Introduction to AsmGofeer

Joachim Schmid
Siemens Corporate Technology
D-81730 Munich
joachim.schmid@tydo.de

March 26, 2001

AsmGofer [12] is an advanced Abstract State Machine (ASM) [2] programming system. It is an extension of the functional programming language Gofer [9] which is similar to Haskell [14]. More precisely, AsmGofer introduces a notion of state and parallel updates into TkGofer [15] and TkGofer extends Gofer to support graphical user interfaces.

AsmGofer has been used successfully for several applications. For instance, *Java and the Java Virtual Machine* [13], the *Light Control Case Study* [3], *Simulating UML Statecharts* [5].

In this document we introduce the AsmGofer programming environment. We assume basic knowledge in functional programming, like algebraic data types, functional expressions, and pattern matching. For an introduction into functional programming we refer the reader to [1, 16]. This document is not a reference manual for AsmGofer.

In Section 1 we briefly explain the Gofer command line interface for loading, editing, and running examples. Section 2 and Section 3 introduce the notations and constructs for sequential and distributed ASMs in AsmGofer. In Section 4 we use sequential ASMs to define the well-known *Game of Life* example. For this example, we show in Section 5 how our GUI generator works, and in Section 6 we define a customized GUI for the game. Finally, Section 7 concludes this document with a summary.

1 The interpreter

Gofer and AsmGofer are interpreters. In this section we describe the command line interface of Gofer and AsmGofer. After starting AsmGofer, the output looks as follows:

```
=== AsmGofer (TkGofer v2.0) ===
AsmGofer Version 1.1 (c) Joachim Schmid 1999-2001
(with modules: guiMonad, widget, time, io, concurrent, asmvar)
Gofer Version 2.30a  Copyright (c) Mark P Jones 1991-1994
```

```
Reading script file "AsmGofer/Preludes/tk.prelude-all-asm":
Gofer session for:
AsmGofer/Preludes/tk.prelude-all-asm
Type :? for help
?
```

At startup the system reads a prelude file where all library functions (*map*, *head*, *tail*, *fst*, ...) are defined. We use the prelude `tk.prelude-all-asm`. The Gofer system is ready for new commands when the `?` prompt occurs. One can now evaluate arbitrary expressions, load and edit files, load projects, and find function and type definitions. In the following subsections we explain these features of the command line.

1.1 Expression evaluation

Expressions can be evaluated in Gofer using the command line. For instance, typing `5+2` and pressing the *enter* key results in `7` being displayed.

```
? 5+2
7
(3 reductions, 7 cells)
?
```

The Gofer system evaluates the expression and reports the result together with information about the used cells[1] and reduction steps. This information gives an impression about the complexity and size of the evaluation. Another short example is the sum of squares of numbers from one to ten.

```
? sum (map (\x -> x*x) [1..10])
385
(124 reductions, 185 cells)
?
```

When the user enters an expression, Gofer first tries to typecheck it. If the expression has a unique type, then Gofer evaluates the expression, otherwise a type error is reported. For example, consider the wrongly typed expression `5+"Hello"`.

```
? 5+"Hello"
ERROR: Type error in application
*** expression     : 5 + "Hello"
*** term           : 5
*** type           : Int
*** does not match : [Char]
?
```

[1] A cell is a synonym for head element.

1.2 Dealing with files

The command line can be used to evaluate expressions, but not to provide definitions. Functions and types can be defined only in files (also called scripts). The command `:l filename.gs` loads the file *filename.gs*. File names do not have to end with `.gs`, but this is a usual extension for gofer scripts. The `:l` command removes all definitions of previously loaded files except the definitions in the prelude file. To append definitions contained in another file *file*2*.gs*, the command `:a file2.gs` can be used.

If a file is modified, then `:r` reloads all dependant files. In case there is a syntax or a type error, the interpreter shows the file name and the line number where the error occurred. Typing `:e` opens an editor with the corresponding file at the corresponding line number. The editor used by Gofer can be determined in the environment variables `EDITOR` and `EDITLINE`.

Especially for many files, it is cumbersome to load files with the `:a` command. Therefore Gofer supports so-called project files; each line in a project file contains a file name. The files in *file.p* are loaded by `:p file.p` in the order in which they appear in `file.p`. The reload command `:r` works for projects, too. Additionally, an arbitrary file can be edited by typing `:e filename.gs`.

1.3 Other commands

As already mentioned, Gofer typechecks every expression. Furthermore, the interpreter supports a command to print the type of a given expression. Consider the input `:t "Hello"`.

```
? :t "Hello"
"Hello" :: String
?
```

The command `:t` determines the type and prints it to standard output. The operator `::` separates the type information from the expression. In our case the type is *String*. Additional information will be displayed by typing `:i String`.

```
? :i String
-- type constructor
type String = [Char]
?
```

Now Gofer tells us, that *String* is an alias for [*Char*] which is a list of characters.

Gofer remembers the file name and line number for each function and type definition. Entering `:f sum` instructs Gofer to open the editor with the corresponding file and to jump to the position where *sum* is defined.

The command to quit the interpreter is `:q`.

```
? :q
[Leaving Gofer]
joe:  >
```

Gofer supports several other commands. Typing `:?` prints a list of known commands and short descriptions. For further information we refer the reader to the Gofer manual which is included in the AsmGofer distribution [12].

2 Sequential ASMs

Encoding ASMs in a purely functional programming language like Gofer seems to be a contradiction on terms, because a pure functional language has no side-effects, whereas each ASM update has a main-effect on the global state. In fact, AsmGofer is not Gofer with additional definitions in Gofer to support ASMs. Rather, AsmGofer modifies the evaluation machine in the Gofer run-time system to support a notion of state. On the other hand, we do not change the Gofer syntax and therefore we have to represent the ASM features as expressions. We provide special functions and operators to define dynamic functions and to perform updates, as we are going to explain in this section.

2.1 Nullary dynamic functions

Since we do not provide any special ASM syntax, we have to represent a dynamic function as an ordinary functional term. For this purpose, the prelude contains a function *initVal* with the following signature (see below for an explanation of *Eq*):

$$initVal :: Eq\ a \Rightarrow String \rightarrow a \rightarrow a$$

With this function we can create a 0-ary dynamic function f by the following definition:

$$f = initVal\ \texttt{"name"}\ init$$

The first argument `"name"` is a name for the dynamic function which is used only in error messages. Due to technical reasons in the Gofer implementation, we cannot access the function name f as given by the lefthand-side expression of the function definition. The second argument *init* is the initial value for f. The *initVal* function is defined in such a way, that the return type of the function is equal to the type of the initial value, as can be seen from the signature declaration above.

In Gofer there is no need to define function signatures for function definitions. Usually, Gofer can deduce the types. However, we suggest to define the signatures anyway, because this enhances the readability of type error reports by Gofer. To improve readability we further suggest to write signatures for dynamic functions with the type alias *Dynamic*:

$$\textbf{type}\ Dynamic\ t = t$$

Note that this definition of *Dynamic* implies that there is *no* semantic difference between a type A and *Dynamic* A. *Dynamic* is added only for better readability.

A declaration of a dynamic function may then look as follows:

$f :: \mathit{Dynamic\ Int}$
$f = \mathit{initVal}$ `"f"` 0

The notation $Eq\ a \Rightarrow \ldots$ in the signature for *initVal* means that the type a must be an instance of type class *Eq*. We refer the reader to [7] for a discussion of type classes. In the following we use the term *class* as an alias for type class. Being an instance of *Eq* implies that the equality operator $==$ is defined for that type. AsmGofer needs this operator for the consistency check of updates, namely to determine whether two values are equal. All basic types in AsmGofer are already defined as an instance of class *Eq* and for all user-defined types, the user has to provide the corresponding definition. Alternatively, one can define a type to be an instance of the class *AsmTerm* (defined in the prelude), which makes it an instance of class *Eq* using the type class features of Gofer. For more information about type classes in Gofer, we refer the reader to [8].

data *MyType* $= \ldots$

instance *AsmTerm MyType*

For AsmGofer, a dynamic function behaves similarly to other functions. For instance, we can enter f in the command line to evaluate the function:

```
? f
0
(5 reductions, 19 cells)
?
```

In our definition of f we defined zero as the initial value for the dynamic function. It is also possible to use the special predefined expression *asmDefault*, which corresponds to an undefined value for each type, defined as an instance of class *AsmTerm*:

$\mathit{asmDefault} :: \mathit{AsmTerm}\ a \Rightarrow a$

We can now use this undefined value to define the dynamic function f:

$f :: \mathit{Dynamic\ Int}$
$f = \mathit{initVal}$ `"f"` $\mathit{asmDefault}$

This *undefined* value is not an implementation of the value `undef` of the *Lipari Guide* [6]. In particular, AsmGofer can not use this undefined value in computations. Therefore, whenever an expression evaluates to *asmDefault*, AsmGofer stops the computation and reports an error message. Note that Gofer uses lazy evaluation [11] which implies that expressions are evaluated only when necessary.

```
? f
(5 reductions, 18 cells)
ERROR: evaluation of undefined 'asmDefault'
*** dynamic function: "f"
?
```

However, it is possible to specify for any type an own *undefined* value. For example we could define 0 as the undefined value for expressions of type *Int*, as in the following instance definition.

instance *AsmTerm Int* **where**
  asmDefault $= 0$

This definition implies that 0 and *asmDefault* are treated as equal. This will be become more interesting for unary dynamic functions as discussed in the next subsection.

The above kind of *undefined* is more flexible than the *undefined* in the *Lipari Guide*, where *undefined* is treated like an ordinary value which can be used in computations. In AsmGofer one can use the predefined *asmDefault* expression where a computation is abrupted whenever this expression occurs, but one can also define an element which should be used instead of *undefined*.

2.2 Unary dynamic functions

We provide a function *initAssocs* which can be used to define unary dynamic functions.

$$\mathit{initAssocs} :: (\mathit{AsmOrd}\ a, \mathit{Eq}\ b) \Rightarrow \mathit{String} \rightarrow [(a, b)] \rightarrow \mathit{Dynamic}(a \rightarrow b)$$

The first argument is the name for the dynamic function, which is used only in error messages. The second argument is an initialization list. If this list is empty, then the dynamic function is undefined (in the sense above) for each argument. In the type signature for *initAssocs* we can see that type a must be an instance of class *AsmOrd* and b an instance of class *Eq*. Requiring b to be an instance of *Eq* allows one to determine whether two values are equal when checking the consistency of updates. Requiring a to be an instance of *AsmOrd* is used to compare two arguments. The equality operator would be sufficient, but we can implement unary dynamic functions more efficiently using binary search, if there is an ordering on the argument type. The class *AsmOrd* is defined as follows.

class *AsmOrd a* **where**
  $\mathit{asmCompare} :: a \rightarrow a \rightarrow \mathit{Int}$

The function *asmCompare* returns for two arguments either -1, 0, or 1 depending on whether the first argument is less than, equal to, or greater than the second argument. If we define a type as an instance of *AsmTerm*, then it automatically becomes an instance of class *AsmOrd*.

Similarly to nullary dynamic functions we can introduce unary dynamic functions, but using *initAssocs* instead of *initVal*.

$$
\begin{aligned}
&g :: \mathit{Dynamic}(\mathit{Int} \to \mathit{Int})\\
&g = \mathit{initAssocs}\ \texttt{"g"}\ [(0,1),(1,1)]
\end{aligned}
$$

The function g can be used like other unary functions, except when the function should be evaluated for an argument which is not in the domain of the function. In that case, AsmGofer uses the expression *asmDefault* already introduced above.

```
? g 1
1
(6 reductions, 16 cells)
? g 2
(4 reductions, 15 cells)
ERROR: evaluation of undefined 'asmDefault'
*** dynamic function: "g"
?
```

Additionally, AsmGofer supports some other functions to determine the domain and the range of unary dynamic functions, to check whether an expression is in the domain of a function, and to compute the current association list (function represented as a finite mapping).

$$
\begin{aligned}
&\mathit{dom} && :: \mathit{Ord}\ a \Rightarrow \mathit{Dynamic}(a \to b) \to \{a\}\\
&\mathit{ran} && :: \mathit{Ord}\ b \Rightarrow \mathit{Dynamic}(a \to b) \to \{b\}\\
&\mathit{inDom} && :: a \to \mathit{Dynamic}(a \to b) \to \mathit{Bool}\\
&\mathit{assocs} && :: \mathit{Dynamic}(a \to b) \to [(a, b)]
\end{aligned}
$$

The predefined Gofer class *Ord* defines the operators $<, \leq, >, \geq$. The type $\{a\}$ is the type corresponding to the power set of type a similar to the list type $[a]$ except that there are no duplicate values. The requirements on class *Ord* are used to sort the expressions in a set. The domain of a dynamic function only contains those expressions which are mapped to a value different from the *undefined* element for the corresponding type.

2.3 Update operator

Up to now we introduced nullary and unary dynamic functions. Now the question arises how we can update dynamic functions. As usual in ASMs we provide the := operator.

$$(:=) :: \mathit{AsmTerm}\ a \Rightarrow a \to a \to \mathit{Rule}\ ()$$

The operator takes two arguments of the same type (which must be an instance of *AsmTerm*) and returns something of the special type *Rule* (). We use this type to represent rules. Additionally, we use the *do notation* for monads [14] in

Gofer to denote parallel execution of rules. The *do notation* and monads [17] are not described in this chapter, because this would explode this introduction. Roughly spoken, the *do notation* for rules in AsmGofer can be viewed as taking a set of rules and combining them to one rule as in the example below for *someUpdate*.

```
someUpdate :: Rule ()
someUpdate = do
  f   := 5
  g 2 := 7 + f
```

The other basic rule is the *skip* rule which has the empty set as update set.

```
skip :: Rule ()
```

This *skip* rule is especially useful in *if-then-else* expressions when no else part is needed.

```
someOtherUpdate :: Rule ()
someOtherUpdate =
  if f == 2 then
    g 2 := 7 + f
  else skip
```

2.4 N-ary dynamic functions

We do not provide syntax for dynamic functions with arity greater than one. However, such dynamic function can be represented as a unary dynamic function by using a tuple for the arguments. Additionally, one can define an auxiliary dynamic function as illustrated in the following example for a 2-ary dynamic function g:

```
g_aux :: Dynamic((Int, String) → String)
g_aux = initAssocs "g" some_init

g :: Int → String → String
g i s = g_aux(i, s)
```

The function g is a 2-ary function. We can use g to access the values of *g_aux*. On the other hand, we can also use g to update the dynamic function *g_aux*, because $g\ i\ s$ and *g_aux*(i, s) are treated equally by AsmGofer.

```
my_updates :: Rule ()
my_updates = do
  g 5 "great" := "hello"
  g 7 "other" := g 3 "strange"
```

2.5 Execution of rules

In the previous subsections we defined dynamic functions and updates to them. The question is how to execute the updates and especially, what to do with the type *Rule* () ? Expressions of type *Rule* () correspond to rules and have a side-effect on the global state. Gofer supports *IO actions* [10] of type *IO* (), which are used to perform input-output-operations like printing a string on standard output. Printing a string is a side-effect. In Gofer, this side-effect is implemented by a primitive function which on evaluation prints the corresponding string on standard output. The monad ensures that the function must be evaluated in order to proceed. We use the same technique to define primitive functions having a side-effect on a global state to implement dynamic functions. However, to distinguish in the type system between IO actions and rules, we use an abstract type *Rule* and we provide the following functions to transform expressions of type *Rule* () into IO actions which can be executed by the Gofer interpreter.

$$
\begin{array}{ll}
\mathit{fire} & :: \mathit{Int} \rightarrow \mathit{Rule}\,() \rightarrow \mathit{IO}\,() \\
\mathit{fire}1 & :: \mathit{Rule}\,() \rightarrow \mathit{IO}\,() \\
\mathit{fireWhile} & :: \mathit{Bool} \rightarrow \mathit{Rule}\,() \rightarrow \mathit{IO}\,() \\
\mathit{fireUntil} & :: \mathit{Bool} \rightarrow \mathit{Rule}\,() \rightarrow \mathit{IO}\,() \\
\mathit{fixpoint} & :: \mathit{Rule}\,() \rightarrow \mathit{IO}\,()
\end{array}
$$

The first function *fire* takes two arguments. The first argument is the number of steps to execute the rule specified by the second argument.

```
? fire 2 someUpdate
2 steps
(102 reductions, 228 cells)
?
```

The *fire*1 function is a specialization of *fire* where the number of steps to execute is fixed to 1. The *fireWhile* and *fireUntil* functions take a condition and a rule as arguments and fire while or until the condition holds. The *fixpoint* function fires its argument as long as the resulting update set is not empty.

Execution of rules is possible only if the corresponding update set is consistent. An update set is inconsistent if it contains updates to assign two different values to the same location.

```
? fire1 (do f := 1; f := 2)
(22 reductions, 60 cells)
ERROR: inconsistent update
*** dynamic function : "f"
*** expression (new) : 2
*** expression (old) : 1
?
```

2.6 Rule combinators

The *Lipari Guide* [6] introduces several rules like *import*, *extend*, *choose*, and *var over*. We have implemented *forall*, *choose*, and *create* where *forall* corresponds to *var over* and *create* to the *import* rule in the *Lipari Guide*. The result type of the rule combinators in this subsection is always *Rule* ().

Our *forall* rule takes a range constraint similar to list comprehension in Gofer and a rule to execute.

```
**forall** $i \leftarrow dom(g)$ **do**
  $g\ i := g\ i + 1$
```

In this example the rule body is executed for each i in the domain of g. It is also possible to loop over several variables.

```
**forall** $i \leftarrow dom(g), j \leftarrow \{1..10\}$ **do**
  $h(i, j) := g\ i + j$
```

On the other hand, the *choose* rule below chooses one i in the domain of g and executes the body. If the domain of g is empty, then the rule is equivalent to *skip*.

```
**choose** $i \leftarrow dom(g)$ **do**
  $f := f + g\ i$
```

We provide an alternative *choose* rule where we can determine with an *ifnone* clause what should happen when the range constraint is empty.

```
**chooseIfNone** $i \leftarrow dom(g)$ **do**
  $f := f + g\ i$
**ifnone**
  $f := 0$
```

In this example the update $f := 0$ is performed if the domain of g is empty, otherwise an element in the domain is chosen as in the *choose* rule above.

The *Lipari Guide* [6] supports an *import* rule which takes anonymous elements from a special universe *reserve*. Imported elements are no longer in that universe and no element of *reserve* is an element of any other universe. In AsmGofer we want to support something similar. However, it is difficult to implement the semantics of the *import* rule according to the *Lipari Guide* in a functional language with algebraic data types. Therefore, we provide a *create* rule which can be used to deal with anonymous elements. The rule only ensures that a "created" expression was never created previously by a *create* rule, and if two *create* rules are executed in parallel, then both elements are different. Consider the following example for creating heap references.

```
**create** $ref$ **do**
  $heap(ref) := Object(..)$
```

The *create* rule works for expressions of type *Int*. When the necessity arises to use the *create* rule for a type different from *Int*, then we have to define this type as an instance of the following type class *Create*.

```
class Create a where
  createElem :: Int → a
```

When defining a type *a* as an instance of class *Create* we must provide an implementation for the *createElem* function. This function expects an integer value as its argument and transforms it to an expression of the corresponding type *a*. It is important, that the definition of *createElem* is an injective function. Otherwise the *create* rule does not "create" always different elements of the corresponding types.

```
instance Create MyType where
  createElem i = ...
```

Sometimes it is useful to choose one rule among a set of rules. For that reason we provide the *choose among* rule.

```
choose among
  f := f + 1
  f := f + 2
  f := f + 3
```

In [4] the concepts of sequential execution and iteration of rules is introduced. Both concepts are implemented in AsmGofer by the functions *seq* and *iterate*.

```
seq     :: Rule () → Rule () → Rule ()
iterate :: Rule () → Rule ()
```

The result of *seq* is the sequential execution of the argument rules. The second rule is executed in the intermediate state established by the first rule. The *iterate* construct is similar to the *fixpoint* function in the previous subsection, except that the result is a rule and not an IO action. Note also, that intermediate states are not visible to other rules. Both constructs are atomic and executed in one step.

3 Distributed ASMs

In the previous section we described constructs for sequential ASMs. In the *Lipari Guide* [6] there is also a definition for distributed (or multi agent) ASMs. In sequential ASMs there is one agent firing always the same set of rules. In a distributed ASM there are several agents firing rules. Furthermore, the set of active agents might be dynamic.

In our implementation of multi agent ASMs we can define for each agent a rule to execute. Such a rule gets as its first argument (*self* in the example in

Figure 1 Dining philosophers

```
type AgentRule a = a → Rule ()
type Philosopher = Int
data Fork        = Up(Philosopher) | Down
data Mode        = Think | Eat

instance AsmTerm Fork where
  asmDefault = Down

instance AsmTerm Mode where
  asmDefault = Think

fork :: Dynamic(Philosopher → Fork)
fork = initAssocs "fork" []

mode :: Dynamic(Philosopher → Mode)
mode = initAssocs "mode" []

phils :: Dynamic(Philosopher → AgentRule Philosopher)
phils = initAssocs "phils" [(ph1, exec), (ph2, exec), ...]

exec :: AgentRule Philosopher
exec self =
  if mode(self) == Think ∧ lfork == Down ∧ rfork == Down then do
    lfork      := Up(self)
    rfork      := Up(self)
    mode(self) := Eat
  else if mode(self) == Eat then do
    lfork      := Down
    rfork      := Down
    mode(self) := Think
  else skip
where lfork = fork(self)
      rfork = fork(right)
      right = (self + 1) `mod` card(dom(phils))
```

Fig. 1) the agent which executes the rule. For instance, consider the definitions for the well known *Dining Philosophers* problem in Fig. 1 where each philosopher is an agent. In the figure the functions *fork* and *mode* are dynamic functions parametrized over a philosopher. It is important that we parametrize the *Up* constructor over a philosopher, too. Otherwise we do not know which fork is used by which philosopher. The dynamic function *phils* assigns to a philosopher the *exec* rule. In our case each philosopher executes the same rule.

We provide a special function *multi* with the following signature to execute agents.

$$multi :: Dynamic(a \rightarrow AgentRule\ a) \rightarrow Rule\ ()$$

This function takes as its argument a dynamic function like the function *phils* in Fig. 1. The function result is a rule. The implementation of *multi* chooses non-deterministically a subset of the domain of *phils* and executes in parallel for each element in this subset the corresponding rule. This could be described by the following pseudo rule.

```
$multi\ actions =$
  forall $act \leftarrow some\_subset(dom(actions))$ do
    $(actions\ act)(act)$
```

Note that not necessarily all agents execute the same rule as in our example. It is important that *multi* never chooses a subset which leads to an inconsistent update. This is useful in particular for our example, because the rule

$$multi\ phils$$

never chooses a set of philosophers where a fork is shared by two philosophers, because then we would get an inconsistent update for the dynamic function *fork*.

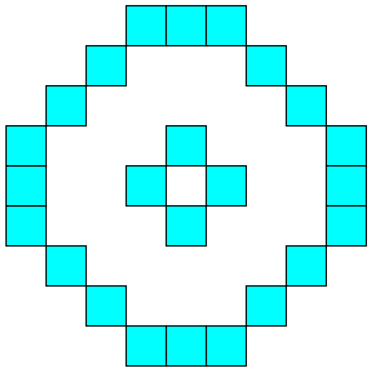
4 An example: Game of Life

In this section we briefly introduce Conway's well-known *Game of Life* and then we show how to formulate the static and dynamic semantics in AsmGofer. Figure 2 shows a typical pattern of this game. The game consists of a $n \times m$ matrix; each cell is either alive or dead. The rules for survival, death and birth are as follows (see [18]):

- *Survival:* each living cell with two or three alive neighbors survives until the next generation.
- *Death:* each living cell with less than two or more than three neighbors dies.
- *Birth:* each dead cell with exactly three living neighbors becomes alive.

In the following two subsections we illustrate the use of AsmGofer by means of the *Game of Life* example. We define the static and dynamic semantics.

Figure 2 Conway's Game of Life



4.1 Static semantics

The definitions in this subsection are ordinary Gofer definitions. In the next subsection when presenting the dynamic semantics we use constructs which are available only in AsmGofer.

Each cell in the matrix is either alive or dead and therefore we define a type *State* consisting of the two constructors *Dead* and *Alive*.

data *State* = *Dead* | *Alive*

Both constructors can be viewed as nullary functions creating elements of type *State*. Note that in Gofer, type names and constructor names must always start with an upper case letter. Functions on the other hand with a lower case letter.

For the representation of cells we use pairs of integer values and we define a type *Cell* as an alias for them.

type *Cell* = (*Int*, *Int*)

Further, we define a nullary function *cN* to denote the number of columns and rows. In order to loop later through all possible cells, we define a function computing such a list. In the definition below we use the concept of list comprehension. The function result is a list of pairs (i, j) where i and j range from 0 to $cN-1$. The order in the list is as follows: $[(0,0),(0,1),\ldots,(0,cN-1),(1,0),\ldots]$.

$cN :: Int$
$cN = 8$

$cells :: [Cell]$
$cells = [(i,j) \mid i \leftarrow [0..cN-1], j \leftarrow [0..cN-1]]$

In the rules for *Game of Life* we need for each cell the number of alive neighbors. A cell has at most 8 neighbors. The following definition computes a list of tuples (i', j') where (i', j') is a neighbor different from (i, j) and a valid position in the game.

```
$neighbors :: Cell \rightarrow [Cell]$
$neighbors(i, j) = [(i', j') \mid i' \leftarrow [i-1..i+1], j' \leftarrow [j-1..j+1],$
                                   $(i, j) \neq (i', j'), valid\ i', valid\ j']$
  where $valid\ i = i \in [0..cN-1]$
```

With the definition of *neighbors*, we can define the number of alive neighbors. This is the length of the list *neighbors* restricted to those elements which are alive.

```
$aliveNeighbors :: Cell \rightarrow Int$
$aliveNeighbors\ cell = length([c \mid c \leftarrow neighbors\ cell, status\ c == Alive])$
```

For the time being, let us assume there is a function *status* with the signature below. In the next subsection we will define *status* as a dynamic function, since it may change its value during execution.

```
$status :: Cell \rightarrow State$
```

4.2 Dynamic semantics

In this subsection we describe the dynamic semantics of the game in terms of ASMs. To do this, we first have to store for each cell whether it is alive or dead. Therefore we use a unary dynamic function *status* initialized with state *Dead* for each cell.

```
$status :: Dynamic(Cell \rightarrow State)$
$status = initAssocs$ "status" $[(c, Dead) \mid c \leftarrow cells]$

instance $AsmTerm\ State$
```

Now we can define two rules *letDie* and *letLive* for the behavior of a cell with n alive neighbors. Both rules correspond to the last two rules (*Death* and *Birth*) at the beginning of this section. The first rule *Survival* is automatically established by ASM semantics. Since everything must be an expression in Gofer (as already stated), we can not omit the *else* branch in the following definitions.

```
$letDie :: Cell \rightarrow Int \rightarrow Rule\ ()$
$letDie\ cell\ n =$
  if $status\ cell == Alive \wedge (n < 2 \vee n > 3)$ then
    $status\ cell := Dead$
  else $skip$

$letLive :: Cell \rightarrow Int \rightarrow Rule\ ()$
$letLive\ cell\ n =$
  if $status\ cell == Dead \wedge n == 3$ then
    $status\ cell := Alive$
  else $skip$
```

It remains to execute both rules for all possible cells. For this purpose we use the *forall* rule of AsmGofer.

```
gameOfLife :: Rule ()
gameOfLife =
  forall cell ← cells do
    let n = aliveNeighbors cell
    letDie cell n
    letLive cell n
```

These definitions are sufficient for the dynamic behavior of the game. We can use the command line to validate our specification. For instance, consider following scenario:

```
? status (2,3)
Dead
(66 reductions, 175 cells)
? fire1 gameOfLife
0 steps
(126277 reductions, 259144 cells, 2 garbage collections)
? status (2,3)
Dead
(66 reductions, 175 cells)
? fire1 (do status (2,3) := Alive; status (3,4) := Alive)
1 steps
(131 reductions, 352 cells)
? status (2,3)
Alive
(66 reductions, 176 cells)
? assocs status
[((0,0),Dead), ((0,1),Dead), ...,  ((2,3),Alive),...]
(386 reductions, 2182 cells)
?
```

AsmGofer reports zero execution steps in the above output for the first *fire*1 expression, because no update was performed.

Obviously, this kind of executing rules and debugging is very time consuming. Fortunately, AsmGofer extends TkGofer whose features can be used to define graphical user interfaces as we will describe in the next two sections.

5 Automatic GUI generation

A useful feature of AsmGofer is the automatic generation of a GUI. Obviously, the generated GUI is independant of the application domain, but it can be used to debug and validate a specification at early stages.

The GUI generator is written in AsmGofer itself and reads some configuration information from a file called `gui.config` which must be located in the

Figure 3 Configuration for Game of Life

```
@FUNS
status
@TERMS
status (2,3)
@RULES
gameOfLife
initField
@CONDS
False
@FILES
output          guiMain.gs
@GUI
main            gmain
title           Game of Life
@DEFAULT
history
update
```

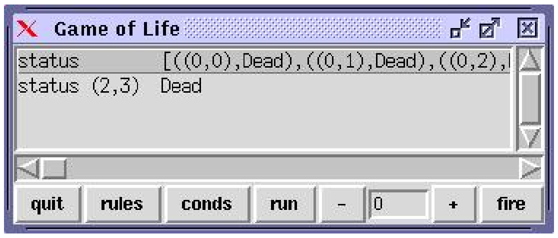
current working directory. The configuration file provides information about dynamic functions to display, expressions to display, rules the user may select in the GUI to execute, and some conditions for iterative execution of rules. With this information, the generator creates a new file containing the corresponding GUI definitions.

Figure 3 shows a configuration file for our example. The file is divided into several sections which we are now going to explain. Section `FUNS` contains dynamic function names which should be displayed in the GUI. Additionally the GUI displays expressions which are defined in the `TERMS` section. In our case we display only the expression *status* (2, 3). The GUI generator creates a list box where the user can select a rule to execute. All names in the `RULES` section are listed in that box. Obviously these names must be of type *Rule* (). We add the following rule *initField* to our definitions for the game to initialize the matrix.

```
initField :: Rule ()
initField = do
  forall c ← cells do
    if c ∈ init then
      status c := Alive
    else
      status c := Dead
where init = [(3,4),(4,4),(5,4),(4,3),(5,5)]
```

Selected rules in the list box can be executed step by step and while or until a certain condition holds. The conditions which can be selected are defined in the `CONDS` section. All expressions in this section must be of type *Bool*.

Figure 4 Generated GUI for Game of Life



```
status        [((0,0),Dead),((0,1),Dead),((0,2),
status (2,3)  Dead
```

The *output* entry in the section `FILES` denotes the file in which the generator should create the corresponding GUI definitions. Additionally, the *main* and *title* entries define the function name to start the resulting GUI and the title for the main window, respectively. In our case we can run the generated GUI with the function *gmain*.

In the `DEFAULT` section we can specify whether we want a history for the execution steps to move forward and backwards and whether the GUI should be updated during execution of a rule while or until a condition holds.

With the configuration file in Fig. 3 we can type `genGUI` in the unix shell. This command calls the GUI generator, reads the information in the configuration file and writes the file `guiMain.gs`. If the file *game.gs* contains the AsmGofer definitions introduced in the previous chapter, then we define the following project file `game.p` for Gofer.

```
game.gs
guiMain.gs
```

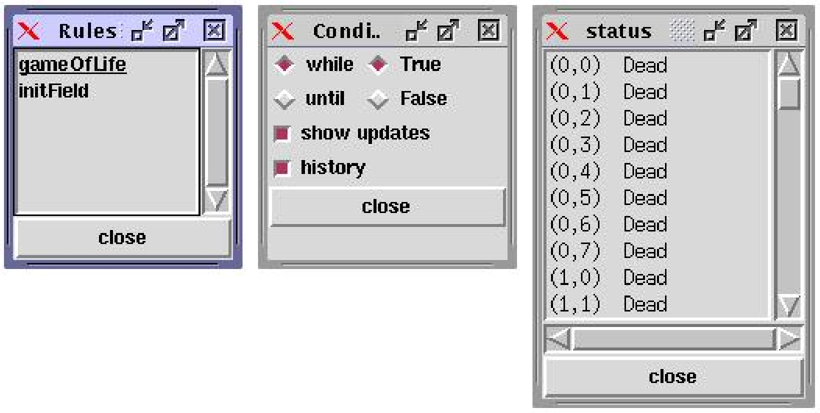
We now load this project file by typing `:p game.p` in the Gofer command line. The expression *gmain* starts the generated GUI and the main window in Fig. 4 appears. Clicking on the `rules` button displays the list box containing the rules specified in the `RULES` section of the configuration file. The same applies for the box displaying the conditions. Figure 5 shows both windows. Note that the condition *True* is always present.

The content of dynamic functions is displayed in the main window. Double clicking on such a dynamic function opens a new window which displays only the values for that dynamic function. Figure 5 also shows this window for the dynamic function *status*.

The `quit` button in Fig. 4 is self explaining. Pressing the `run` button executes the rule selected in the rule window while or until the selected condition in the condition window holds. If the check button `show updates` is enabled, then the GUI will be updated in each step during iterative execution with `run`; otherwise only after the iterative execution.

If the history button is enabled, then AsmGofer stores the complete history of execution steps. The buttons `+` and `-` move backwards and forwards in the history. Finally, the `fire` button executes one step of the selected rule.

Figure 5 Rule and condition window, Dynamic function *status*



gameOfLife
initField

[✓ while] until
[✓ True] False
[✓ show updates]
[✓ history]

(0,0)	Dead
(0,1)	Dead
(0,2)	Dead
(0,3)	Dead
(0,4)	Dead
(0,5)	Dead
(0,6)	Dead
(0,7)	Dead
(1,0)	Dead
(1,1)	Dead

6 User defined GUI

This section describes how to fire rules from a GUI. For this reason, we define a graphical user interface for Conway's *Game of Life* using the features of TkGofer. Since this section does not introduce TkGofer (for an introduction see [15]), the reader should have basic knowledge of TkGofer.

As already mentioned, the function *fire*1 transforms a rule into an IO action and is of type *Rule* () → *IO* (). On the other hand, TkGofer provides a function *liftIO* transforming an IO action into a GUI action. This implies that the following definition for *oneStep* is a GUI action which first executes one step of the rule *gameOfLife* and then updates each field in the list *fields* containing pairs of buttons and cells. GUI actions are always executed sequentially from top to bottom (see [15] for more information).

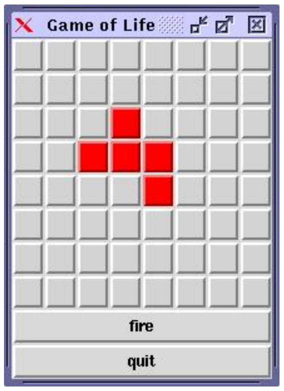
```
oneStep :: Fields → GUI ()
oneStep(fields) = do
  liftIO (fire1 gameOfLife)
  seqs [cset f (background(color c)) | (f, c) ← fields]

type Field  = (Button, Cell)
type Fields = [Field]
```

We are now going to define the remaining functions for the GUI (see Fig. 6) and we start with a function yielding a color to represent a cell depending on its status.

```
color :: Cell → String
color cell = if status cell == Dead then "lightgrey" else "red"
```

Since we do not want to initialize the dynamic function *status* by entering

Figure 6 Graphical user interface



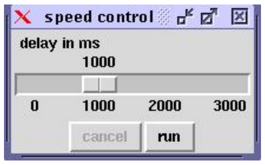
commands on the command line we define each field in the matrix in Fig. 6 in such a way, that the status of the corresponding cell toggles whenever we click on the field. Therefore we define a *toggle* action similarly to the *oneStep* action above. It first switches the status for the given cell and then updates the background color of the corresponding button in the GUI.

```
toggle :: Field → GUI ()
toggle(b, cell) = do
  liftIO (fire1(if status cell == Dead then
                  status cell := Alive
                else
                  status cell := Dead))
  cset b (background(color cell))
```

A Gofer program is usually started by a function called *main*. Hence we define *main* as creating the main window and the elements as shown in Fig. 6. All functions used in this definition are already introduced or described in [15] except *controlWindow*.

```
main :: IO ()
main = start $ do
  w      ← window [title "Game of Life"]
  q      ← button [text "quit", command quit] w
  bfire  ← button [text "fire"] w
  fields ← binds [button[background(color c), font "courier"] w
                  | c ← cells]
  let fields′ = zip fields cells
  seqs [cset f (command(toggle(f, c))) | (f, c) ← fields′]
  cset bfire (command(oneStep fields′))
  pack (matrix cN fields ^^ fillX bfire ^^ fillX q)
  controlWindow(fields′)
```

Figure 7 Control execution



The function *controlWindow* creates an additional window (see Fig. 7) to control the execution. This window contains a `run` button to start automatic execution (iterative execution of *oneStep*) and a `cancel` button to stop it.

```
controlWindow :: Fields → GUI ()
controlWindow(fields) = do
  w  ← window [title "speed control"]
  t  ← timer [initValue 1000, active False, command(oneStep(fields))]
  bc ← button [text "cancel", active False] w
  br ← button [text "run"] w
  s  ← hscale [scaleRange (0, 3000), tickInterval 1000,
                 text "delay in ms", height 200, initValue 1000] w
  pack (flexible(flexible s^^bc << br))
  cset s  (command( do v ← getValue s; setValue t v))
  cset br (command(animation(t, s, bc, br)))
  cset bc (command( do cset t (active False)
                       cset br (active True)
                       cset bc (active False)))
```

The definition creates a new window with a timer object, two buttons `cancel` and `run`, and a scaler. Whenever the timer expires, the *oneStep* action is executed. In our definition, the timer expires every second (assuming the timer is active).

When clicking on the `run` button, the action *animation* is executed which disables the `run` button, enables the `cancel` button, and activates the timer. This implies that then *oneStep* is executed every second.

```
animation :: (Timer, Scale, Button, Button) → GUI ()
animation(t, s, bc, br) = do
  cset br (active False)
  cset bc (active True)
  cset t  (active True)
```

Loading the source code (downloadable at [12]) in AsmGofer and typing *main* starts the GUI. We can initialize the dynamic function *status* by clicking on the corresponding fields to establish the glider pattern as shown in Fig. 6. The `fire` button in the main window executes one step of the *gameOfLife* rule.

On the other hand, the `run` button in the control window execues *gameOfLife* iteratively. Pressing the `quit` button terminates the GUI and the interpreter is ready for further commands.

7 Summary

AsmGofer is a tool which can be used for a wide spectrum of applications. Since it is an interpreter, it is usually too slow for performance critical applications, but it is rather suitable for building prototypes. AsmGofer is supported by other tools. Since Gofer is similar to Haskell, for instance, the lexer and parser tools for Haskell can be used for Gofer, too.

AsmGofer is a conservative extension of TkGofer and Gofer. This implies that all features of Gofer and TkGofer can be used in AsmGofer. In particular, TkGofer allows to rapidly build graphical user interfaces to control the ASM definitions.

AsmGofer has been used successfully for instance for the ASM models in *Java and the Java Virtual Machine* [13] where the static and dynamic semantics of Java are defined in AsmGofer. This includes parsing and type checking of Java programs as well as a GUI which shows the evaluation of a Java program step by step.

Acknowledgments. We thank Egon Börger, Vincenzo Gervasi, and Peter Päppinghaus for many comments on this introduction.

References

[1] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[2] E. Börger and J. Huggins. Abstract State Machines 1988–1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105–127, February 1998. Updated bibliography available at http://www.eecs.umich.edu/gasm.

[3] E. Börger, E. Riccobene, and J. Schmid. Capturing requirements by Abstract State Machines: The Light Control case study. *Journal of Universal Computer Science*, 6(7), 2000.

[4] E. Börger and J. Schmid. Composition and submachine concepts. In P. G. Clote and H. Schwichtenberg, editors, *Computer Science Logic (CSL 2000)*, number 1862 in Lecture Notes in Computer Science, pages 41–60. Springer-Verlag, 2000.

[5] A. Cavarra and E. Riccobene. Simulating UML statecharts. In R. Moreno-Dìaz and A. Quesanda-Arencibia, editors, *Formal Methods and Tools for Computer Sciene, Eurocast*, 2001. Extended Abstract.

[6] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[7] C. Hall, K. Hammond, S. P. Jones, and P. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[8] M. P. Jones. An introduction to Gofer, 1991. Available in the Gofer distribution at http://www.cse.ogi.edu/~mpj/goferarc/index.html.

[9] M. P. Jones. Gofer – Functional programming environment, 1994. Web page at http://www.cse.ogi.edu/~mpj/goferarc/index.html.

[10] S. P. Jones and P. Wadler. Imperative functional programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, 1993.

[11] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3), 1994.

[12] J. Schmid. Executing ASM specifications with AsmGofer. Web pages at: http://www.tydo.de/AsmGofer, 1999.

[13] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification and Validation*. Springer-Verlag, 2001. In press, see web pages at http://www.inf.ethz.ch/~jbook.

[14] S. Thompson. *Haskell – The Craft of Function Programming*. Addison-Wesley, 1999. second edition.

[15] T. Vullinghs, W. Schulte, and T. Schwinn. An introduction to Tk-Gofer, 1996. Web pages at http://pllab.kaist.ac.kr/seminar/haha/tkgofer2.0-html/user.html.

[16] P. Wadler. The essence of functional programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages*, pages 1–14, 1992.

[17] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, pages 24–52. Springer-Verlag, 1995.

[18] J. Walton. Conway's Game of Life, 2000. Web pages at http://www.reed.edu/~jwalton.