Electronic Notes in Theoretical Computer Science 86 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume86.html 17 pages

Towards Translating Embedded Curry to C [1]

Michael Hanus Klaus Höppner Frank Huch

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh,klh,fhu}@informatik.uni-kiel.de

Abstract

This paper deals with a framework to program autonomous robots in the declarative multi-paradigm language Curry. Our goal is to apply a high-level declarative programming language for the programming of embedded systems. For this purpose, we use a specialization of Curry called Embedded Curry. We show the basic ideas of our framework and an implementation that translates Embedded Curry programs into C.

Key words: functional logic programming, process-oriented programming, embedded systems, domain-specific languages

1 Motivation

Although the advantage of declarative programming languages (e.g., functional, logic, or functional logic languages) for a high-level implementation of software systems is well known, the impact of such languages on many real world applications is quite limited. One reason for this might be the fact that many real-world applications have not only a logical (declarative) component but also demand an appropriate modeling of the dynamic behavior of a system. For instance, embedded systems become more important applications in our daily life than traditional software systems on general purpose computers, but the reactive nature of such systems seems to make it fairly difficult to use declarative languages for their implementation. We believe that this is only partially true since there are many approaches to extend declarative languages with features for reactive programming. In this paper we try to apply one such approach, the extension of the declarative multi-paradigm language Curry [12,16] with process-oriented features [6,7], to the programming of concrete embedded systems.

[1] This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2 and by the DAAD/NSF under grant INT-9981317.

©2003 Published by Elsevier Science B. V.

Fig. 1. The RCX, the "heart" of a Mindstorms robot

The embedded systems we consider in this paper are Lego Mindstorms robots.[2] Although these are toys intended to introduce children to the construction and programming of robots, they have all typical characteristics of embedded systems. They act autonomously, i.e., without any connection to a powerful host computer, have a limited amount of memory (32 kilobytes for operating system and application programs) and a specialized processor (Hitachi H8 16 MHz 8-bit microcontroller) which is not powerful compared to current general purpose computers. In order to explain the examples in this paper, we briefly survey the structure of these robots.

The Robotics Invention System (RIS) is a kit to build various kinds of robots. Its heart is the Robotic Command Explorer (RCX, see Fig. 1) containing a microprocessor, ROM, and RAM. To react to the external world, the RCX contains three input ports to which various kinds of sensors (e.g., touch, light, temperature, rotation) can be connected. To influence the external world, the RCX has three output ports for connecting actuators (e.g., motors, lamps). Programs for the RCX are usually developed on standard host computers (PCs, workstations) and cross-compiled into code for the RCX.

The RIS is distributed with a simple visual programming language (RCX code) to simplify program development for children. However, the language is quite limited and, therefore, various attempts have been made to replace the standard program development environment by more advanced systems. A popular representative of these systems is based on replacing the standard RCX firmware by a new operating system, *brickOS*,[3] and writing programs in C with specific libraries and a variant of the compiler `gcc` with a special back end for the RCX controller. The resulting programs are quite efficient and provide full access to the RCX's capabilities.

In this paper we will use the declarative multi-paradigm programming language Curry with synchronization and process-oriented features to program the RCX. The language Curry [12,16] can be considered as a general purpose declarative programming language since it combines in a seamless way functional, logic, constraint, and concurrent programming paradigms. In order to use it for reactive programming tasks as well, different extensions have been proposed. [13] contains a proposal to extend Curry with a concept of

[2] http://mindstorms.lego.com Note that these names are registered trademarks although we do not put trademark symbols at every occurrence of them.
[3] http://www.brickos.sourceforge.net

ports (similar concepts exist also for other languages, like Erlang [4], Oz [19], etc) in order to support the high-level implementation of distributed systems. These ideas have been applied in [6] to implement a domain-specific language for process-oriented programming, inspired by the proposal in [7] to combine processes with declarative programming. The target of the latter is the application of Curry for the implementation of reactive and embedded systems. In [15] we have applied this framework to a concrete embedded system, the Mindstorms robots described above, together with a simulator. In this paper we present a compiler for a subset of this framework.

The paper is structured as follows. In the next section we sketch the necessary features of Curry. Section 3 presents an example for a Mindstorms robot and shows how to program it in Embedded Curry. Then we survey the possibilities to translate Embedded Curry programs to C in Section 4 and conclude in Section 5 with a discussion of related work.

2 Curry

In this section we survey the elements of Curry which are necessary to understand the examples in this paper. More details about Curry's computation model and a complete description of all language features can be found in [12,16].

Curry is a multi-paradigm declarative language combining in a seamless way features from functional, logic, and concurrent programming and supports programming-in-the-large with specific features (types, modules, encapsulated search). From a syntactic point of view, a Curry program is a functional program [4] extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Thus, a Curry program consists of the definition of data types and functions. Functions are evaluated lazily. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. However, this feature will not be used in this paper.

Example 2.1 The following Curry program defines the data types of Boolean values, the polymorphic type `Maybe`, and a type `SensorMsg` (first three lines). Furthermore, it defined a test and a selector function for `Maybe`:

```
data Bool      = True    | False
data Maybe a   = Nothing | Just a
data SensorMsg = Light Int

fromJust :: Maybe a -> a
```

[4] Curry has a Haskell-like syntax [18], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

```
fromJust (Just x) = x

isJust :: Maybe a -> Bool
isJust Nothing  = False
isJust (Just _) = True
```

The data type declarations introduce `True` and `False` as constants of type `Bool`, `Nothing` (no value) and `Just` (some value) as the constructors for `Maybe` (`a` is a type variable ranging over all types), and a constructor `Light` with an integer argument for the data type `SensorMsg`.

The (optional) type declarations ("`::`") of the functions `fromJust` and `isJust` specify that they take a `Maybe`-value as input and produce a Boolean value or a value of the (unspecified) type `a`. [5]

The operational semantics of Curry, described in detail in [12,16], is based on an optimal evaluation strategy [2] and can be considered as a conservative extension of lazy functional programming and (concurrent) logic programming.

FlatCurry is an intermediate language that can be used as a common interface for connecting different tools for Curry programs or programs written in other (functional logic) declarative languages (e.g., Toy). In FlatCurry, all functions are defined at top level (i.e., local function declarations in source programs are globalized by lambda lifting). Furthermore, the pattern matching strategy is made explicit by the use of case expressions. Thus, a FlatCurry program basically consists of a sequence of data type declarations and a sequence of function definitions. Current Curry implementations like PAKCS [14] use FlatCurry as intermediate language so that the front end can be used with different back ends. The FlatCurry representation of the function `isJust` from Example 2.1 is the following:

```
isJust :: Maybe a -> Bool
isJust v0 = case v0 of
     Nothing -> False
     Just v1 -> True
```

We use FlatCurry as the source language for our compiler.

3 Programming Robots in Embedded Curry

In this section we survey the framework for programming autonomous robots in Curry as proposed in [15]. In this framework we separate the entire programming task into two parts. To control and evaluate the connected sensors of the RCX, we use a synchronous component which generates messages for relevant sensor events (e.g., certain values are reached or exceeded, a value

[5] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

has changed etc.). An Embedded Curry program contains a specification that describes the connected sensors of the robot and the kind of messages that are generated for certain sensor events. The description of the actions to be executed in reaction to the sensor events are described in an asynchronous manner as a *process system*. A process system consists of a set of processes (p_1, p_2,...), a global state and mailbox for sensor messages. The behavior of a process is specified by

- a *pattern matching/condition* (on mailbox and state),
- a sequence of *actions* (to be performed when the condition is satisfied and the process is selected for execution), and
- a *process expression* describing the further activities after executing the actions.

A process can be activated depending on the conditions on the global state and mailbox. If a process is activated (e.g., because a particular message arrives in the mailbox), it performs its actions and then the execution continues with the process expression. The check of the condition and the execution of the actions are performed as one atomic step. Process expressions are constructed similarly to process algebras [8] and defined by the following grammar:

p	::=	`Terminate`	successful termination
	|	`Proc (p` $t_1 \ldots t_n$`)`	run process `p` with parameters $t_1 \ldots t_n$
	|	p_1 `>>>` p_2	sequential composition
	|	p_1 `<|>` p_2	parallel composition
	|	p_1 `<+>` p_2	nondeterministic choice

In order to specify processes in Curry following the ideas above, there are data types to define the structure of actions (`Action` *inmsg outmsg state*) and processes (`ProcExp` *inmsg outmsg state*). Furthermore, we define a *guarded process* as a pair of a list of actions and a process term:

```
data GuardedProc inmsg outmsg state =
                 GuardedProc [Action inmsg outmsg state]
                             (ProcExp inmsg outmsg state)
```

For the sake of readability, we define an infix operator to construct guarded processes:

```
acts |> pexp = GuardedProc acts pexp
```

In order to exploit the language features of Curry for the specification of process systems, we consider a *process specification* as a mapping which assigns to each mailbox (list of incoming messages) and global state a guarded process (similarly to Haskell, a `type` definition introduces a type synonym in Curry):

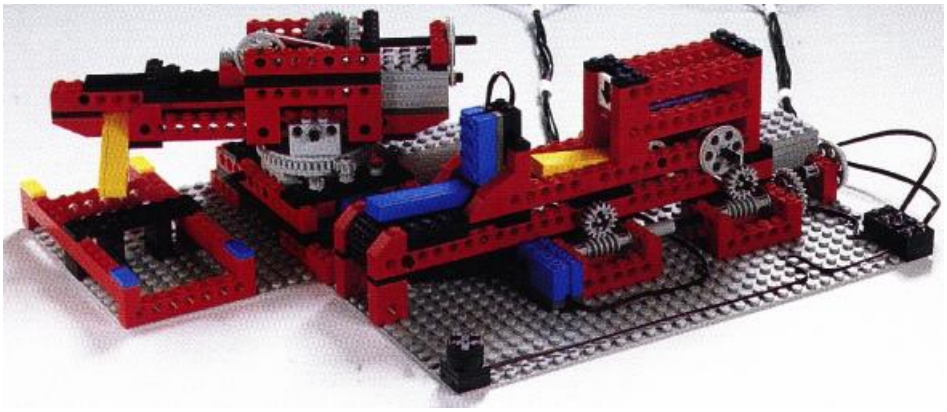

Fig. 2. Example of a robot that sorts bricks

```
type Process inmsg outmsg state =
                [inmsg] -> state -> GuardedProc inmsg outmsg state
```

This definition has the advantage that one can use standard function definitions by pattern matching for the specification of processes, i.e., one can define the behavior of a process `ps` with parameters $x_1, \ldots, x_n$ in the following form:

$$\begin{array}{l} ps\ x_1 \ldots x_n\ mailbox\ state \\ \qquad \texttt{|}\ <\ condition\ on\ x_1, \ldots, x_n,\ mailbox,\ state\ > \\ \qquad \texttt{=}\ \texttt{[}actions\texttt{]}\ \texttt{|>}\ process\ expression \end{array} \tag{1}$$

Thus, Embedded Curry is Curry plus a library containing the type definitions sketched above and an interpreter for the processes according to the operational semantics [15]. This paper presents a compiler for a subset of Embedded Curry.

As a concrete example, we want to use this framework to program a robot that sorts bricks depending on their color (yellow and blue) into two bins. The robot consists of two more or less independent parts, a robot arm that sorts the bricks into the bins and a conveyor belt that moves the bricks from a storage to the pick-up area of the arm. On its way, the conveyor belt passes a light sensor. This sensor has two purposes: it detects when the next brick is coming and it recognizes the different colors of the bricks. Fig. 2 shows a picture of this robot. In the following we will call it "sorter".

The synchronous component for the sorter has to control the light sensor which is connected to the sensor port `In_2`. When a brick passes the light sensor, the reflected light of the brick will increase the brightness measured by the sensor. A brick is recognized if the brightness value exceeds the threshold of a blue brick th_{blue} (yellow bricks are even brighter). The specification of the synchronous component is a list of pairs, mapping a sensor port to a sensor specification. This specification is stored in the definition of a distinguished constant with name `sensors`:

```
sensors = [(In_2, LightSensor [OverThresh th_blue Light])]
```

`OverThresh` states that a message should be sent if the brightness exceeds the given value. The second argument of `OverThresh` is a function which maps a brightness value (an `Int`) to a message. Here we use the constructor `Light` from Example 2.1.

Our implementation divides the sorter into two independent processes as well: the conveyor belt and the robotic arm. The communication between these two processes is handled through the global state. When a brick reaches the pick-up area, the belt process changes the global state to `BlueBrick` or `YellowBrick`, respectively, to inform the arm process of the detected brick. After the robotic arm has grasped the brick, the arm process changes the global state back to `Empty` to indicate that the pick-up area is empty again and the conveyor belt can be restarted.

After starting the conveyor belt, the belt process waits until the light sensor detects a brick. This event is indicated by a message `(Light` br`)` in the mailbox of the process system (br is the brightness value of the brick). Waiting for this message can be expressed by pattern matching on the mailbox. This means that the process will only be activated when the first message in the mailbox is of the form `(Light` i`)`:

```
waitBrickSpotted ((Light br):_) _ =
  [Deq (Light br)] |> wait t_end >>>
                      atomic [Send (MotorDir Out_C Off),
                              Set (brickType br)] >>>
                      Proc transportBrick
```

Since messages are not automatically deleted, we have to delete the message `(Light` br`)` explicitly after receiving it to prevent from multiple reactions on the same message. Deletion of messages in the mailbox is done by the action `(Deq` msg`)`. Since it is performed in the initial action sequence, the matching on the mailbox and deletion of the message are performed in one atomic step.

When a brick is detected, it has not reached the pick-up area yet. Hence, the process waits for a period t_{end} of time until it stops the belt. This can be done by a call of the predefined process expression `(wait` t`)` that suspends for t milliseconds. Then the action `(Send` cmd`)` is used to send a command cmd to the actuators of the robot. The RCX has three ports for actuators: `Out_A`, `Out_B` and `Out_C`. The motor that controls the conveyor belt is connected to port `Out_C` and the command `(MotorDir Out_C Off)` will stop it. To execute a list of actions inside a process expression, one can use the function `atomic` which is implemented as follows:

```
atomic actions = Proc (\_ _ -> actions |> Terminate)
```

To inform the arm process of the brick in the pick-up area, we change the global state to `BlueBrick` or `YellowBrick`. This is done by the action `(Set` e`)` that sets the global state to e. The function `brickType` determines the values `BlueBrick` or `YellowBrick` out of its brightness value. Because the conveyor belt should only be restarted when the pick-up area is empty, we have to wait

until the global state changes back to `Empty`. This behavior can be expressed by a second process specification that is activated by pattern matching on the global state, starts the belt with the command `(MotorDir Out_C Fwd)` and then calls the first specification:

```
transportBrick _ Empty =
  [Send (MotorDir Out_C Fwd)] |> Proc waitBrickSpotted
```

The arm process is equally simple: the arm stays over the pick-up area until there is a brick ready to be picked up, i.e., the global state changes to either `BlueBrick` or `YellowBrick`. Then it closes the gripper and sets the global state back to `Empty` (this allows the conveyor belt to restart). Finally, it calls the process `putBrick` with the amount of time it takes to turn to the blue or the yellow bin. These times are provided by the function `brickTurnTime`:

```
sortBrick _ brickKind | brickKind /= Empty =
  [] |> closeGripper >>>
        atomic [Set Empty] >>>
        Proc (putBrick (brickTurnTime brickKind))
```

The function `closeGripper` defines a process expression that closes the gripper of the robotic arm.

The process `putBrick` has a parameter `time` that specifies the amount of time it takes to turn to the correct bin. It starts the motor that turns the arm (`Out_A`) and waits for `time` milliseconds before it turns the motor off. The arm should now be placed above the bin and the gripper can be opened to drop the brick. Then the arm turns back for the same amount of time to the pick-up area and restarts the arm process:

```
putBrick time _ _ =
  [Send (MotorDir Out_A Rev)] |>
          wait time >>>
          atomic [Send (MotorDir Out_A Off)] >>>
          openGripper >>>
          atomic [Send (MotorDir Out_A Fwd)] >>>
          wait time >>>
          atomic [Send (MotorDir Out_A Off)] >>>
          Proc sortBrick
```

These two processes can be combined to the whole system in two different ways: sequentially, but then the conveyor belt will not restart until the arm is back at the pick-up area, or, by using two parallel processes. Using the latter, the conveyor belt can be restarted immediately after the brick has been picked up by the arm:

```
go _ _ = [Set Empty] |> Proc transportBrick <|> Proc sortBrick
```

As a result the bricks are sorted faster.

In this example, we use time periods to determine when the bins and the pick-up area are reached. This is the simplest implementation, but different battery charge levels result in different motor speeds so that the time the arm takes to turn to a certain position can differ from one use to another. In most cases it is much better to use rotation sensors to determine the progress of a movement. This can easily be added to the implementation by replacing the wait expressions by processes listening for messages from the rotation sensors. In our current implementation of the sorter, we use two rotation sensors to determine the position of the arm and when a brick, detected by the light sensor, reaches the pick up area. For simplicity of the example, we do not present these details here.

In this configuration the sorter uses all three sensors and all three motor ports of the RCX. This means that the sorter is already one of the most complex examples that are possible for Lego mindstorms robots. However, this is not an unrealistic example since real world embedded systems usually also have very restricted sensor suits and do not control very complex systems by themselves.

4 Translation

Our goal is to use Embedded Curry to control Lego Mindstorms Robots. Due to the (speed and time) limitations of the RCX, a simple approach, like porting a complete Curry implementation (e.g., PAKCS [14]) to the RCX, will not work. This contrasts with [17] where a functional robot control language is proposed which is executed on top of Haskell running on a powerful Linux system. Our previous implementation of the process extension of Curry [6] is based on an interpreter (written in Curry) for the process expressions following the operational semantics of the process algebra [6,7,15]. This implementation is fairly simple (approximately 200 lines of code) but causes too much overhead. Thus, a direct compilation of the process specifications into more primitive code is necessary. As mentioned in Section 1, one of the more flexible operating systems for the RCX is brickOS. It is a POSIX-like operating system offering an appropriate infrastructure like multi-threading, semaphores for synchronization etc. Therefore, using C as an intermediate language and the brickOS compiler as back end is appropriate to implement a Curry compiler for the Lego Mindstorms. Nevertheless, many optimizations are required to map the operational semantics of Curry into features available in brickOS. However, we do not intend to cover all features of Curry (e.g., constraint solving) with our compiler since they are not necessary for our applications on the limited target architectures.

Our current implementation is restricted to a first-order functional subset of Curry with nonrecursive data declarations. Furthermore, we ignore laziness and generate strict target code. These restrictions were made for the following reasons: lazy evaluation (as well as higher-order partial applications) are

memory consumptive. Furthermore, dynamic data structures need garbage collection which can be critical to guarantee constant reaction times. Finally, our experiences with the programming of Mindstorms shows that these restrictions seem to be acceptable in practice although we intend to extend our implementation in the future.

We use FlatCurry as the source code for our translation. To implement the parallel execution of processes, we use processes of the operating system (we will call them OS processes in future). An extra OS process is used for the synchronous component. The implementation of this component can be generated out of the sensor specification given in the Embedded Curry program.

To describe our translation, we start with algebraic data types which are declared as:

$$\texttt{data}\ A\ \texttt{=}\ C_0\ A_{0,0}\ \ldots\ A_{0,m_0}\ \texttt{|}\ \ldots\ \texttt{|}\ C_n\ A_{n,0}\ \ldots\ A_{n,m_n}$$

One can easily map such a declaration to structures of the target language C (i.e., records). The structure consists of an enumeration of the constructors (C_0,...,C_n) and a union of all argument structures (structures that can store all the arguments of a certain constructor).

Function declarations of Curry are mapped to C function declarations. Because in FlatCurry all conditions have been resolved into `if`-expressions and pattern matching has been made explicit by the use of case expressions, this can be easily achieved. Case expressions in FlatCurry contain no nested patterns, i.e., all patterns are of the form $C\ x_1 \ldots x_n$. Hence, they can be translated to switch statements over the enumeration of the constructors.

4.1 Global State and Mailbox

The global state and mailbox parameters of an Embedded Curry program have a special interpretation that differs from standard Curry parameters. For each check of the guards (i.e., pattern matching and rule conditions) of a process, the current values of the global state and mailbox are used (and not the values at the time of the creation of that process). This behavior can be implemented in C by using global variables for the mailbox and the global state. To guarantee mutual exclusion of the guard checking for all processes, we use a global semaphore. Before checking the guards, the processes waits for this semaphore and releases it afterwards. In the case that one of the guards is satisfied, the semaphore is released only after performing the list of actions of the selected process. The actions are translated accordingly to the operational semantics in [6,15]. Note that all actions are performed within an initial phase of a process (see definition of `action`). Hence, in the translation all modifications of the state and the mailbox are synchronized by the global semaphore as well.

In Embedded Curry the mailbox is a list of messages. Since the mailbox is extended by the synchronous sensor component, we implement the mailbox

as a queue with corresponding operations. Therefore, our translation contains an abstract data type (ADT) definition for the mailbox. For the sake of simplicity, we use a queue implementation of fixed size at the moment but this implementation can be easily replaced by a dynamic queue. To allow pattern matching on the mailbox, one has to implement it similarly to pattern matching on algebraic data types using the selector functions of the ADT mailbox. Although the mailbox can be seen as a recursive data structure, all modifications of the mailbox are made explicit by actions. Hence, no garbage arises during the execution.

4.2 Process Specifications

In Embedded Curry a process specification is a function that maps a mailbox and a state to a pair of actions and process expressions, i.e., it can be easily identified by its type. To generate an executable robot control program, these function declarations are compiled in a different way than other function declarations. Their guards (i.e., pattern matching and rule conditions) have to be checked continously until one of them is satisfied.

We translate a process specification as shown in (1) to the following general form:

```
void ps (x1,...,xn) {
  while (1) {
    lockState();
    <check guards sequentially. If satisfied, execute process>
    unlockState(); suspend();}
}
```

Since the global state and mailbox are global variables, we do not need to pass them as parameters. The arguments x_i represent the local state of the process. To implement the continuous checking of the guards until one guard is satisfied, we use an infinite loop. Inside the loop we first lock the state (by setting the semaphore) in order to guarantee mutual exclusion on the global state and mailbox. Then we sequentially check all guards of the process specification. If none is satisfied, we unlock the state (by releasing the semaphore) and suspend this process until a change to the global state or mailbox occurs.

If one of the guards is satisfied, the corresponding code is executed and the procedure *ps* is terminated by `return`. This code includes an unlocking of the state after the actions are performed.

Example 4.1 The process specification `transportBrick` has the following FlatCurry representation:

```
transportBrick v0 v1 = case v1 of
    Empty -> [Send (MotorDir Out_C Fwd)] |>
             (Proc waitBrickSpotted)
```

This code is translated to the C procedure:

```
void transportBrick () {
  while (1) {lockState();
             switch ((state).kind) {
               case Empty : motor_c_dir(fwd); unlockState();
                            waitBrickSpotted(); return;
               default : unlockState(); suspend();}
            }
}
```

In this example the guard consists of a pattern matching on the global state. If the global state is `Empty`, then the motor connected to port `Out_C` is started and then the process `waitBrickSpotted` is called.

4.3 Tail Call Optimization

Reactive programs, as in embedded systems, are usually non-terminating. In a declarative programming language, non-terminating programs are implemented by recursion. At this point a problem arises because the GNU C compiler we use[6] has no general tail call optimization. If a program contains non-terminating recursion, the execution of the program would result in a stack overflow. However, this problem is only relevant for process specifications. All other function calls should terminate. A workaround for this problem is to execute a tail call of a process outside the process specification. This means that the process specification is terminated before the new process is called, i.e., the stack frame of the old process specification is deleted before the stack frame of the new process is put on the stack.

We implement this by using a small interpreter to execute all process calls of a process system. The information of a process call (i.e., the name of the called process and the parameters of the call) are stored in a data structure and passed to the interpreter procedure. The call information of the tail calls is the result of a process specification. The interpreter stores this information and executes the next call.

Our sorter example consists of five process specifications: `go`, `sortBrick`, `putBrick`, `transportBrick`, and `waitBrickSpotted`. In the current implementation this specification is translated into the following interpreter procedure:

```
int process_system (ProcType np) {
  while (np.next != p_Terminate) {
    switch (np.next) {
      case p_go : np = go(); break;
      case p_transportBrick : np = transportBrick(); break;
      case p_waitBrickSpotted : np = waitBrickSpotted();
                                break;
```

[6] gcc version 2.95.3

```
        case p_sortBrick : np = sortBrick(); break;
        case p_putBrick : np = putBrick(np.arg.putBrick.arg0);
                          break;
      }
    }
  }
```

The parameter `np` is used to store the information about the next process call. The attribute `next` is the name of the called process and `arg` is a union of all possible parameter combinations. The prefix `p_` is used to prevent name conflicts between functions and data type constructors. Termination of a process is handled as a call of a process `Terminate` and results in a termination of the interpreter procedure.

Example 4.2 The process specification `transportBrick` from Example 4.1 now returns the information of the tail call of the process `waitBrickSpotted` as result, instead of calling it directly.

```
ProcType transportBrick () {
  while (1) {lockState();
             switch ((state).kind) {
               case Empty : motor_c_dir(fwd); unlockState();
                            return cNext_waitBrickSpotted();
               default : unlockState(); suspend();}
             }
}
```

The function `cNext_waitBrickSpotted` creates a data structure (of the type `ProcType`) with the information for the call of the process `waitBrickSpotted`.

On the other hand, not all calls of processes must be located in tail positions. Non-tail calls are handled in the following way. Consider a call inside a sequential process expression:

`Proc` $(ps\ e_1\ \ldots\ e_n)$ `>>>` pe

Here we can use the C call stack by the translation:

`process_system(cNext_`ps`(`e_1`,...,`e_n`));` pe'

`cNext_`ps is a function that creates the data structure with the information for the call of the process ps. pe' is the remaining translation of pe.

For process calls in parallel expressions, we create a new OS process for one of the processes, the other can run in the actual OS process. A problem arises because it is not possible to pass arbitrary parameters to a new OS process. This can be solved by supplying a global variable `nextProcess` were the call information is stored. Thus, a parallel expression containing at least one process call

`Proc` $(ps\ e_1\ \ldots\ e_n)$ `<|>` pe

is translated to:

```
nextProcess = cNext_ps(e1,...,en);
execi(&process_system,0,NULL,PRIO,STACK_SIZE);
pe'
```

`execi` forks a new OS process that will execute the procedure `process_system` without parameters. The procedure then retrieves the call information (i.e., parameters) from `nextProcess`.

In our implementation we use this global variable also for all other process calls (i.e., sequential and tail recursive). One global variable `nextProcess` is sufficient for the whole program because it is only needed for the short time of the initialization of the new process. Since multiple processes could be called in parallel, we ensure mutual exclusion on the variable `nextProcess` by an additional semaphore. This means that all process calls are executed sequentially. Only after the call is executed and the new process has been initialized, another OS process can use `nextProcess` for another process call. However, the execution of a process call takes only a small amount of time, so it will cause no problems for the reactivity of the whole system.

Example 4.3 The process specification `go` of our sorter example has the following FlatCurry representation:

```
go v0 v1 =
   [Set Empty] |> (Proc transportBrick <|> Proc sortBrick)
```

This code is translated to the C procedure:

```
ProcType go () {
  unlockNext();
  lockState(); state = Empty; unlockState();
  lockNext(); nextProcess = cNext_sortBrick();
  execi(&process_system,0,NULL,PRIO,STACK_SIZE);
  lockNext();
  return cNext_transportBrick();
}
```

Shortly before the information for the call of `sortBrick` is stored in the global variable `nextProcess`, the semaphore protecting this global variable is set (`lockNext`). Then the new OS process for the execution of `sortBrick` is created with `execi`. No other OS process will then be able to use `nextProcess` until the initialization of `sortBrick` is finished. The tail call of the process `transportBrick` uses the same mechanism by locking `nextProcess` immediately before returning to the C function `process_system`, where the information for the tail call is stored in `nextProcess`.

Because all process calls are protected by the locking of `nextProcess`, each process function unlocks it before it enters the code that represents the execution of the process.

Another approach to solve this problem is to combine all process specifications into one C function. Inside this function it is then possible to realize tail calls by jumps without any overhead. We have not implemented this approach yet, but want to compare both approaches to use the more efficient for our implementation.

5 Conclusions

We have implemented a compiler for Embedded Curry based on the ideas described above. The compiler translates Embedded Curry programs into C programs which can then be translated into executable code for the RCX. To get an impression of the code size (which is important for embedded systems), the size of the Curry sorter program is 2947 bytes, the FlatCurry code has 24261 bytes, the generated C code has 10407 bytes, and the RCX binary code has 2986 bytes. Our previous Curry interpreter of Embedded Curry produces executables of more than 1 MB (by compilation into Prolog [3]). Since we use C as intermediate language, it is also possible to generate executable code for nearly every platform that has a C compiler.

At the moment we do not have a time analysis that would allow us to guarantee certain reaction times. We have restricted our subset to nonrecursive data structures, because the otherwise necessary garbage collection could interfere with the reaction times of the system. The reactivity depends on the fairness of the scheduling of the brickOS. Nevertheless, our experience showed that the reactivity of the resulting programs is sufficient for our applications.

For embedded system programming, synchronous languages like Esterel [5] or Lustre [9] are often used. Thus, one can also apply such languages to program embedded systems like the Lego Mindstorms robots. Actually, there already exist compilers for those languages into C. Hence, one can use the brickOS compiler to produce RCX code for these languages as well. The translation of the synchronous languages normally produces sequential code by synthesizing the control structure of the object code in the form of an extended finite automaton [10]. This is a major drawback since one does not have much control on the size of the generated code. In some cases only slight modifications in a robot specification can result in a big increase in the size of the generated code. Due to state explosion this could result in too large programs for the limited amount of memory in the RCX. Another proposal to use high-level languages for programming embedded or process-oriented systems is [17]. In contrast to our approach, they propose a functional robot control language which is executed on top of Haskell running on a powerful Linux system. The domain-specific language proposed in [11] is comparable to our language with similar results.

For future work, we want to enlarge the subset of Curry translatable by our compiler. First, we will investigate possibilities to compile higher-order functions by translating to first-order using partial evaluation [1] or by means

of pointers. Further interesting fields of research could be the high-level specification of Embedded Curry programs. This could be integrated in a tool also supporting debugging, testing and formal verification.

References

[1] E. Albert, M. Hanus, and G. Vidal. A practical partial evaluator for a multi-paradigm declarative language. *Journal of Functional and Logic Programming*, 2002(1), 2002.

[2] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[3] S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.

[4] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] B. Braßel, M. Hanus, and F. Steiner. Embedding processes in a declarative programming language. In *Proc. Workshop on Programming Languages and Foundations of Programming*, pages 61–73. Aachener Informatik Berichte Nr. AIB-2001-11, RWTH Aachen, 2001.

[7] R. Echahed and W. Serwe. Combining mobile processes and declarative programming. In *Proc. of the 1st International Conference on Computation Logic (CL 2000)*, pages 300–314. Springer LNAI 1861, 2000.

[8] W. Fokkink. *Introduction to Process Algebra*. Springer, 2000.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[10] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 207–218. Springer LNCS 528, 1991.

[11] K. Hammond and G. Michaelson. Hume: a domain-specific language for real-time embedded systems. In *Proc. of GPCE '03 – Generative Programming and Component Engineering*. to appear in Springer LNCS, Sep 2003.

[12] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

[13] M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.

[14] M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. Pakcs: The portland aachen kiel curry system. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2003.

[15] M. Hanus and K. Höppner. Programming autonomous robots in Curry. *Electronic Notes in Theoretical Computer Science*, 76, 2002.

[16] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.

[17] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *Proc. of the First International Workshop on Practical Aspects of Declarative Languages*, pages 91–105. Springer LNCS 1551, 1999.

[18] S.L. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, 1999.

[19] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.