating data sequences.

Recursion provides a facility to include a module inside itself. This very simple concept leads to functional recursion, in which the module uses the recursively-referenced module to solve a part of the problem. Recursion also supports the definition of structured recursive data structures (for example lists and trees).

Finally, the integration of these concepts enables Wizcell users to visually create programs in a simple manner. To illustrate correctness proofs of programs the example of a quick-sort program is presented.

INTRODUCTION

The family of programming languages is quite rich. However, only few major types of languages are extensively used: mainly object-oriented languages (C++, JAVA), older structured imperative languages (C, FORTRAN), and some declarative languages on very specialized field (artificial intelligence, mathematical programming). Spreadsheet languages form an intriguing family. While they are not generally accepted as a general programming language, they are the most used paradigm if we refer to the number of "programmers". This is the basis of the research project on Wizcell, a programming language derived from spreadsheet languages, adding several important concepts: modularity, time, iteration, recursion, and data structures.

In this thesis I present my work on several additions to the Wizcell language:

- Time concept: support for real-time delays and iterative computations
- Functional recursion: support for recursive computations, using a concept of an module containing itself
- Array support: possibility of using multidimensional arrays
- Structural recursion: addition of the concept of a data type, which is an abstract module only containing input cells, and reuse of the functional recursion mechanism to support recursive data structures (structural recursion).

The presentation of these concepts will concentrate on their application to the Wizcell language, but the methods developed are usable in other spreadsheet languages as well.

The first four chapters present the main architecture of the Wizcell language, including the important modularity concept. Then in chapters 5 to 9 each of the above concepts is discussed and compared to other possible solutions, some examples are shown and the current implementation is presented. Chapter 10 presents a correctness proof of a quick-sort Wizcell program.

CHAPTER 1 : WIZCELL MODEL

Wizcell can be defined as a visual spreadsheet-based language. It uses the same programming paradigm as spreadsheet languages (declarative, cell based), while the program is designed and displayed visually.

The spreadsheet paradigm, as defined by Burnett [1], states that computations are defined by cells and their formulas. This notion is based on Kay's value rule [2], which expresses that a cell's value is exclusively defined by a formula.

A Wizcell program is a set of named cells organized in a hierarchical structure. The program defines the *model* of each cell, e.g. its name, attributes, and possibly its formula. The most important types of cells are *CellGroup*, *FormulaCell*, and *InputCell*. A *CellGroup* is a cell containing other cells. A *FormulaCell* is a cell whose value is defined by a formula. An *InputCell* is a cell whose value is externally defined (e.g. it doesn't have any formula). An important attribute of a cell is the *OutputCell* attribute. A cell can be externally accessed if and only if it is an *OutputCell* (e.g. it has the *OutputCell* attribute). The set of *InputCell* and *OutputCell* define the external interface of the program (see Chapter 2 for more details).

The value of a cell can be of several kinds: integer, real, boolean, string, or array. The value's type is not specified in the model of the cell.

The state of the program is the set of the states of each cell, which include the current value and possibly other information (depending on the update algorithm used). The Wizcell Virtual Machine computes the transitions between each state of the program, observing the language rules (Kay's value rule and other rules presented later). Depending on the update model used, these transitions can have different properties. In this work, I will only consider a model where all cells are updated simultaneously. This means that the values of the cells at step I will be computed by using the values of the cells at the previous step I-1. Other possibilities include sequentially recalculating each cell, but this requires defining an order of the cells, which doesn't fit well to the visual representation (there is no order as in a text-based source code). The formulas themselves use a quite standard syntax, with the usual arithmetic, comparison, and conditional operators (see Chapter 4 for more details). They must follow 2 important properties:

- A formula has no side effect. This means that its only result is the computed value.
- A formula exclusively depends on the current value of the cells (e.g. given the values of all the cells of the program, the formula will always result in the same value).

The main consequence is that it doesn't matter how often the virtual machine uses the formula, as long as it is recomputed often enough to update the value of the cell when it is needed. This allow us to use a simple mathematical model where each cell is updated at each step, and a more optimized practical model where the value of a cell is only recomputed when needed. As formulas have no side effect, the two models are

strictly equivalent in terms of correctness; but the second one is much more efficient in terms of time complexity. To implement the second model we define 2 properties of each cell:

- The set of referenced cells, containing all the cells used to compute the value of the cell.
- The set of dependant cells, containing all the cells using the value of the current cell (this is the inverse relation)

The optimization is to recalculate a cell's value if and only if one or several referenced cells' value changed in the previous step.

Another consequence of the second property is that no history needs to be stored, only the current value of each cell.

This leads to the following update algorithm:

```
OutdatedCellList = all cells
do
     NextOutdatedCellList = empty
     // first calculate all the new values
     for all cell C in OutdatedCellList do
          C.NextValue = CalcValue(C)
          if (C.NextValue != C.Value)
               NextOutdatedCellList += dependant(C)
          end if
     end for
     // then activate the new values
```

```
            for all cell C in OutdatedCellList do
                 C.Value = C.NextValue
            end for
            OutdatedCellList = NextOutdatedCellList
      while (OutdatedCellList != empty)
```

The last but not least important aspect of a Wizcell program is the visual representation. Each cell can be graphically represented, allowing the user to see (and construct) the model of program, and the current value of each cell (and change *InputCell*s' value).

The extension of this model to support the concepts of modularity, time, iteration, recursion, array and data structures will occupy the remaining chapters.

CHAPTER 2 : PROGRAM ARCHITECTURE: MODULARITY

In order to be useful, a programming language must be able to interact with other languages, sometimes using a totally different paradigm (like the omnipresent procedural languages). This is one of the reasons why Wizcell is very modular. A program is defined by a set of modules. Theses modules can use different programming languages. Currently cellular-based and Java based modules are supported but several other models could be added (e.g. state-graph, CORBA). A module can contain other modules. In fact, the program is a module only containing other modules. In order to achieve this, a clean interface must be defined so that these modules can exchange information. This chapter defines this interface and how it works for cellular-based modules.

Note that in the other chapters only the cellular-based module paradigm is described. However, when a *Module* is used, its internal paradigm is unknown, and only the programming interfaces defined here are available to interact with it.

2.1 Definition of a module

A module is an opaque entity, which communicates with other modules using a set of input cells and output cells. The module structure and behavior are defined by its model. During the execution of the program, a module can have several instances, each

of which has a specific state, and possibly a different view. Those components are described in the following sections.

A module is used as follow. First the model is created (loaded from a file), then it is initialized (the compilation phase). From this point several instances can be created and used (the execution phase).

2.2 Model

The model defines the structure of the module, which – from an outside point of view – is the list of names of the "public" cells. A cell is "public" if it is either an *InputCell* or an *OutputCell*. The model also specifies the "internal" (e.g. non public) cells and the behavior of the module (e.g. the way to recompute the module's state, which is defined by the formulas in a cellular-based module).

The cells are accessed by their names. Since searching for a cell given its name is not a very efficient operation, cells are stored in an array and referred by their index in this array. In some cases, one index is not sufficient. Therefore a list of indices is returned instead. This list indicates the path from the initial group to the final cell by specifying the indices of each cell on the path. This is used, for example, to implement arrays and sub-modules. In order to access a cell in a module, we must translates its name into an indices list IDS, then ask for the cell with index IDS[0], then its sub-cell IDS[1], …, up to the final cell. We can then query the cell about its type and attributes (most importantly if it is an input/output cell), or store the indices list for future references. So the name lookup operation is performed only during the "compilation" phase of the module. As the model is shared by all the instances of the module, it doesn't store the value itself of the cells. This is important for the efficiency in the case where

there are many instances of the same module, as it is often the case when recursion is used.

The following operations can be performed on a module model:

- Load the model from a file/url (note that this operation is not implemented by the module but by a *ModuleFactory*, which is a component of the Wizcell Virtual Machine).
- *init*: This method is called by the *ModuleFactory* to initialize the newly loaded module.
- *getName*: Get the name of the model.
- *resolvePath*: Get the indices list of a cell given its name.
- *getSubcellModel*: Get the model of a sub-cell given its index.
- *instantiate*: Create a new instance of the module. This creates a new state structure. This is the only operation used during the execution phase so it must be as efficient as possible (e.g. not copy the whole model for each instance).

When a module contains another module, it must first load the sub-module's model using the *ModuleFactory*. Then it should search for the public cells it uses, check the permissions, and store the obtained index.

The following operation can be done on a cell model (obtained by the *getSubcellModel* operation on the module):

- *getName*: Get the name of the cell.
- *getSubcellModel*: Get the model of a sub-cell given its index.

- *resolvePath*: Get the indices list of a cell given its name (relative to this cell).

2.3 Instance (Module State)

When a module is instantiated, the state of the new instance is created and used by the Wizcell VM to manage the module. The instantiated modules form a tree, with the program as the root module and the sub-modules as leafs. Once the instance is created, public cells can be accessed given their indices list. The value of the *InputCell*s can be set, and the value of the *OutputCell*s can be queried.

When a module wants to set the value of a sub-module's *InputCell,* it must use the *bind* operation of the cell, which means that the *InputCell* is now programmatically controlled by this module and is not allowed to be externally modified anymore.

If an external component needs to change the value of an *InputCell* (e.g. the GUI, or an external software or hardware), it should use *isBound* before calling *setValue* to ensure that the operation is allowed. Similarly the container module must call *bind* to prevent any external component from modifying the cell.

In order to support the optimized update mechanism, the value of an *OutputCell* is never queried directly, but the dependant cell must first register a *CellLink,* indicating to the *OutputCell* which cell to notify when its value changes. This registration procedure is also used for lazy instantiation (see Chapter 8). A *CellLink* is a small data structure containing a reference to the referenced cell and the dependant cell, as well as fast facilities to be added and removed dynamically at runtime (two *CellLink* references forming a double linked list, see Chapter 5 for a justification of this approach).

Also, a module state can have an associated graphical representation (see next section).

A module state supports the following operations:

- *init*: initialize the newly created instance (instantiating the sub-modules for example).
- *getContainer*: get the container module (or null if this is the root module).
- *getSubcell*: get a sub-cell given its index.
- *getModel*: get the model associated with this module instance.

A cell state supports the following operations:

- *addLink*: register a link to this cell (specifying the dependant cell).
- *getValue*: get the current value.
- *removeLink*: unregister a link.
- *notifyChange*: notify of a change from a referenced cell (e.g. a cell with an active link from this cell).
- *setView*: set the view associated with this instance (or null if none).
- *getView*: get the graphical representation associated with this instance.

The operation *addLink* must always be called before *getValue*.

Note that one can only add a link to and get a value from an *OutputCell*.

An *InputCell* state additionally supports the following operations:

- *bind*: bind the cell (e.g. no more external value modifications allowed)
- *setValue*: set the value of the cell
- *unbind*: unbind the cell (e.g. the cell's value is not programmatically controlled anymore)

- *isBound*: test if the cell is bound or not

2.4 View

To each module instance one or more views can be associated. A view is a graphical window displaying the value of the cells and enabling the user to type the value of the unbound *InputCell*s. A module must notify its view when a cell's value is changed.

A module view supports the following operations:

- Load the view from a file/url (note that this operation is not implemented by the module but by a *ViewFactory*, which is a component of the Wizcell Virtual Machine).
- *init*: initialize the view after it has been associated with a module (creating the graphical components.
- *setShow*: temporarily hide or show the view
- *close*: permanently close the view.

2.5 Module References

Using a module inside another doesn't only require specifying the path/url of the file where the module's model is stored; but the appropriate references of public cells must be established with the containing module so that the modules can be linked together.

For each module we can define a public interface, which is a version of the module containing only the declaration of the public cells. This cell group should be cloned wherever this module must be instantiated. This enables the containing module to

optionally bind the *InputCell*s to the needed formulas, and use the results computed in the *OutputCell*s.

This referencing mechanism offers several advantages:

The public interface is clearly defined, and it is the only information needed by a client.

The interface can be placed in a palette containing other modules, thereby building a set of useful readily usable components by only dragging the interface anywhere it is useful.

Also, the implementation of the module can be modified or extended, as long as the public interface still includes the cells contained in the previous version. On the other hand, if the implementation involves radical changes to the public interface, the error will be detected and reported cleanly.

CHAPTER 3 : CELLULAR MODEL

Wizcell modules considered in this work mainly use the cellular model. This chapter presents the practical aspect of this model, from the point of view of the user/module developer.

3.1 Component

We use MacOS X Cocoa tool as a prototype environment to quickly develop Wizcell modules. The palette of components is presented in Figure 1.

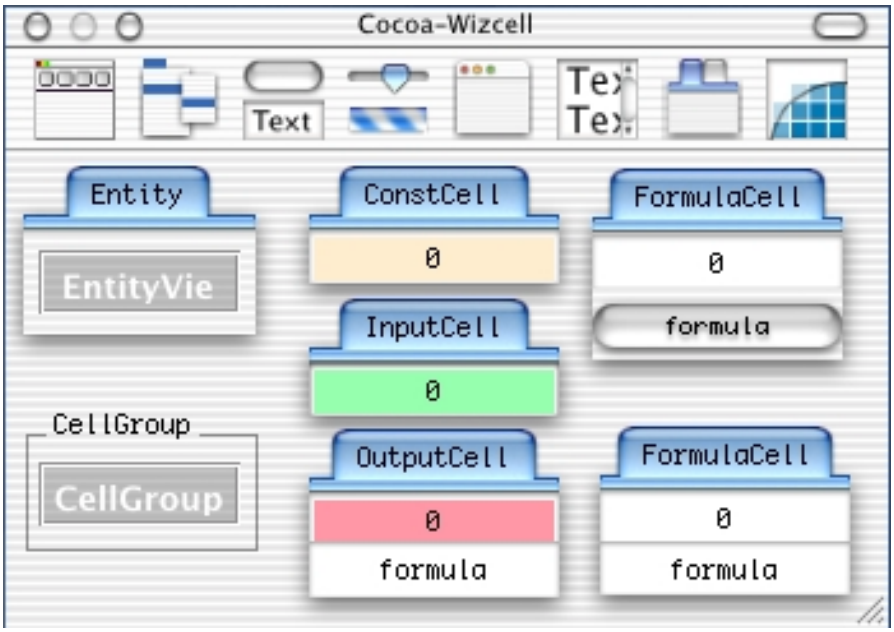

Figure 1: Graphical representation of some of Wizcell's cell types.

The basic work surface is a graphical window representing a Module.

The cell types available are:

- *InputCell*: get values from users or from external sources (other modules' *OutputCell*s). An initial value is specified. It is represented in green.
- *ConstantCell*: store a constant value.
- *FormulaCell*: the most used cell type. It contains an initial value and a formula.
- *CellGroup*: basic container. Group cells together in a hierarchical manner.
- *Entity*: a *CellGroup* meant to represent, in a reusable way, some real-world entity.
- *Module*: a independent *CellGroup*. It has its own graphical window. It is generally used to compose the final program, but can also be nested using a *ModuleRef* (e.g. a module can have sub-modules).
- *Program*: a "top-level" module. It is generally meant to be used as-is by the end-user, but can also be linked with other modules (e.g. for testing purposes).
- *ModuleRef*: A special group referring to another module and containing a copy of the "public" cells of the module that can be modified (for *InputCell*s), or referenced (for *OutputCell*s). This is the mechanism to link modules together. The title of the group consists of the name itself followed by either *ref* or *lazyref*, followed by the path to the module's model file. A *lazyref ModuleRef* only instantiates the sub-module when the first reference to itself or to one of its sub-cells is used (see Chapter 8).

*InputCell*s, *FormulaCell*s, *ConstantCell*s, *CellGroup*s, *Entities* and *ModuleRefs* can be placed in any container.

*Module*s and *Program*s are top-level groups. Their model can't be placed in another container (only a *ModuleRef* and not a Module can). A *Module* instance can be in a container, a *Program* can't.

A Wizcell program is either a *Program* or a *Module*.

*FormulaCell*s and *InputCell*s have a boolean property "Output" indicating whether the cell's value can be queried by another *Module*. A *FormulaCell* where this property is true is sometimes referred as an *OutputCell*.

3.2 Formulas and Cell References

*FormulaCell*s contain a formula defining its value.

These formulas manipulate values of type number, string, boolean, or complex data structures (which will be discussed later). Constants and cell's values are combined using the usual arithmetic operators. Conditional construct of the following form is available:

(condition1) ? expression1 ; expression2

References to cells are composed by a base specification, indicating the start of the reference. This can be a '$' meaning the base of the module, or a '.' meaning the base of the entity, or a possibly empty list of ':' indication the number of groups to walk up from the container group of the current cell. Then a series of cell names separated by period specify which sub-cell must be used, ending by the actually referenced cell.

These concepts will be detailed and expanded in the following chapters.

CHAPTER 4 : FORMULAS

In cellular-based modules, a cell's value is exclusively defined by its formula. Therefore, the details of the formula language used are quite important. This chapter presents the syntax of the formulas.

4.1 Formula Syntax

As Wizcell is a visual language, most elements stem from graphical objects or XML files, not regular source files. However, formula is a string containing some expression.

We use Right-Recursive formal grammar to define the syntax. This allows to easily implement the parser without needing any specific tool. The syntax for an expression is:

```
Expr        := DelayExpr   |   DelayExpr ? DelayExpr ; Expr
DelayExpr   := LogExpr   |   LogExpr @ Delay
Delay       := ()   |   ( Realtime )   |   ( Realtime , Number )
            |   ( Realtime , Number , Number )
Realtime    := Number   |   Number Ident
LogExpr     := LogFactorExpr
            |   LogFactorExpr '|' LogExpr
            |   LogFactorExpr # LogExpr
```

LogFactorExpr := LogOperand | LogOperand & LogFactorExpr

LogOperand := RelExpr | ! LogOperand

RelExpr := ArithmExpr

| ArithmExpr < ArithmExpr | ArithmExpr > ArithmExpr

| ArithmExpr = ArithmExpr | ArithmExpr != ArithmExpr

ArithmExpr := FactorExpr

| FactorExpr + ArithmExpr | FactorExpr – ArithmExpr

FactorExpr := ExpExpr | ExpExpr * FactorExpr

| ExpExpr / FactorExpr | ExpExpr % FactorExpr

ExpExpr := Operand | Operand ^ ExpExpr

In this formulation several elements are used but not defined:

Number: a string beginning with a digit and representing a valid real value (*0-9+ [. 0-9+][E [+|-] 0-9+]*)

Ident: an identifier beginning with an alphabetic character or '_' and continuing up to the first non-alphanumeric character (*a-zA-Z_ (a-zA-Z0-9_)**).

Operand: defined below.

4.2 Operand Syntax

An operand can be of several types: a number, a boolean, a string, or an array.

To support the advanced features (arrays, recursion, data structure) the operand syntax is quite complicated, particularly concerning cell references. See Chapter 7 for more details on cell references.

A *CellGroup* can have a value, and a reference to a cell can contain array indices, and maybe values, for example we can allow formulas like *(array1+array2)[i].cell1*

This means that the '.' is now an operator, taking a value and one of its sub-cells name and resulting into the value of this cell. Also the array operator [] is added, taking a group and an index and returning the cell with the specified index. Ranges of indices are also allowed.

An array can be specified as a list of expressions enclosed by '{' and '}'.

This syntax also supports the specification of functions. A function has a name and takes a list of arguments enclosed by parenthesis.

The operand syntax is:

```
Operand     := true   |   false
             |   "string"   |   Number
             |   + Operand   |   - Operand
             |   Array
             |   Function
             |   CellPath
Array       := { }   |   { ExprList }   |   { ExprList } Subcell
Function    := Ident ( )   |   Ident ( ExprList )
ExprList    := Expr   |   Expr , ExprList
CellPath    := BaseCell   |   BaseCell Subcell
             |   ( Expr )   |   ( Expr ) Subcell
BaseCell    := Ident
             |   $ BaseCell   |   : BaseCell
Subcell     := . Ident   |   . Ident Subcell
             |   [ Expr ]   |   [ Expr ] Subcell
```

| [Expr .. Expr] | [Expr .. Expr] Subcell

| [Expr ..] | [Expr ..] Subcell

| [.. Expr] | [.. Expr] Subcell

| . length

CHAPTER 5 : TIME AND ITERATION

5.1 Cyclic Reference

A consequence of the Wizcell's update mechanism is the possibility of using cyclic references, e.g., a cell references its own value either directly or through other cells. In commercial spreadsheets, this mechanism is disallowed, preventing the programmer from implementing calculations involving loops and memory-like structures.

A simple example illustrates this concept:

A cell can memorize its previous value by referencing itself in its formula. This can be used to create memory-like structures, as shown in Figure 2. When *Memorize* cell is set to *true*, the value of *IN* will be stored in *Memory*, and will persists after *Memorize* changes to *false*.

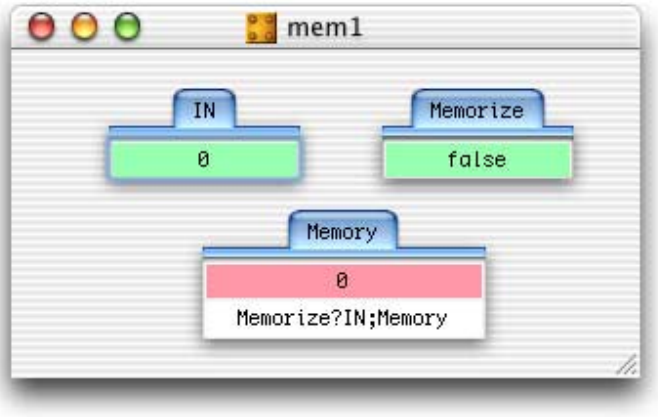

Figure 2: Implementing memory with a self-reference

Unfortunately, this is not enough to support the iteration concept. Even if cells playing the role of loop variables can be created, it is not trivial to coordinate the loop increment with all calculations of the current iteration inside the loop. Interestingly, it turns out that iteration has a very close relationship to the employed time model. In fact, the loop coordination will be accomplished by delaying the evaluation of certain cells by infinitesimally small amounts of time. The next section will present this idea in conjunction with a concept for real-time support.

5.2 Time Model

The time model serves to include a real-time clock and also another clock, the VM time, for iteration support. Further, we need to add some kind of delay in the cells' evaluation. Several solutions are possible:

- *Delayed output*: the actual output of the cell's value of the cell is delayed, meaning that a queue is contained in the cell which holds the previous output values for a certain amount of time.
- *Delayed input*: in a formula, a dependency on the value of another cell can include a certain delay, meaning that some past value of the referenced cell is used.
- *Delayed update*: a delayed cell changes its value only if the formula's result has been stable during the specified time delay, so that the cell is guaranteed to be stable during this specified period of time.

Each solution seems appropriate to some applications. The first two solutions seem useful for simulation purpose, when latency must be modeled. However, the last

solution is more intuitive for programming mechanisms such as loops and sequential calculations, where some cells must be stable while others are updated. As Wizcell is more targeted to general calculations, it uses the *delayed update* solution.

Another important advantage of the delayed update solution is its overhead. Because it doesn't require the maintenance of several queues, it involves much less overhead in the implementation than the other methods. Only a modification of the propagation mechanism for updates among cells is required.

The resulting values of a Wizcell program that are visible to its environment should always be correct. In other words, when a loop is used to perform a calculation, the environment should only see the result once the loop has stabilized, and not all the intermediate (invalid) values. This leads to the requirement that the real time (environment) clock of the virtual machine is incremented only when the non-delayed cells are stable. A real-time delayed cell is not updated until the current calculation is complete, e.g., all other non-delayed cells are stable. This ensures that a real time delayed cell will not change its value in the middle of a calculation, leading to indeterministic results.

In the following approach of integrating iteration with real-time support, two time scales will be distinguished:

The previously mentioned restrictions ensure that the VM time granularity is always infinitely small compared to the one of the real time, e.g. any real time step will always be larger than any number of VM time step. This concept is very similar to the hyper-real based time introduced by [3]. This idea will be extended in the following section to include the proposed iteration concept.

5.3 Iteration Concept

Many algorithms or mathematical functions are based on iterative loops. This construct is very important for the usability of a language. In Wizcell, a loop can be implemented using a circular reference.

The iteration construct must ensure that:

- Each iteration leads to a stable result. In other words, the loop variable is incremented only if the calculation for the current iteration is finished.
- Nested loops must be supported.
- During each iteration step, a sequence of calculations can be enforced in a particular order.

The last point is important for complex calculations.

The time concept introduced in the previous section will be extended to suit these needs. This concept is based on a hierarchical set of time scales, in which the VM iteration update scale is the infinitely smallest scale, and the real time clock is the largest scale. This scheme is extended by adding an infinite number of intermediate time scales h(i) defined by:

$$h(0) = \text{VM time}$$

$$h(\infty) = \text{real time}$$

$$\forall i \ \forall a, b \ \ a * h(i) < b * h(i+1)$$

where h(i) are infinitesimally small hyper-real numbers, i is a natural number including 0, and a, b are positive real numbers. The current time value can be represented by a list of numbers $\{t_0, t_1, \ldots, rt\}$, where t_i is the clock value of time scale h(i). In the implementation the hyper-time values are stored only up to the last non-zero value.

A time value v_1 is less is defined as before another time value v_2 if and only if $v_1.rt < v_2.rt$ or ($v_1.rt = v_2.rt$ and $\exists\, i \mid$ s.t. $v_1.t_i < v_2.t_i$ and $\forall\, j > i$ $v_1.t_j = v_2.t_j$).

The use of this representation of time is to delay the update of cells that need to wait until the end of another calculation before proceeding (such as a loop variable).

Using this time concept, a delay is defined as a triple (R,H,S) where:

- R = real time delay (delay on the h(∞) time scale)
- H = hierarchical time scale of this cell (meaning that the cell is updated only on h(H) time scale increments)
- S = sequence number of this cell

This delay is integrated into Wizcell formula syntax as an option at the end of each conditional branches and preceded by an '*@*' sign. Note that absent delays are considered to be (0,0,0).

R is used for real-time related cells (such as controlling a delay before the activation of an alarm).

H is used to control loops.

S means that the cell will be updated when h(H) clock's value equals to S.

Each branch of a conditional expression (*cond*?*br1*;*br2*) can have a different delay, enabling to specify for example a light timer cell, with a different "ON" (short) delay that the (long) "OFF" delay.

When a cell's value is recalculated, the current time *t* and the delay *d* (*d.R*, *d.H*, *d.S*) is used to compute the time *t'* when the new value will be set as the value of the cell using the following algorithm:

```
if d.R > 0
   t’.realtime = t.realtime + d.R
   t’.hypertime = all zeros
   t’.hypertime[d.H] = d.S
else
   t’.realtime = t.realtime
   for all 0 ≤ i < d.D do
      t’.hypertime[i] = 0
   end for
   if d.S == 0
      t’.hypertime[d.H] = t.hypertime[d.H]+1
      for all i > d.H do
         t’.hypertime[i] = t.hypertime[i]
      end for
   else
      t’.hypertime[d.H] = d.S
      for all i > d.H do
         t’.hypertime[i] = t.hypertime[i]
         if t.hypertime[d.H] < d.S
           t’.hypertime[d.H+1] = t.hypertime[d.H+1]+1
         end if
      end for
   end if
```

Each cell's state contains in addition the following information to the current value:

- nextTime: time when the new value of the cell will be activated
- nextValue: new value of the cell
- delay: last computed delay

An infinite value for *nextTime* means that no update is necessary (*currentValue=nextValue*).

When a new value *v* and a delay *d* is calculated for a cell, its state is updated as follows:

```
if v == currentValue
   // no more update necessary
   nextValue = currentValue
   nextTime = infinity
else if v != nextValue or d != delay
   // new update
   nextValue = v
   nextTime = currentTime+delay //using previous
                                //algorithm
// else v!=currentValue and v==nextValue and d==delay
// nothing changes
end if
```

The current time starts with all zeros. The new update algorithm sets the time as the lowest next update time of all cells and only updates cells such that *nextTime* = *currentTime*:

```
nextTime = lowest nextTime of all cells
while nextTime != Infinity and nextTime.realtime <
      RealTimeClock do
     currentTime = nextTime
     // First activate the new values
     OutdatedCellList = empty
     for all cell C where C.nextTime == currentTime do
          C.Value = C.NextValue
          OutdatedCellList += dependant(C)
     end for
     // Then recalculate all the outdated values
     for all cell C in OutdatedCellList do
          newValue = CalcValue(C)
          // update nextValue and nextTime according
          // to the previous algorithm
     end for
     nextTime = lowest nextTime of all cells
end while
```

5.4 Example: Exponential Approximation

To illustrate this concept, the exponential function is approximated using the following formula:

$$\exp(x) = 1 + \sum_{i=1}^{n} \frac{x^i}{i!}$$

In this example, the calculation is simple enough that it could be implemented with one cell, but to better illustrate the time concept it will be decomposed into several steps.

The exponential approximation function is implemented as shown in Figure 3. This module approximates the exp function of *X* using *N* iteration. In this implementation, the loop index cell *I* has a delay of (0,1,1), meaning it is updated on h(1) increments where the second hyper-time clock is equals to 1. The internal loop's calculations are updated sequentially between these increments. Their activation order is controlled by the delay's sequence number S.

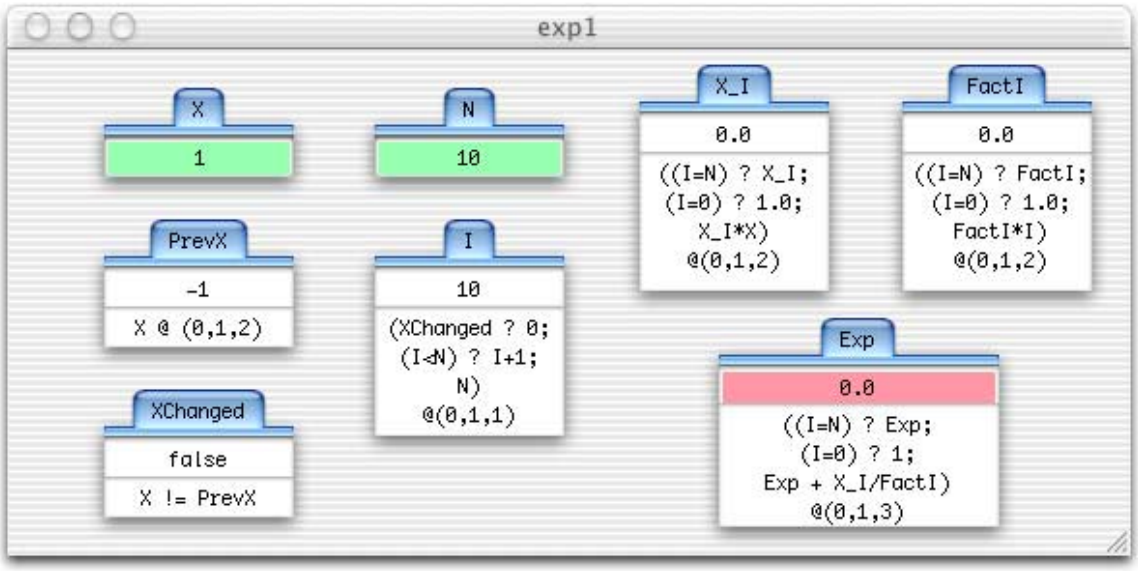

Figure 3: Implementation of the approximation of the exponential function

A loop iteration occurs within each h(2) step, so the exponential value is completely calculated at each h(3) increment. If the input/output values are directly linked with the user interface, which only works on real time steps, then the user will always see coherent results, because h(3) steps are infinitely small from his point of view.

The steps of calculation of exp(1) are detailed in Table 1. The “Time” column shows the first three hyper time counters *hypertime[2].hypertime[1].hypertime[0]*. Time *0.0.0* corresponds to the initial values of the cells, and the following time steps show only the modified values.

Time	X	N	PrevX	XChanged	I	X_I	Fact_I	Exp
0.0.0	1	10	-1	false	10	0.0	0.0	0.0
0.0.1				true				
0.1.0					0			
0.2.0			1			1.0	1.0	
0.2.1				false				
1.1.0					1			
1.3.0								2.0
2.1.0					2			
2.2.0							2.0	
2.3.0								2.5
3.1.0					3			
3.2.0							6.0	
3.3.0								2.66667
4.1.0					4			
4.2.0							24.0	
4.3.0								2.70833
5.1.0					5			
5.2.0							120.0	
5.3.0								2.71667
6.1.0					6			
6.2.0							720.0	
6.3.0								2.71806
7.1.0					7			
7.2.0							5040.0	
7.3.0								2.71825
8.1.0					8			
8.2.0							40320.0	
8.3.0								2.71827
9.1.0					9			
9.2.0							362880.0	
9.3.0								2.71828
10.1.0					10			

Table 1: calculation steps for exp(1).

CHAPTER 6 : ARRAY SUPPORT

Arrays are used quite often in programming languages. In basic spreadsheet languages a 2 dimensional array is the only available structure for the data and the program itself. In this chapter, I will concentrate on one-dimensional arrays, but extensions to more dimensions should be easy (expect for the graphical point of view with more than two dimensions).

6.1 Array specification

In Wizcell an array is a special *CellGroup* with a first cell named "*length*" containing the length of the array, and a set of cells of size *length* accessible by an index from 1 to *length*. Arrays can either be defined cell by cell or with a single formula.

Cell by cell specification includes:

- A length cell specifying the length of the array, containing either an initial constant and possibly a formula.
- A set of cells, also containing an initial constant and possibly a formula. The cell *i* specify the formula for the array's cell *i*, if this cell exist in the array

  (e.g. $1 \leq i \leq length$).
- A last cell specifying the remaining cells of the array if the length of the array is greater than the number of specified cells.

A formula can specify an array using either:

- A direct specification $\{e_1,e_2,\ldots,e_n\}$ of a *n* cells arrays with the indicated values.
- A reference to an existing array, which will copy the value of the array.
- A concatenation of several arrays, using the '+' operator
- The result of a function returning an array. (see the select function is section 6.4

6.2 Accessing an array

An array can be accessed using either:

- *name* = the entire array named *name*
- *name*.*length* = length of array *name*
- *name*[*index*] = cell *index* of array *name*
- *name*[*index1*..*index2*] = range of array *name* from index *index1* to index *index2*. If *index1* is unspecified it equals 1 by default. If *index2* is unspecified it equals *length* by default.
- *(expr)* instead of *name* in all previous cases

Any access to an invalid index will result in an exception.

6.3 Example: Reversing an Array

In the following example, an array A is specified as the input, and an array B is computed containing the values of array A is reverse order. The length of B is the same as A, and the formula of all B's cell is A[length-ID+1]. The resulting module is shown in Figure 4.

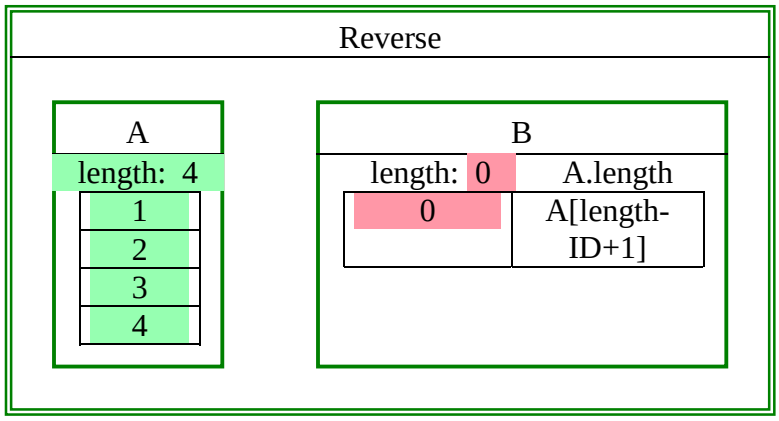

Figure 4: Reversing an array

6.4 Operations on arrays

In addition to the previously described access methods, two basic operations are available on arrays: concatenation and selection.

Using the '+' operator two arrays can be concatenated. The length of the result is the addition of the length of both array, and the content is the cells of the first array followed by the cells of the second array.

Another important operation is the *select* function, which filters the value of an array. This function takes as parameters an array and a conditional expression of a variable *x*. This conditional expression is applied to all cells of the specified array, replacing *x* by the value of the cell, and the cells for which the expression is true are kept in the resulting array.

6.5 Implementation

Arrays are implemented similarly to sub-modules. The cell with ID 0 is the length of the array; other cells have the same ID as their index in the array.

To refer to an array as a whole, a value containing the array of all the cell values is constructed. This operation is only performed if the array is ever referenced. In this case, the array itself depends on the value of each of its cells, so that any cell that depends on this array is notified as soon as one of the array's cells of the array has changed. The same mechanism will be used for other data structures (see Chapter 9).

CHAPTER 7 : CELL REFERENCES

Cell references are a very important element of Wizcell. They are used in most formulas and determine how a cell can access another thereby determining modularity and access rights. As references are used very often, their implementation must be as fast as possible, while still allowing for all the required features, mainly modularity, recursion and arrays.

As explained in Chapter 1, each cell in a module is given an ID, corresponding to its index in the array of cells of the module. This allows fast references to cells using simple array lookup.

After the parsing phase, each reference is constructed by a base group and a list of sub-cell specifications. The base group can be either:

- A specific group: either the cell itself, or one of its containers (parent group, parent module, or parent program).
- An expression: in this case the group is dynamically constructed. Note that the result must be a group.

Each sub-cell specification gives access to one of the current group's cells, given its name or its index (in case of an array). A reference resolution phase takes place at the end of the parsing, whose role is to find the IDs that correspond to the specified names and to set up the static reference lists (see below).

In order to improve the efficiency of this mechanism we can use the fact that the groups inside of a module are static, e.g., cell I of group J is located at index I+J0, where J0 is the index of the first cell of group J. Thus, instead of storing *sub-cell "I"* of *sub-cell "J"* we can directly store *sub-cell "I+J0"*.

I distinguish two different kinds of references: static references and dynamic references. Static references are references that always exist. For example, if a cell A contains the formula "B+C", then the reference to B is static. The other kind is dynamic reference. A reference is dynamic if it can't be statically defined during compilation. This includes:

- references containing an array index;
- references inside a conditional expression (e.g. not used all the time); and
- references to a cell in another module.

This distinction is very important for the optimized update mechanism of Wizcell. When a cell's value is changed, all dependant cells must be notified. In the static case, the list of dependant cells is computed during the compilation phase, but in the dynamic case this list cannot be completely precomputed.

Functional recursion also depends on this reference mechanism. It is based on lazy instantiation, meaning that a recursive module is not instantiated until it is really needed, and the number of dependant cells of the module's output cells determines this (see Chapter 8).

To justify this approach, the example shown in the previous chapter will be used again (see Figure 4): In a simple module we have two arrays A and B each of size N

(where N is possibly dynamic). A is an input array. B is reversed copy of A, meaning that the formula for cell B[i] is A[N+1-i].

Let us assume that we store only one cell description for each array. Thus, each cell of the same array has the same formula and – more important – the same list of dependencies and dependant cells. As a consequence, all cells of array B depend statically on all cells of array A. If dynamic references are neglected all cells of array B will be updated each time one cell value of array A is modified, resulting in possibly $O(N^2)$ calculations each time the whole array A is modified. This is very inefficient and shows the need to manage these dependencies in a more intelligent way.

One quite simple solution is to specify the index of the modified cell in the notification call, so that the notified cell can easily compare it with the currently used index and ignore the message if it is different. This will still require $O(N^2)$ notifications but only O(N) calculations.

A better solution is to notify only the cells that really use the modified cell. This requires storing the list of dependant cells on each instance of the array's cell.

The data structure used to store this dynamic dependency list must implement the following operations as fast as possible:

- Moving a dependency from a list to another
- Iterating through the list

A double-linked list is able to support these operations with O(1) complexity to move a dependency and O(n) complexity for iterating through the list, where n is the number of elements in the list. Another possibility would be a dynamic array, but this solution seems less efficient.

In the implementation, a reference consists of two lists of cell IDs, specifying the path from one cell to another. These arrays contain the path to each cell starting from the first common cell group. This means that to reach the second cell from the first one, we must move N1 levels up, where N1 is the length of the first array, and then move to the sub-cells whose ID is stored in the second array. The reverse path is specified by swapping the arrays. The reverse path is needed for finding the cell to be notified when the destination cell is modified. Note that a negative ID for an array means all the cells from the opposite index to the length of the array. It is used because the cells at the end of an array can share the same formula, and so the same cell reference (this is only the case when the reference is static).

We could argue that the references across modules could be statically determined during the compilation phase. This approach would definitely be more efficient but it would introduce a dependency between the container module and the contained module. This in turn would require managing different versions of the same module depending on the module containing it and would make the initialization much more complicated, especially in the presence of recursion. The overhead of having dynamic references in this case should be quite small, as it is only slower when the instances are initialized.

CHAPTER 8 : FUNCTIONAL RECURSION

8.1 Definition

In this work, functional recursion is defined as the ability to use a module as a sub-module of itself in order to perform a recursive calculation. We do not consider circular references (when the graph of dependency contains a loop) as a functional recursion. Also, functional recursion differs from structural recursion (which will be discussed in Chapter 9) by the fact that the recursive module is not used as a data structure, but as a computation module.

The modularity mechanism is modified to support functional recursion. In Wizcell, a module can refer to another module using input/output cells mechanism. In the recursive case, the referred module is the same as the referring module. This defines a potentially infinite tree, which cannot be fully instantiated. In order to avoid instantiating this infinite tree, I use the concept of lazy instantiation, meaning that the module will only be instantiated if it is really needed, i.e. it is used by one or more cells. This is the point where the previously discussed dynamic cell references mechanism is used.

A lazy module is instantiated only when a cell refers to it, and it is deleted when the last reference is removed.

This is the only addition required to support recursion.

8.2 Implementation

Recursive references introduce several difficulties for the implementation and some care must be taken.

When the module is initialized, it will call the *load* operation on the *ModuleFactory*. This factory must return the module itself as a result, which means that it must be added in the table of loaded modules before calling the *init* function. If this rule were not respected then several versions of the same module would be repeatedly loaded and initialized. Secondly, the *load* operation would return a module which is not fully initialized, as the *init* operation is not finished. It is the responsibility of the module to initialize itself properly for the support of the public operations before trying to load any sub-module (since it is always possible that one of the sub-modules recursively refers to the original module).

8.3 Example 1: Fibonacci

This module computes the Fibonacci function defined by:

$$f(n) = \begin{cases} 1 & \text{if } n \leq 2 \\ f(n-2) + f(n-1) & \text{else} \end{cases}$$

The implementation is a direct mapping of the mathematical definition, as shown in Figure 5. It is a recursive module *FIB* calculating the fibonnaci function *F* of an input value *N*, using two recursive references *F1* and *F2* to compute *f(N-1)* and *f(N-2)*.

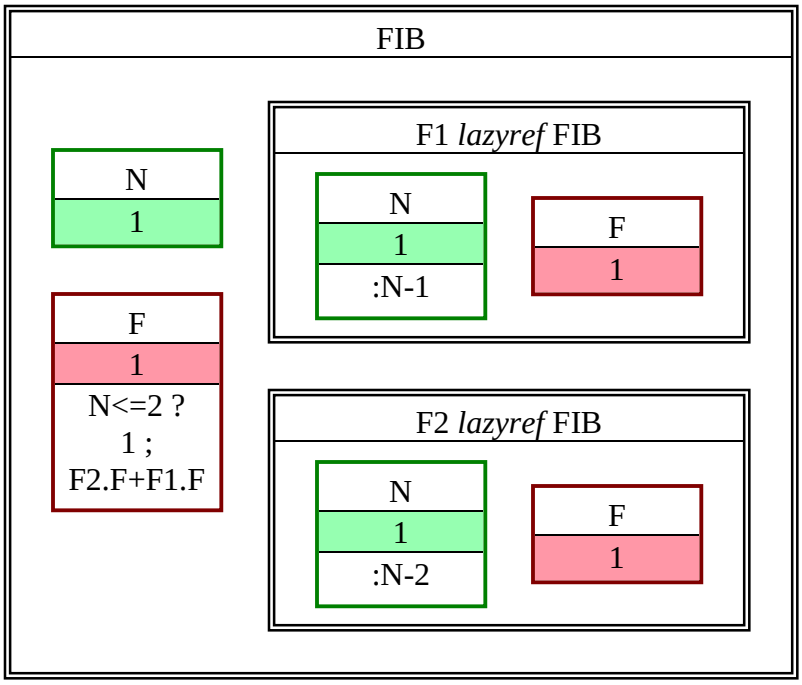

Figure 5: Implementation of the Fibonacci function

This example shows how a recursive mathematical function can be implemented using the functional recursion facility of Wizcell nearly directly. Even if the Fibonacci function can be implemented more efficiently using a loop, the main advantage of the recursive implementation is the direct mapping of the mathematical definition. We could implement a “cached reference” for F1 and F2 storing the already computed results. This would require a data structure containing the last computed instance of the FIB module in the format (N,F) so that optimally each f(n) is computed only once for each value of n, resulting in the same time complexity than the iterative implementation. Note that this will only work when the result of the module only depends on the current value of the input cells (e.g., no time-based mechanisms such as loops are used).

8.4 Example 2: Quick Sort

The Qsort module sorts an array contained in the InputCell IN, giving the result in the OutputCell OUT.

Two cases occurs:

1. The array is less than two elements. The result is the input array array (end of recursion).

2. The array is two elements or more. A *root* (pivot) element is chosen (the first element in this implementation), and the module is recursively applied to the *Left* and *Right* part of the input array (e.g. the elements less than the pivot and the elements greater than or equal to the pivot). The result is computed by concatenating the *Left* result with the *root* element and the *Right* result.

The Wizcell implementation is shown in Figure 6, where the input array is specified in the cell *IN*, and the result is calculated in the *OUT* cell.

A very interesting property of this approach is the fact that all created cell structures are kept in memory after the computation and can be reused later on. This can be very efficient in cases in which the input change is only partial. For instance, the same sub-array doesn't need to be re-sorted.

Note that all the root cells implicitly create a sorting tree. In fact the difference between this implementation of QuickSort and a Binary Sorting Tree implementation is quite small. See the final example at the end of the next chapter for a comparison.

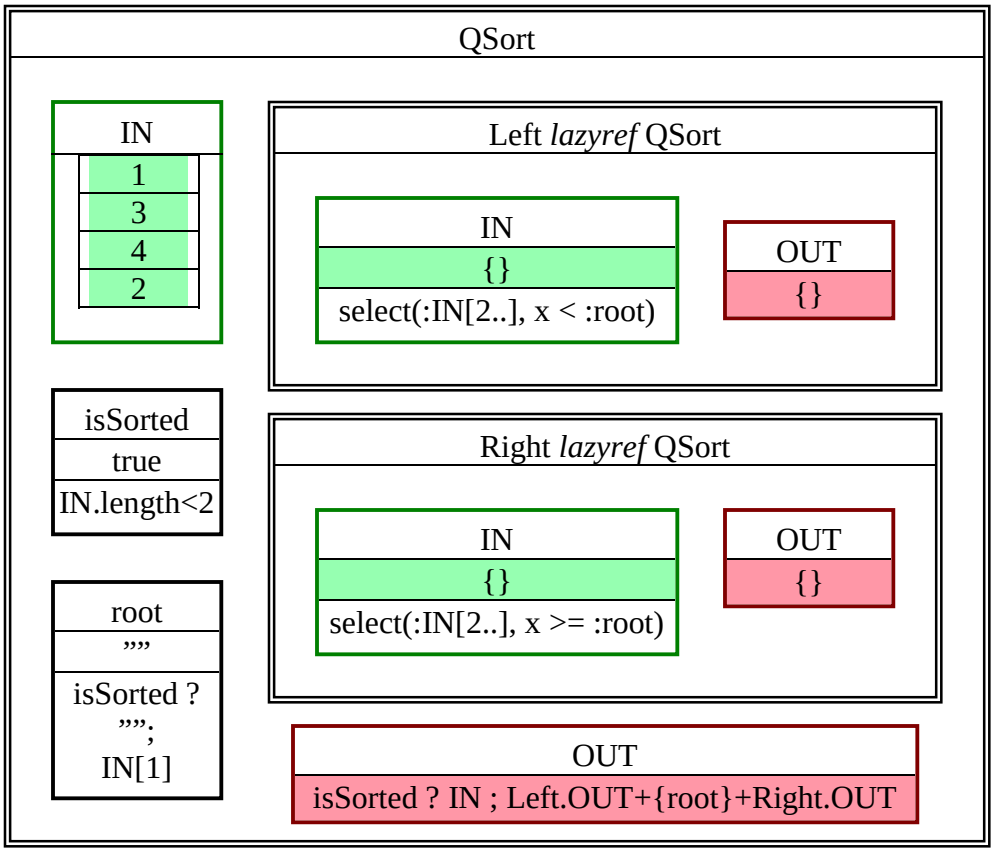

Figure 6: Implementation of the recursive QuickSort algorithm

CHAPTER 9 : STRUCTURAL RECURSION

Structural recursion support is pretty straightforward by now. We have the array mechanism that already gathers the values of multiple cells into one data structure ready for operations, and the functional recursion concept that defines self-containing modules. The structural recursion mechanism derives from both to define a simple way to provide complex unbounded data structures in Wizcell.

The only difficulty concerns the specification of the data type of each cell. If we want an efficient manipulation of the data, we must enforce the type of structure used, so that we can resolve the textual names into indexing IDs during the compilation phase for fast access later on.

9.1 Data type specification

A special type of Module, which only contains InputCells, defines a data type. It is equivalent to structured data structures in procedural languages.

For example, a structure storing a banking operation is defined in Figure 7. It contains several cells, each one defining an attribute and its default value.

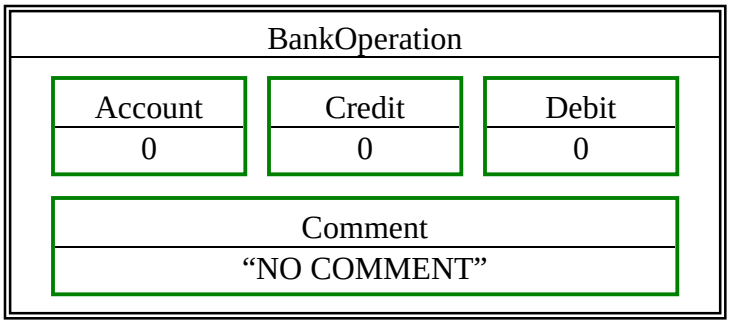

Figure 7: Definition of a Bank Operation data structure

9.2 Data type use

All cells using a data type must specify the name of the type as the initial value, preceded by the character ‘#’.

The sub-cells of a data structure can themselves refer to a data type, even the parent data type, allowing for recursive data structure. For example, a binary tree data structure is defined in Figure 8.

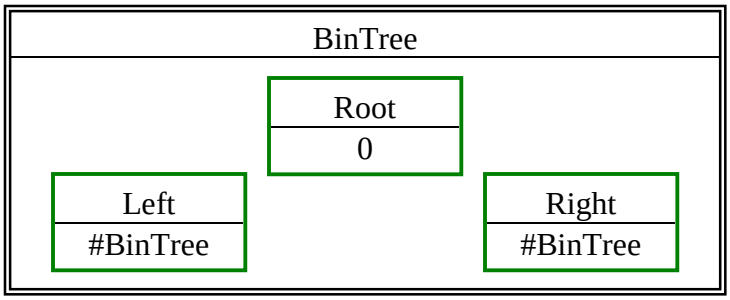

Figure 8: Definition of a Binary Tree data structure

When a cell refers to a data type using the ‘#’ operator, its initial value is *null*, which is a special constant designating an empty data structure. To instantiate a data type we must use a module reference identical to the one used for functional recursion. For example, Figure 9 defines the instantiation of the tree (1, (2), 3).

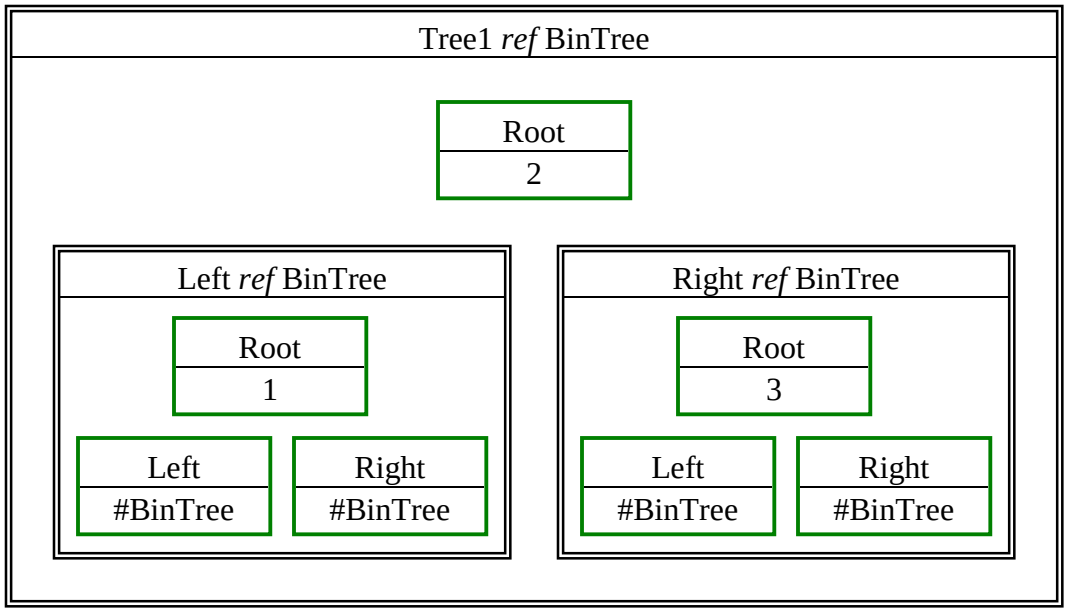

Figure 9: Instantiation of a binary tree

Note: we could remove the empty Left and Right sub-modules, as they are empty by default. This would lead to a quite clean representation of the tree.

9.3 Final example

As a final example, a sorting algorithm will be presented using a binary sorting tree as intermediate data structure. Three modules are used. The *BuildTree* module (Figure 10) takes the array as input and produces the sorted binary tree. The *Infix* module (Figure 11) takes the sorted binary tree as input and produces an array with the elements of the tree in infix (sorted) order. Finally the *Main* module (Figure 12) is the program itself. The data type used is the *BinTree* presented earlier (Figure 8).

This example shows how structural recursion can be used to communicate complex data structures between modules. Processing a recursive data structure is quite straightforward using functional recursion. The *Main* program shows how a program is constructed by referring to the used modules and linking their public cells.

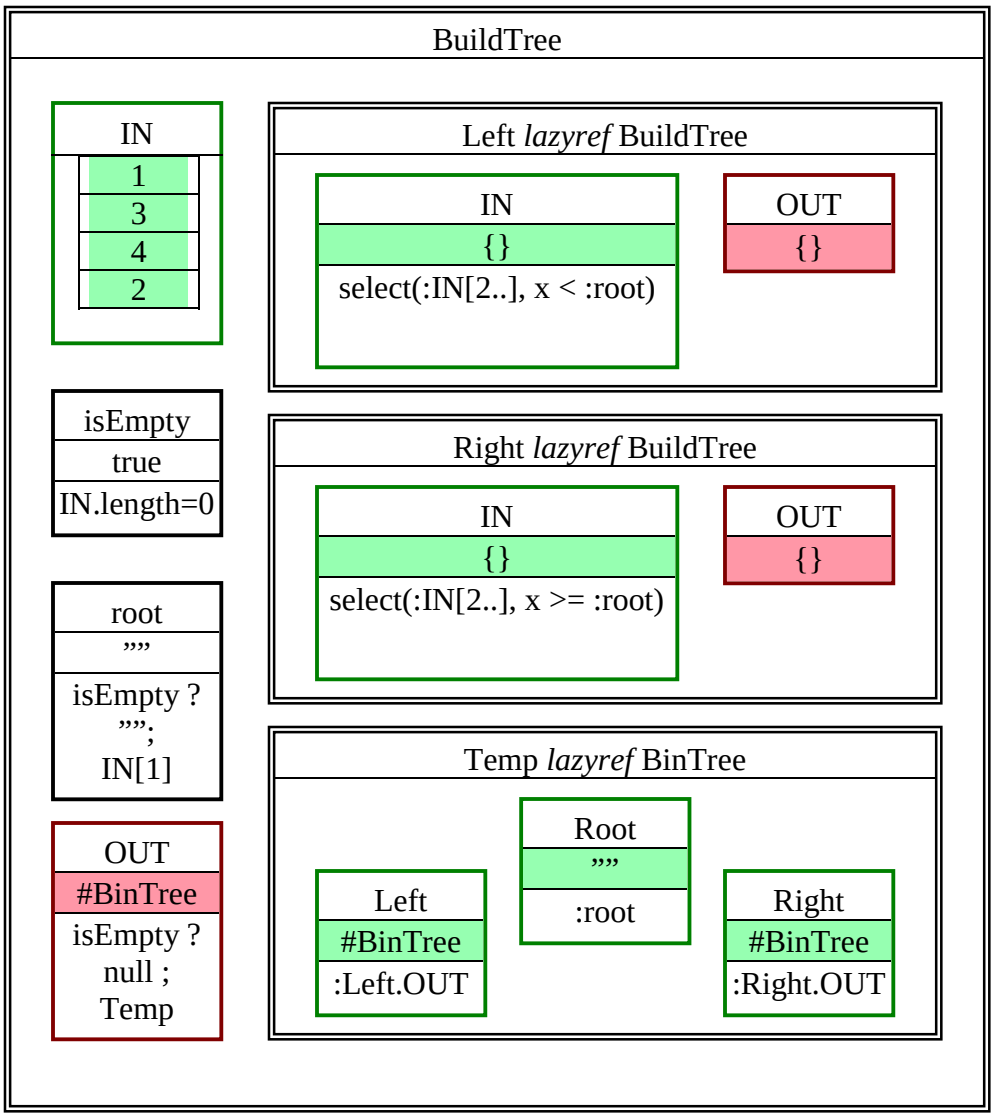

Figure 10: *BuildTree* sorted binary tree creation module

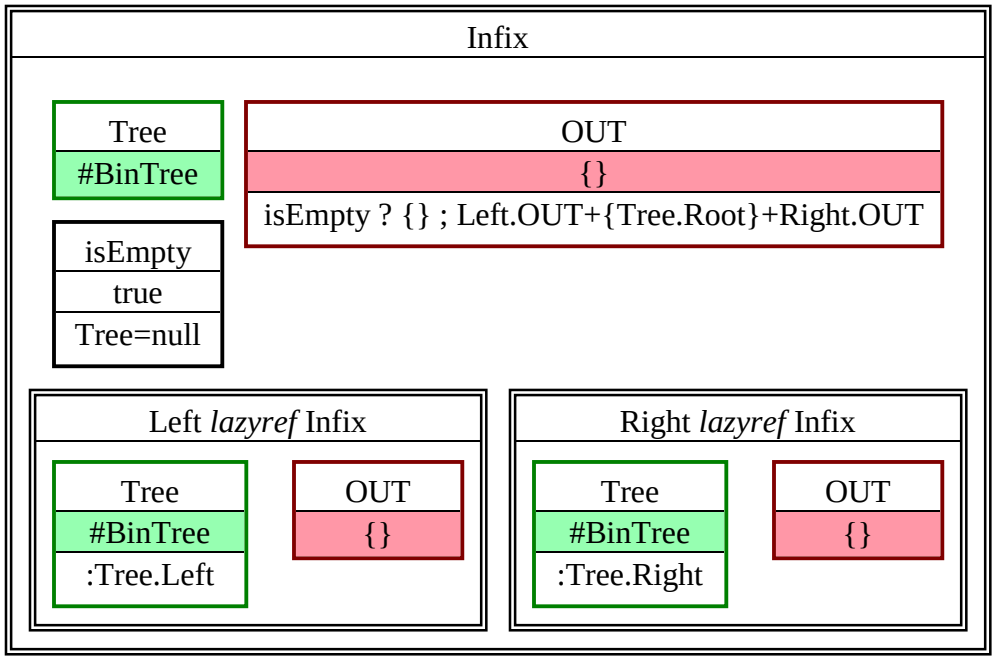

Figure 11: *Infix* binary tree infix conversion to array module

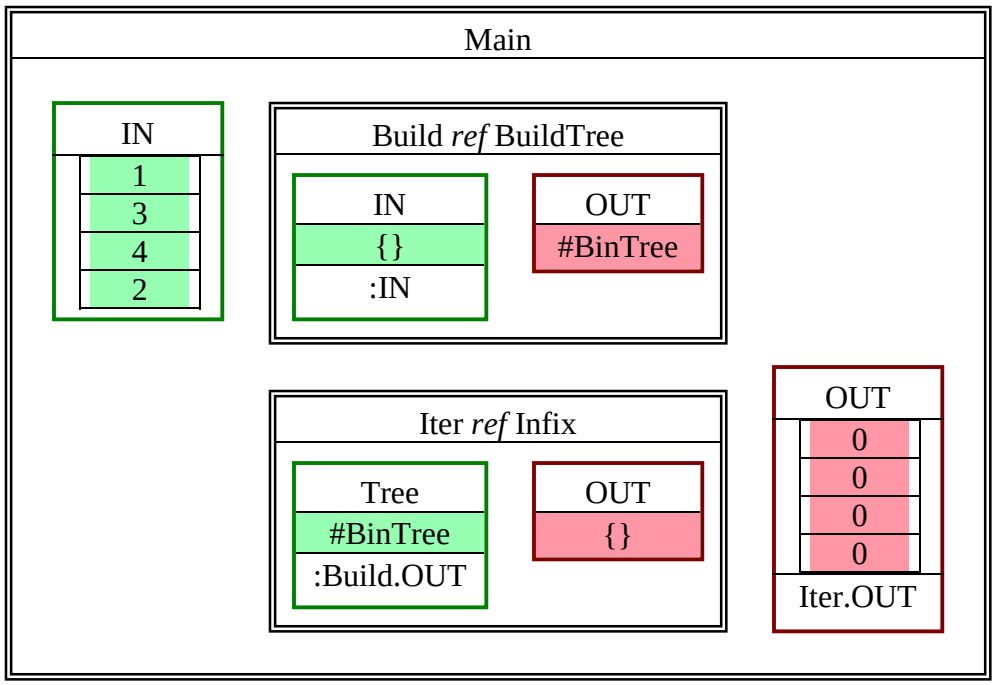

Figure 12: *Main* program

CHAPTER 10 : FORMAL PROOFS

10.1 Foundations

One of the goals of Wizcell is to obtain a language where formal correctness proofs are easier to obtain than in other programming paradigms (e.g. procedural and object-oriented languages). This is due to the presence of Kay's value rule [2], which expresses that a cell's value is exclusively defined by a formula. Therefore, an invariant is associated with each cell, which is the formula of the cell itself. One particularity of Wizcell is that contrarily to "regular" spreadsheet languages, cells' values are not constant through time, and so the update mechanism must be taken into account. The update model I used in this work is the simultaneous update model, where each cell is calculated using the values from the previous step (see Chapter 1). This leads to the first rule, derived from Kay's rule:

Wizcell Value Rule: The value of the cell is equal to the value of its formula applied to the values of the previous step.

This rule reduces to Kay's rule when the program has stabilized, which means when the values of all the cells are stable (in a sense this corresponds to the termination of a Wizcell program).

10.2 Application

I will reuse the example presented in section 9.3 as a proof example. This is the most advanced example presented so far, so it should be the most interesting to prove.

The goal is to prove the correctness of the program (if the program terminates, then the result is correct). The termination itself is not proved.

We need to prove the following proposition:

P_1: when the *Main* program is stabilized, the *Main.OUT* array is a sorted combination of the IN array (in increasing order).

As we only consider the stable case, we can use Kay's value rule.

Proof 1: the *OUT* tree of module *BuildTree exactly* contains the elements of the *IN* array.

Let us prove this by induction on the size n of the *IN* array:

- n=0: *IN* is empty, we have:

IN.length = 0 as *IN* is empty

⇒ *isEmpty* = true from Kay's value rule on cell *isEmpty*

⇒ *OUT* = null from Kay's value rule on cell *OUT*

So *OUT* contains exactly the elements of *IN* (e.g. no element).

- n>0 and ∀ n'< n the proposition is true. We have:

IN.length > 0 as *IN* is not empty

⇒ *isEmpty* = false from Kay's value rule on cell *isEmpty*

(1) *root = IN[1]* from Kay's value rule on cell *root*

(2) *OUT = Temp* from Kay's value rule on cell *OUT*

Left.IN contains the elements of *IN[2..]* less than *IN[1]* (from Kay's value rule on cell *Left.IN*).

$\Rightarrow$ *Left.IN.length* < n as it is at most all the (n-1) elements of *IN[2..]*

So from the induction hypothesis we have:

(3) *Left.OUT* contains the elements of *IN[2..]* less than *IN[1]*

Using the same arguments, we also have:

(4) *Right.OUT* contains the elements of *IN[2..]* greater than of equal to *IN[1]*.

From Kay's value rule on cells *Temp.Left, Temp.Root,* and *Temp.Right*, and from (1), (3), and (4), we have:

Temp.Left contains the elements of *IN[2..]* less than *IN[1]*

Temp.Right contains the elements of *IN[2..]* greater than of equal to *IN[1]*.

Temp.Root contains *IN[1]*

So *Temp* contains:

{ elements of *IN[2..]* less than *IN[1]*} $\cup$ { elements of *IN[2..]* greater than of equal to *IN[1]*} $\cup$ {*IN[1]*}

= *IN[2..]* $\cup$ *IN[1]*

= *IN*

So *Temp* exactly contains the elements of *IN*

From this result and (2) we proved that *OUT* exactly contains the elements of *IN*.

By induction the proposition is proved for all n.

Proof 2: in module *BuildTree*, *OUT* is a sorted binary tree, i.e. all nodes of the tree verify that the *Left* elements are all less than *Root* and the *Right* elements are all greater than or equal to *Root*.

This proof use similar arguments as the first one.

Using the same recursion on the length n of *IN*:

- When the *IN* array is empty, the *OUT* tree is null, so it doesn't have any node and directly verify the proposition.
- When the *IN* array is not empty, and the proposition is true for all smaller arrays, we have:

(1) *Left.OUT* is a sorted binary tree.

(2) *Right.OUT* is a sorted binary tree.

isEmpty = false

OUT = Temp

From Proof 1 we know that in this case we can deduce:

OUT.Left contains the elements of *IN[2..]* less than *IN[1]*

OUT.Right contains the elements of *IN[2..]* greater than of equal to *IN[1]*.

OUT.Root contains *IN[1]*

Combining these three properties leads to the fact that all elements of *OUT.Left* elements are less than *OUT.Root* and all elements of *OUT.Right* are greater than or equal to *OUT.Root.*

From this result, (1) and (2) we obtain the proposition to be proved.

Proof 3: in module *Infix*, if *Tree* is a sorted binary tree, then the *OUT* array contains the elements of *Tree* sorted in increasing order.

This can be proves by induction on the number of elements of *Tree*

- n = 0: *Tree* is null.

We have:

isEmpty = true

$\Rightarrow$ *OUT* = {}

This verifies the proposition.

This can be proves by induction on the number of elements of *Tree*

- n > 0 and the proposition is true for all n' < n.

We need to prove that all consecutives elements of *OUT* are sorted.

From the induction hypothesis, we know that this is true for the elements of *Left.OUT* and of *Right.OUT*. Thus we must only verify that the last element if *Left.OUT* is less than or equal to *Tree.Root* and the first element of *Right.OUT* is greater than or equal to *Tree.Root*.

All elements of *Left.OUT* come from *Left.Tree* = *Tree.Left*. As *Tree* is sorted all elements of *Tree.Left* are less than *Tree.Root*. So the last element of *Left.OUT* is less than *Tree.Root*. Using the same argument on *Tree.Right* we obtain that the first element of *Right.OUT* is greater than or equal to *Tree.Root* so we can conclude that *OUT* is sorted.

We also know from the induction hypothesis that *Left.OUT* contains the elements of *Tree.Left* and *Right.OUT* contains the elements of *Tree.Right*.

So from the definition of the array concatenation operation *OUT* contains the elements of *Tree*.

So *OUT* contains the elements of *Tree* sorted in increasing order.

By induction the proposition is verified for all n.

Conclusion:

From Kay's value rule on cell *Main.Build.IN*, we have:

(1) *Main.Build.IN = Main.IN*

From Proof 1 we know that:

(2) the tree *Main.Build.OUT* contains the elements of *Main.Build.IN*

From Proof 2 we know that:

(3) the tree *Main.Build.OUT* is a sorted binary tree

From Kay's value rule on cell *Main.Iter.Tree*, we have:

(4) *Main.Iter.Tree = Main.Build.OUT*

From Proof 3 we know that:

(5) *Main.Iter.OUT* contains the elements of *Main.Iter.Tree* in increasing order

From (3),(4),(5) we know that:

(6) The elements of *Main.Iter.OUT* are sorted in increasing order

From Kay's rule on cell *Main.OUT* we have:

(7) *Main.OUT = Main.Iter.OUT*

Combining (1), (2), (4), (5) and (7) we have:

(8) *Main.OUT* contains the elements of *Main.IN*

Propositions (6) and (8) forms the proposition P_1 which was to be proved.

CONCLUSION

In this thesis several concepts were presented and included in the Wizcell language: modularity, time, recursion, array, functional recursion and structural recursion. Their combination leads to language with improved features, usable for both fast software prototyping and complex software engineering.

Currently, the Wizcell virtual machine implementation supports all the presented concepts except structural recursion, which should be added very soon. The graphical tools (program development and execution) needs also to be modified to support arrays and recursion.

Several issues are still unresolved. Integrating a loop in a module adds a lot of complexity in the code and seems not feasible for all potential users. Graphical representation of complex data structures (structured types, array of structures containing cells and maybe other arrays) also needs to be improved.

As a future extension, support for more object-oriented like features in structural recursion (inherence, formulas) can be added. The current design should allow this addition, but this kind of features may be too advanced and confusing for the user.

The primitive ideas on formal proofs presented at the end of this thesis are promising. It seems that at least for simple cases (no delay involved) invariants from cells' formula are really helpful to prove the correctness of the program. This is a very

important advantage of this paradigm. It may even be possible to create semi-automatic theorem provers to help the user in this domain. Proofs of termination as well as efficiency/complexity analysis are still an open area.

REFERENCES

[1] Burnett, M. et al.: Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. Journal of Functional Programming, vol. 11, no 2. (2001) 155–206

[2] Kay, A.: Computer Software. Scientific American 251(3), September (1984) 52–59

[3] Rust, H.: Modeling the generalized Railway Crossing with Hybrid State Machines. Transportation 2000, June (2000) 138–145

VITA

Jérémie Allard was born in Chatenay, France. He received his License and Maîtrise in Computer Sciences from the Université d'Orléans, France in 2001.