©Springer-Verlag
In *Proc. of the 12th International Workshop on Implementation of Functional Languages, IFL 2000*, Aachen.
Springer LNCS 2011, pp. 89–106, 2001

ObjectCurry: An Object-Oriented Extension of the Declarative Multi-Paradigm Language Curry

Michael Hanus[1]* and Frank Huch[2] and Philipp Niederau[2]

[1] Institut für Informatik, Christian-Albrechts-Universität zu Kiel, Germany,
`mh@informatik.uni-kiel.de`
[2] Lehrstuhl für Informatik II, RWTH Aachen, Germany
`hutch@i2.informatik.rwth-aachen.de` `phn@navigium.de`

Abstract. Curry combines the concepts of functional, logic and concurrent programming languages. Concurrent programming with ports allows the modeling of objects in Curry similar to object-oriented programming languages. In this paper, we present ObjectCurry, a conservative extension of Curry. ObjectCurry allows the direct definition of templates which play the role of classes in conventional object-oriented languages. Objects are instances of a template. An object owns a state and reacts when it receives a message—usually by sending messages to other objects or a transformation of its state. ObjectCurry also provides inheritance between templates. Furthermore, we show how programs can be translated from ObjectCurry into Curry by exploiting the concurrency and distribution features of Curry. To implement inheritance, we extend the type system of Curry, which is based on parametric polymorphism, to include subtyping for objects and messages.

1 Introduction

Curry [4, 6] is a multi-paradigm declarative language which integrates functional, logic, and concurrent programming paradigms (see [3] for a survey on integrated functional logic languages). The syntax of Curry is similar to Haskell [15], e.g., functions are defined by rules of the form "$f\, t_1 \ldots t_n$ `=` e" where f is the function to be defined, $t_1, \ldots, t_n$ are the pattern arguments, and e is an expression which replaces a function call matching the left-hand side. In addition to Haskell, local names introduced in `let` and `where` clauses can be declared as "`free`" which means that their value is unknown. Such *free* or *logical variables* in expressions supports logic programming features like partial data structures and search for solutions. Furthermore, functions in Curry can be defined by conditional equations "l `|` c `=` r" where the condition c is a *constraint* (an expression of the predefined type `Success`) which must be solved in order to apply the equation. Basic constraints are "`success`" (the always satisfiable constraint) and equational constraints of the form "e_1 `=:=` e_2" which are satisfied if both sides e_1

* This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2 and by the DAAD under the PROCOPE programme.

and e_2 are reducible to the same value (data term). More complex constraints can be constructed with the concurrent conjunction operator `&`. A non-primitive constraint like "c_1 `&` c_2" is solved by solving both constraints c_1 and c_2 concurrently. Finally, "c_1 `&>` c_2 denotes the sequential conjunction of two constraints, i.e., first the constraint c_1 is solved and, if this was successful, the constraint c_2 is evaluated.

Using both functional and logic features of Curry, it is possible to model objects with states (see Section 2) at a very low level. Therefore, we propose an extension of Curry, called ObjectCurry, which provides all standard features of object-oriented programming, like (concurrent and distributed) objects with state that can be defined by class templates and inheritance between templates.

This paper is structured as follows. In the next section, we review the modeling of concurrent objects in Curry as proposed in [5]. We present ObjectCurry in the subsequent section and show the translation of ObjectCurry programs into Curry in Sect. 4. Section 5 describes an extended type system for ObjectCurry in order to detect type errors related to inheritance at compile time before we discuss related work in Sect. 6 and conclude in Sect. 7.

2 Implementing Objects in Curry

It is well known from concurrent logic programming [16] that objects can be easily implemented as predicates processing a stream of incoming messages. The internal state of the object can be implemented as a parameter which may change in recursive calls when the message stream is processed. Since constraints play the role of predicates in Curry, we consider objects as functions with result type `Success`. These functions take the current state of the object and a stream of incoming messages as arguments. If the stream is not empty, the "object" function calls itself recursively with a new state, depending on the first element of the message stream. Thus,

$$o :: \mathit{State} \rightarrow \texttt{[}\mathit{MessageType}\texttt{]} \rightarrow \texttt{Success}$$

is the general type of an object where *State* is the type of the internal state of the object and *MessageType* is the type of messages. Usually, we define a new algebraic data type for the messages.

The following example shows a counter which understands the messages `Inc`, `Set s`, and `Get v`. Thus, we define the data type

```
data CounterMessage = Inc | Set Int | Get Int
```

The counter has an integer value as an internal state. Receiving `Inc` increments the internal state and `Set s` assigns it to a new value `s`. To get the current state of the counter as an answer, we send the message `Get v` to the object where `v` is a free logical variable. In this case the counter object binds this variable to its current state:

```
counter :: Int -> [CounterMessage] -> Success
```

```
counter eval rigid
counter x (Inc   : ms) = counter (x+1) ms
counter _ (Set s : ms) = counter s     ms
counter x (Get v : ms) = v =:= x  &  counter x ms
counter _ []           = success
```

The evaluation of the constraint "`counter 42 s`" creates a new counter object with initial value 42. Messages are sent by instantiating the variable `s`. The object terminates if the stream of incoming messages is empty. In this case the constraint is reduced to the trivial constraint `success`. For instance, the constraint

```
let s free in counter 41 s & s=:=[Inc, Get x]
```

is successfully evaluated where `x` is bound to the value `42`. The annotation

```
counter eval rigid
```

marks `counter` as a rigid function. This means that an expression "`counter x s`" can be reduced only if `s` is bound.[1]

If there is more than one process sending messages to the same counter object, it is necessary to merge the message streams from different processes into a single message stream. Doing that with a merger function causes a set of problems as discussed in [5, 8]. Therefore, Janson et al. [8] proposed the use of ports for the concurrent logic language AKL which are generalized in [5] to support distributed programming in Curry. In principle, a *port* is a constraint between a multiset and a stream which is satisfied if the multiset and the stream contain the same elements. In Curry a port is created by a constraint "`openPort p s`" where `p` and `s` are free logical variables. This constraint creates a multiset and a stream and combines them over a port. Elements can be inserted into the multiset by sending them to `p`. When a message is sent to `p`, it will automatically be added to the stream `s` in order to satisfy the port constraint. For sending a message, there is a constraint "`send m p`" where `m` is the message and `p` is a port created by `openPort`.

Using ports, we can rewrite the counter example as follows

```
openPort p s &>  counter 0 s & (send Inc p &> send (Get x) p)
```

3 ObjectCurry, an Object-Oriented Extension of Curry

Using the technique presented above is troublesome and error-prone, in particular, if the state consists of many variables, because the programmer has to

[1] In contrast to rigid functions, Curry also provides flexible functions which nondeterministically instantiate their arguments in order to allow the reduction of function calls, which provides for goal solving like in logic programming. As a default (which can be changed by `eval` annotations), constraints are flexible and all other functions are rigid.

repeat the whole state in the recursive calls. This motivated us to introduce some special syntax for defining *templates*. Templates play the role of classes in conventional object-oriented programming languages. We use the word "template" instead of class to avoid confusion between classes in an object-oriented meaning and Haskell's type classes. For instance, a *template for counter objects* can be defined in ObjectCurry as follows:

```
template Counter =
constructor
  counter init = x := init
methods
  Inc    =  x := x + 1
  Set s  =  x := s
  Get v  =  v =:= x
```

A template definition starts with the reserved keyword `template` followed by the name of the template. Similar to a data type declaration, the name of the template is its own type. The *constructor* is a function which we use to instantiate new objects. The left-hand side is constructed as in conventional function declarations. The right-hand side is a set of assignments describing the *attributes* of the object and their initial values. The assignments are consecutively written using the offside rule.

The messages which are understood by the object and the reactions to these messages are defined by *methods*. Messages are defined similarly as the constructor. The left-hand side of a method declaration consists of the name of the method followed by a list of patterns as in a function declaration and describes the signature of a message with the same name as the method. The right-hand side describes the behavior of the object in response to receiving a message. A reaction can be a transformation of the internal state of the object. The transformation of a state can be expressed by a set A of assignments of the form "v `:=` e". If the tuple $(v'_1, \ldots, v'_n)$ is the current state of the object where the template has the attributes $v_1, \ldots, v_n$, A specifies the state transformation $(v'_1, \ldots, v'_n) \mapsto (v''_1, \ldots, v''_n)$ defined by

$$v''_i = \begin{cases} e_i & \text{if } v_i \mathrel{\texttt{:=}} e_i \in A \\ v'_i & \text{otherwise} \end{cases}$$

Additionally, the right-hand side of a method can also include constraints, i.e., expressions of the type `Success`, because constraints offer further possibilities to express reactions, e.g., equational constraints are used to yield an answer by binding a logical variable, or messages are sent to other objects by the `send` constraint.

The assignments and constraints in the right-hand side of a method are treated as a set (where for each component of the state at most one assignment is allowed), i.e., they can be placed in any order: an assignment has no side effect to another assignment in the same method.

A template definition introduces the type of the template, the constructor function and the messages at the top level of the Curry program. If T is the type of the template and the constructor function has n arguments $\tau_1, \ldots, \tau_n$, the type of the constructor function is

$$\tau_1 \to \ldots \to \tau_n \to \texttt{Constructor}\ T$$

In a similar manner, a method has the type

$$\tau_1 \to \ldots \to \tau_n \to \texttt{Message}\ T$$

if it takes n arguments. Additionally, each object understands the predefined message `Stop` which terminates the object.

To instantiate a template, there is a constraint

```
new :: Constructor α → Object α → Success .
```

`new` takes a constructor function and a free logical variable and binds the variable to a new instance of the template α. Messages can be sent to such an object using the constraint `send :: Message α → Object α → Success`. For instance, the evaluation of the following expression binds the variable `v` to the value `42`:

```
new (counter 41) o
 & (send Inc o  &>  send (Get v) o  &>  send Stop o)
```

To give an object the possibility to send a message to itself, there is a predefined identifier `self`. `self` is visible in the right-hand side of each method and bound to the current object. Note that sending a message to `self` has no immediate side effect to the attributes of the object because the objects can only react to this message after the evaluation of the current method is finished.

As a true extension to the modeling of objects in Curry as described in Sect. 2, ObjectCurry also provides inheritance. A template can inherit attributes and methods from another template, which we call *parent*, where inherited methods can be redefined or new attributes and methods can be added. A *supertemplate* of a template T is T or one of its ancestors w.r.t. the parent relation. *Subtemplates* are analogously defined.

For instance, we define a new template `maxCounter` which inherits the attribute `x` and the methods `Inc`, `Set`, and `Get` from `counter`. It also introduces a new attribute `max` which represents an upper bound for incrementing the counter. The method `Inc` will be redefined to avoid incrementing `x` to a value greater than `max`. Additionally, we define a new method `SetMax v` to set the upper bound:

```
template MaxCounter extends Counter =
constructor
  maxCounter init maxInit = counter init
                            max := maxInit
methods
  Inc           =  x   := (if x < max then x+1 else x)
  SetMax newMax =  max := newMax
                   x   := (if x<max then x else max)
```

The reserved keyword `extends` followed by the name of the parent specifies that the template inherits the attributes and methods from Counter.

The first expression in the right-hand side of the constructor of a subtemplate must be the function call of the constructor of the parent. In this way the initial values of the inherited attributes are determined.

Methods can be redefined by defining a method with the same name in the subtemplate. All methods which are not redefined will be inherited.

4 Translating ObjectCurry into Curry

To translate ObjectCurry programs into Curry, we basically use the technique presented in Sect. 2. An abstract data type `Msg` contains data constructors for each message defined in all templates and the additional message `Stop`. We decided to use only one data type for all messages to obtain a maximum of flexibility. Of course, ObjectCurry programs translated in this way are not type safe in a sense that messages can be sent to objects which cannot understand these messages. We will discuss this issue and propose a solution for this in Sect. 5.

For our counter example, we generate one data type for all messages:

```
data Msg = Inc | Set Int | Get Int | SetMax Int | Stop
```

Next we define a function which defines the initial state of a new object. If the state of the object consists of more than one attribute, the state is implemented as a tuple.

```
counterInitState init = init
```

The initialization function of a subtemplate uses the initialization function of its parent to obtain the initial values for the inherited attributes:

```
maxCounterInitState (init,maxInit) =
  let r_x = counterInitState init
  in (r_x,maxInit)
```

Given a state and a message, the following action function computes the next state defined by the corresponding method.

```
counterAction x self Inc              = State (x+1)
counterAction x self (Set s)          = State s
counterAction x self (Get v) | v =:= x = State x
counterAction x self Stop             = Final
```

We use the abstract data type "`data State a = State a | Final`" to distinguish normal states and the final state.

In a subtemplate, redefined and new methods are similarly translated:

```
maxCounterAction (x,max) self Inc
  = State (if x < max then x + 1 else x, max)
```

```
maxCounterAction (x,max) self (SetMax newMax)
  = State (if x < max then x else max, newMax)
```

The action function of a subtemplate also contains an equation for each inherited method. Such an equation calls the action function of the parent of the template for receiving the next state:

```
maxCounterAction (x,max) self (Get v)
  = let State r_x = counterAction x self (Get v)
    in State (r_x,max)
maxCounterAction (x,max) self Stop = Final
```

To create a new object, we use the constructor function and the `new` constraint. The constructor function determines the initial state of the object using the translated function for the initialization defined above and transfers the initial state and the action function of the object to a generic function `loop` which handles the recursive calls until the final state is reached:

```
counter init self =
  loop (counterInitState init) counterAction self
```

For each template the same function `loop` is used which is defined by:

```
loop eval rigid
loop state action self (m:ms) = continuation nextState self ms
  where
    nextState = action state self m
    continuation (State ns) self ms = loop ns action self ms
    continuation Final      _    _  = success
```

The function `new` has a constructor function and a free logical variable as arguments. It creates a port to which the logical variable is bound and passes a stream associated with the port to the constructor function. Additionally, it passes the port to the constructor as the value for the identifier `self`:

```
new constructor port =
  let stream free in
    openPort port stream &> constructor port stream
```

In the transformation, each message has the type `Msg`. Objects are represented by ports, so an object has the type `Port Msg` instead of `Object Template`.

We have implemented a compiler for ObjectCurry which translates a program from ObjectCurry to Curry following the ideas sketched in this section. The compiler is written in Curry itself.

5 Type Safeness

The presented translation into Curry programs is not type safe in the sense that messages can be sent to objects which cannot understand these messages.

To detect such a kind of type errors without restricting the use of objects and messages, it is necessary to define a new type system and implement a new type checker which supports subtyping.

5.1 Subtyping

We introduce a new type system which uses subtype constraints for expressing the types of objects, messages and functions which have such argument types or deliver objects or messages as their results.

First we take a look at the type of constructor functions, objects, messages and the predefined functions `send` and `new`. In a first step, we define three new predefined type constructors named `Constructor`, `Object` and `Message` with arity one. An object as an instance of a template T has type `Object` T. A message has type $\tau_1 \to \cdots \to \tau_n \to$ `Message` T, where $\tau_1, \ldots, \tau_n$ are the types of the arguments of this message and T is the template which defines this message. A constructor of a template T has type $\tau_1 \to \cdots \to \tau_n \to$ `Constructor` T, where again $\tau_1, \ldots, \tau_n$ are the types of the arguments of this constructor. For example, an instance of the template `Counter` has type `Object Counter`, the message `Get` has type `Int` $\to$ `Message Counter` and the constructor function `counter` has type `Int` $\to$ `Constructor Counter`. With these types the function `send` must have the type

$$\texttt{send} :: \texttt{Message}\ \alpha \to \texttt{Object}\ \alpha \to \texttt{Success}$$

and `new` has the type

$$\texttt{new} :: \texttt{Constructor}\ \alpha \to \texttt{Object}\ \alpha \to \texttt{Success}$$

These types do not allow subtyping w.r.t. a Hindley/Milner-like type system [2] as used in Curry. Therefore, we need subtyping in three cases in order to support object-oriented programming techniques and to combine them with the advantages of parametric polymorphism:

1. We want to send messages defined in a template T to instances of subtemplates of T.
2. It should be possible to keep objects of different templates in a polymorphic data structure, e.g., in a list: If these objects have a common supertemplate, there are common messages which all of these objects understand.
3. We also want to store messages defined in different templates in a polymorphic data structure if these templates have a common subtemplate.

Therefore, we introduce subtype constraints and constrained types. We use them to define new types of objects and messages which supports subtyping in the three described cases. Note that, in contrast to other approaches to subtyping or order-sorted types, we consider only subtype relations between templates and not subtyping of standard data types, like numbers or functions, since this is sufficient for our purposes.

Definition 1. *A* subtype constraint *is an expression* $\tau_1 \leq \tau_2$ *where* τ_i *(*$i = 1, 2$*) is a type variable or the name of a template.*

Definition 2. *A* constrained type *is a pair* $\tau|C$ *consisting of a type expression* τ *and a set* C *of subtype constraints. A* constrained type scheme *has the form* $\forall\alpha_1 \ldots \alpha_n.\tau|C$*.*

Intuitively, a constraint of the form $\tau_1 \leq \tau_2$ expresses that τ_1 must be a subtemplate of τ_2. To allow keeping instances of different templates in one polymorphic data structure, an object gets the type $\texttt{Object}\ \alpha \mid \{T \leq \alpha\}$. For example, an instance of `Counter` gets the type $\texttt{Object}\ \alpha \mid \{\texttt{Counter} \leq \alpha\}$ and an instance of `MaxCounter` gets the type $\texttt{Object}\ \alpha \mid \{\texttt{MaxCounter} \leq \alpha\}$. We can keep both objects in a list where this list has the type $\texttt{[Object}\ \alpha\texttt{]} \mid \{\texttt{Counter} \leq \alpha, \texttt{MaxCounter} \leq \alpha\}$. The type of the list is inferred by using standard typing rules but additionally collecting all subtype constraints in one set.

Intuitively, this constraint set can be satisfied because there exists a template T which is a supertemplate of `Counter` and a supertemplate of `MaxCounter`: `Counter` is a supertemplate of both `Counter` and `MaxCounter` . If we mix objects which do not have a common supertemplate, the constraint set cannot be satisfied. This makes sense because these objects do not have a common message and so there is no reason to store them in one data structure. We will formally define the satisfiability of a constraint set later.

Using this type for an object, we must also modify the type of `new` as follows:

$$\texttt{new} :: \texttt{Constructor}\ \alpha \rightarrow \texttt{Object}\ \beta \rightarrow \texttt{Success} \mid \{\alpha \leq \beta\}$$

A similar modification of the type of a message allows to mix messages of different types in a common data structure: A message gets the type

$$\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \texttt{Message}\ \alpha \mid \{\alpha \leq T\}$$

where $\tau_1, \ldots, \tau_n$ are the types of the arguments of this message.

With these definitions it is possible to send a message defined in a template T to an instance of a subtemplate of T: The resulting constraint set can be satisfied iff the object understands the message. For instance, if we send the message `Inc` to an object of the instance `MaxCounter`, we get the typed expression

$$\texttt{send Inc maxCounterObject} :: \texttt{Success} \mid \{\alpha \leq \texttt{Counter}, \texttt{MaxCounter} \leq \alpha\}$$

Unfortunately, we must also modify the type of `send`. Consider the following example:

```
f m1 m2 o1 o2 = send m1 o1 & send m2 o2 & send m1 o2
```

`f` has two messages and two objects as arguments. It sends the first message to the first object, the second message to the second object, and also the first message to the second object. With the type of `send` defined above, we get the type

$$\texttt{f} :: \texttt{Message}\ \alpha \rightarrow \texttt{Message}\ \alpha \rightarrow \texttt{Object}\ \alpha \rightarrow \texttt{Object}\ \alpha \rightarrow \texttt{Success}$$

For our running example, we assume:

$$\begin{array}{ll}
\texttt{Inc} & :: \texttt{Message}\ \alpha\ \mid\ \{\alpha \le \texttt{Counter}\} \\
\texttt{(SetMax 42)} & :: \texttt{Message}\ \alpha\ \mid\ \{\alpha \le \texttt{MaxCounter}\} \\
\texttt{counterObject} & :: \texttt{Object}\ \alpha\ \mid\ \{\texttt{Counter} \le \alpha\} \\
\texttt{maxCounterObject} & :: \texttt{Object}\ \alpha\ \mid\ \{\texttt{MaxCounter} \le \alpha\}
\end{array}$$

Thus, the application of `f` to these arguments would yield the type

$$\begin{array}{l}
\texttt{f Inc (SetMax 42) counterObject maxCounterObject} :: \\
\texttt{Success}\ \mid\ \{\alpha \le \texttt{Counter}, \alpha \le \texttt{MaxCounter}, \texttt{Counter} \le \alpha, \texttt{MaxCounter} \le \alpha\}
\end{array}$$

The set of constraints of this type is not satisfiable because there is no substitution for α such that all constraints are elements of the inheritance hierarchy. This does not match our intuition because it is possible to send `Inc` to `counterObject` and `maxCounterObject` and `(SetMax 42)` to `maxCounterObject`.

The problem can be easily solved if we modify the type of `send`:

$$\texttt{send} :: \texttt{Message}\ \alpha \rightarrow \texttt{Object}\ \beta \rightarrow \texttt{Success}\ \mid\ \{\beta \le \alpha\}$$

This type corresponds to the intuition that a message defined in template α can be send to all instances of template β provided that β is a subtemplate of α. Now the type of `f` is

$$\begin{array}{l}
\texttt{Message}\ \alpha \rightarrow \texttt{Message}\ \beta \rightarrow \texttt{Object}\ \gamma \rightarrow \texttt{Object}\ \delta \rightarrow \texttt{Success} \\
\qquad\qquad\qquad\qquad\qquad \mid\ \{\gamma \le \alpha, \delta \le \beta, \delta \le \alpha\}
\end{array}$$

and "`f Inc (SetMax 42) counterObject maxCounterObject`" has type

$$\begin{array}{l}
\texttt{Success}\ \mid\ \{\gamma \le \alpha, \delta \le \beta, \delta \le \alpha, \alpha \le \texttt{Counter}, \beta \le \texttt{MaxCounter}, \\
\qquad\qquad \texttt{Counter} \le \gamma, \texttt{MaxCounter} \le \delta\}
\end{array}$$

These subtype constraints are satisfiable by the following substitution σ:

$$\sigma(\alpha) = \texttt{Counter}, \sigma(\beta) = \texttt{MaxCounter}, \sigma(\gamma) = \texttt{Counter}, \sigma(\delta) = \texttt{MaxCounter}$$

5.2 Core ObjectCurry

In order to define the type system of ObjectCurry, we introduce a simplified core language to provide a more compact representation of ObjectCurry's typing rules. The expressions and templates of the core language are defined in Fig. 1.

An expression E of the core language is either a variable, a lambda abstraction, an application of two expressions, an expression combined with the declaration of free variables, or a conditional expression. A template T consists of an initial assignment I, which defines the attributes and initial values of the template, and a set of methods. A template can also be defined as a subtemplate of another template by an `extends` clause. I' contains additionally to the initial assignments of the subtemplate a call to the constructor function of its supertemplate. This ensures that each inherited attribute gets an initial value.

$$
\begin{array}{lll}
E & ::= x & \text{variable} \\
 & \mid \ \lambda x.E & \text{abstraction} \\
 & \mid \ E_1\ E_2 & \text{application} \\
 & \mid \ \texttt{let}\ x\ \texttt{free in}\ E & \text{free variable} \\
 & \mid \ \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 & \text{conditional} \\
T & ::= \texttt{Template}\ \mathit{name}\ I\ M^* & \text{template} \\
 & \mid \ \texttt{Template}\ \mathit{name}_1 & \\
 & \qquad\qquad \texttt{extends}\ \mathit{name}_2\ I'\ M^* & \text{subtemplate} \\
A & ::= (x := E)^* & \text{assignment} \\
I & ::= A & \text{initial assignment} \\
 & \mid \ \lambda x.I & \text{abstraction} \\
I' & ::= E, A & \text{initial assignment of subtemplates} \\
 & \mid \ \lambda x.I' & \text{abstraction} \\
M & ::= E \Rightarrow A & \text{body} \\
 & \mid \ \lambda x.M & \text{abstraction}
\end{array}
$$

Fig. 1. A core language for ObjectCurry

A block of assignments A consists of assignments of the form $x := E$ where E is any expression. Due to the fact that a constructor function of ObjectCurry can have some arguments, we allow lambda abstraction on initial assignments.

A method M is defined by an expression E and a block of assignments A. E has to be a constraint (a function with the result type `Success`) which has to be solved when the method is called. The assignments define the transformation of the current state of the object.

A *program* of Core ObjectCurry is a set of definitions of functions and templates. The definition of a function has the form *functionName* $= E$ (where E is usually a lambda abstraction) and the definition of a template is written as

$$(\mathit{constrName}, \mathit{methodName}_1, \ldots, \mathit{methodName}_n) = T\ .$$

Such a program contains no local definitions, i.e., all identifiers are introduced on top level (thus, local declarations in ObjectCurry programs are globalized in Core ObjectCurry by lambda lifting).

As an example, our original `Counter` and `MaxCounter` template definitions are transformed into the core language as follows:

$$
\begin{array}{lll}
(\texttt{counter}, \texttt{Inc}, \texttt{Set}, \texttt{Get}) = & \texttt{Template Counter} & \\
 & \lambda i\ .\ \texttt{x} := i & \textit{(body of } \texttt{counter}\textit{)} \\
 & \texttt{success} \Rightarrow \texttt{x} := \texttt{x+1} & \textit{(body of } \texttt{Inc}\textit{)} \\
 & \lambda s\ .\ \texttt{success} \Rightarrow \texttt{x} := s & \textit{(body of } \texttt{Set}\textit{)} \\
 & \lambda v\ .\ (v =:= \texttt{x}) \Rightarrow \epsilon & \textit{(body of } \texttt{Get}\textit{)}
\end{array}
$$

```
(maxCounter, Inc, SetMax) = Template MaxCounter extends Counter
                            λi . λmi . counter i, max := mi
                            success ⇒ x := if x<max then x+1 else x
                            λv . success ⇒ max := v,
                                        x := if x<max then x else max
```

5.3 A Type System for ObjectCurry

Before we present a type system for this core language, we define the satisfiability of a set of constraints.

Definition 3. *A (type)* substitution σ *is a mapping from type variables to types such that* $\sigma(\alpha) \neq \alpha$ *only for finitely many type variables* α. *We write a substitution as follows:* $\sigma = [x_1/\tau_1, \ldots, x_n/\tau_n]$ *if* $\sigma(x_i) = \tau_i$ *for all* $i = 1, \ldots, n$ *and* $\sigma(y) = y$ *for all* $y \notin \{x_1, \ldots, x_n\}$. *The extension of a substitution to types and constraint sets is obvious.*

In the following we assume that P is a Core ObjectCurry program.

Definition 4. *Let* $\mathcal{H}$ *be the relation of subtemplates of* P *defined by its* `extend` *clauses. The reflexive and transitive closure of* $\mathcal{H}$ *is denoted by* $\mathcal{H}^*$, *also called* inheritance hierarchy.

Definition 5. *A* substitution σ satisfies a subtype constraint $\tau_1 \leq \tau_2$ *w.r.t. the inheritance hierarchy* $\mathcal{H}^*$, *denoted* $\sigma \models_{\mathcal{H}^*} \tau_1 \leq \tau_2$, *if there is a substitution* σ *with* $(\sigma\tau_1, \sigma\tau_2) \in \mathcal{H}^*$.

A substitution σ *satisfies a set* C *of subtype constraints* ($\sigma \models_{\mathcal{H}^*} C$) *if for all* $c \in C$: $\sigma \models_{\mathcal{H}^*} c$.

A set C of subtype constraints is satisfiable *w.r.t. the inheritance hierarchy* $\mathcal{H}^*$, *denoted* $\models_{\mathcal{H}^*} C$, *if there is a substitution* σ *with* $\sigma \models_{\mathcal{H}^*} C$.

Type environments collect the type information for named entities in a program:

Definition 6. *A* type environment Γ *is a mapping from names to constrained type schemes. In the following we denote by TE the set of all type environments.*

The union of two type environments Γ_1 and Γ_2 with non-overlapping domains is defined as follows:

$$(\Gamma_1 \cup \Gamma_2)(\alpha) = \begin{cases} \Gamma_2(\alpha), & \text{if } \Gamma_1(\alpha) \text{ is undefined} \\ \Gamma_1(\alpha), & \text{if } \Gamma_2(\alpha) \text{ is undefined} \end{cases}$$

Additionally, we define another concatenation of two type environments Γ_1 and Γ_2 which gives preference to Γ_2 if an identifier is a member of the domains of both environments. We need this operation in order to extend the global type environment with the attributes of a template.

$$(\Gamma_1 \oplus \Gamma_2)(\alpha) = \begin{cases} \Gamma_2(\alpha), & \text{if } \Gamma_2(\alpha) \text{ is defined} \\ \Gamma_1(\alpha), & \text{otherwise} \end{cases}$$

Generic instances of constrained type schemes are defined as usual:

Definition 7. *A constrained type* $\tau'|C'$ *is a* generic instance *of a constrained type scheme* $\forall\alpha_1 \ldots \alpha_n.\tau|C$ *if there is a substitution* σ *with* $\sigma\ \tau \mid \sigma\ C\ =\ \tau'\ |C'$ *and* $\sigma(\beta) = \beta$ *for all* $\beta \notin \{\alpha_1, \ldots, \alpha_n\}$.

An attribute which is defined in a template T is also visible in the subtemplates of T with the same type. To specify the visibility of attributes in the methods of all subtemplates, we introduce attribute type environments:

Definition 8. *An* attribute type environment Θ : *Templates* $\rightarrow$ *TE maps the name of a template to a type environment. This type environment contains the types of the attributes defined in this template.*

Now we are able to define the well-typedness of Core ObjectCurry programs:

Definition 9. *A function definition* $f = \lambda x_1 \ldots \lambda x_n.e$ *is well-typed w.r.t. a type environment* Γ*, an attribute type environment* Θ*, and an inheritance hierarchy* $\mathcal{H}^*$*, if the following conditions are satisfied:*

- $\Gamma(f) = \forall\alpha_1 \ldots \alpha_m.\tau|C$
- $\Gamma, \Theta, \mathcal{H}^* \vdash \lambda x_1 \ldots \lambda x_n.e : \tau \mid C$ *can be deduced by the rules of Fig. 2 and 3*
- $\models_{\mathcal{H}^*} C$

A template definition $(c, m_1, \ldots, m_n)\ =\ e$ *is well-typed w.r.t. a type environment* Γ*, an attribute type environment* Θ*, and an inheritance hierarchy* $\mathcal{H}^*$*, if*

- $\Gamma(c) = \tau_0|C_0,\ \Gamma(m_i) = \forall\alpha_i.\tau_i|C_i \quad$ *for* $i = 1, \ldots, n$,
- $\Gamma, \Theta, \mathcal{H}^* \vdash e : (\tau_0|C_0, \tau_1|C_1, \ldots, \tau_n|C_n)$ *can be deduced by the rules of Fig. 2 and 3*
- $\models_{\mathcal{H}^*} C_0 \cup C_1 \cup \ldots \cup C_n$

A Core ObjectCurry program is well-typed if there exist a type environment Γ*, an attribute type environment* Θ *and an inheritance hierarchy* $\mathcal{H}^*$ *such that all function and template definitions are well-typed w.r.t. these environments and*

$$\Gamma(\mathit{send}) = \forall\tau_1, \tau_2\ .\ \texttt{Message}\ \tau_1\ \rightarrow\ \texttt{Object}\ \tau_2\ \rightarrow\ \texttt{Success}\ \mid\ \{\tau_2 \leq \tau_1\}$$
$$\Gamma(\mathit{new}) = \forall\tau_1, \tau_2\ .\ \texttt{Constructor}\ \tau_1\ \rightarrow\ \texttt{Object}\ \tau_2\ \rightarrow\ \texttt{Success}\ \mid\ \{\tau_1 \leq \tau_2\}$$

In the inference rules of Fig. 2 and 3, we use the auxiliary functions *super* and *templates* which yield all supertemplates of a template (including the template itself) and all templates of a program, respectively.

In order to check the well-typedness of a program by the rules of Fig. 2 and 3, the type environment Γ must contain the types of each defined function and template. The attribute type environment Θ maps the name of each template to a new type environment which contains the types of the attributes defined in that template. The inheritance hierarchy consists of the subtype relations between all templates which are defined in the program.

The inference rules [Axiom], [Abstraction], [Existential], and [Application] are defined in the usual way, compare the Curry Report [6]. The only modification is the collection of all constraints of all subexpressions into one set of constraints. The satisfiability of this constraint set is checked outside the typing

$$[\text{Axiom}] \quad \frac{}{\Gamma, \Theta, \mathcal{H}^* \ \vdash x : \tau|C} \text{ if } \tau|C \text{ generic instance of } \Gamma(x)$$

$$[\text{Abstraction}] \quad \frac{\Gamma[x/\tau|C], \Theta, \mathcal{H}^* \ \vdash E : \tau'|C'}{\Gamma, \Theta, \mathcal{H}^* \ \vdash \lambda x.E : \tau \to \tau'|C'}$$

$$[\text{Existential}] \quad \frac{\Gamma[x/\tau|C], \Theta, \mathcal{H}^* \ \vdash E : \tau'|C'}{\Gamma, \Theta, \mathcal{H}^* \ \vdash \texttt{let}\ x\ \texttt{free in}\ E : \tau'|C \cup C'}$$

$$[\text{Application}] \quad \frac{\Gamma, \Theta, \mathcal{H}^* \ \vdash E_1 : \tau_1 \to \tau_2|C_1 \qquad \Gamma, \Theta, \mathcal{H}^* \ \vdash E_2 : \tau_1|C_2}{\Gamma, \Theta, \mathcal{H}^* \ \vdash E_1\, E_2 : \tau_2|C_1 \cup C_2}$$

$$[\text{Template}] \quad \frac{\begin{array}{l} name \in templates(\mathcal{H}^*) \\ (name, x) \notin \mathcal{H}^* \text{ for all } x \in templates(\mathcal{H}^*) \text{ with } x \neq name \\ \Gamma' = \Gamma \oplus \Theta(name) \\ \Gamma', \Theta, \mathcal{H}^* \vdash_I^{name} I : \tau_0|C_0 \quad \Gamma', \Theta, \mathcal{H}^* \vdash_M^{name} M_i : \tau_i|C_i \ \ (i = 1, \ldots, m) \end{array}}{\Gamma, \Theta, \mathcal{H}^* \vdash \texttt{Template}\ name\ I\ M_1\ \ldots\ M_m \ : (\tau_0|C_0, \tau_1|C_1, \ldots, \tau_m|C_m)}$$

$$[\text{Subtemplate}] \quad \frac{\begin{array}{l} (name_1, name_2) \in \mathcal{H}^*, (name_2, name_1) \notin \mathcal{H}^* \\ \Gamma' = \Gamma \oplus \bigcup_{p \in super(\mathcal{H}^*, name_1)} \Theta(p) \\ p_i \in super(\mathcal{H}^*, name_1) \quad (i = 1, \ldots, m) \\ \Gamma', \Theta, \mathcal{H}^* \vdash_{I'}^{\substack{name_1 \\ name_2}} I' : \tau_0|C_0 \quad \Gamma', \Theta, \mathcal{H}^* \vdash_M^{p_i} M_i : \tau_i|C_i \ \ (i = 1, \ldots, m) \end{array}}{\begin{array}{c} \Gamma, \Theta, \mathcal{H}^* \vdash \texttt{Template}\ name_1\ \texttt{extends}\ name_2\ I'\ M_1\ \ldots\ M_m \\ : (\tau_0|C_0, \tau_1|C_1, \ldots, \tau_m|C_m) \end{array}}$$

Fig. 2. Typing rules for ObjectCurry programs (1)

rules in the definition of a well-typed program (see Def. 9). In the rule [Abstraction] we do not have to collect the constraints C of the type of the variable x: If E contains an occurrence of x, the constraints of the type of x are collected into the set of constraints of E by the other rules. Otherwise, x is never used and its constraints can be ignored.

In addition to Curry's type system, we introduce new rules [Template] and [Subtemplate] for checking the types of templates and subtemplates. In the rule [Template], which is applicable if there is no true supertemplate in $\mathcal{H}^*$, we extend the type environment Γ by the type assumptions for the attributes of the template in order to make the attribute types visible in the type checking of the methods. Note that the global type environment Γ contains the types of all identifiers defined in the program (including the method identifiers) so that we can use the methods of the template also inside the template and we do not need a special rule for recursion.

$$[\text{Assignment}_1] \quad \frac{\Gamma, \Theta, \mathcal{H}^* \vdash x : \tau | C_1 \qquad \Gamma, \Theta, \mathcal{H}^* \vdash E : \tau | C_2 \qquad \Gamma, \Theta, \mathcal{H}^* \vdash_A A : C_A}{\Gamma, \Theta, \mathcal{H}^* \vdash_A x := E, A : C_1 \cup C_2 \cup C_A}$$

$$[\text{Assignment}_2] \quad \frac{}{\Gamma, \Theta, \mathcal{H}^* \vdash_A \epsilon : \emptyset}$$

$$[\text{Init}] \quad \frac{\Gamma, \Theta, \mathcal{H}^* \vdash_A A : C}{\Gamma, \Theta, \mathcal{H}^* \vdash_I^{name} A : \texttt{Constructor}\ name | C}$$

$$[\text{Init'}] \quad \frac{\Gamma, \Theta, \mathcal{H}^* \vdash E : \texttt{Constructor}\ name_2 | \epsilon \qquad \Gamma, \Theta, \mathcal{H}^* \vdash_A A : C}{\Gamma, \Theta, \mathcal{H}^* \vdash_{I'}^{\substack{name_1 \\ name_2}} E, A : \texttt{Constructor}\ name_1 | C}$$

$$[\text{Method}] \quad \frac{\Gamma, \Theta, \mathcal{H}^* \vdash E : \texttt{Success} | C \quad \Gamma, \Theta, \mathcal{H}^* \vdash_A A : C' \quad v \text{ new type variable}}{\Gamma, \Theta, \mathcal{H}^* \vdash_M^{name} E \Rightarrow A : \texttt{Message}\ v | \{v \leq name\} \cup C \cup C'}$$

$$[\text{Abstraction}_{\mathcal{X}}] \quad \frac{\Gamma[x/\tau|C], \Theta, \mathcal{H}^* \vdash_{\mathcal{X}}^n X : \tau' | C'}{\Gamma, \Theta, \mathcal{H}^* \vdash_{\mathcal{X}}^n \lambda x.X : \tau \to \tau' | C'} \qquad \mathcal{X} \in \{I, I', M\}$$

Fig. 3. Typing rules for ObjectCurry programs (2)

The rule [Subtemplate] is similar to [Template] except for the following differences:

- The type environment Γ' also contains the type assumptions of the inherited attributes, i.e., the attributes of the current template and all its supertemplates.
- I' contains a call to the constructor function of the parent. It must be checked that this has the type `Constructor` $name_2$ where $name_2$ is the name of the parent. This is ensured by using $\vdash_{I'}$ instead of $\vdash_I$.
- Furthermore, we have to ensure that $(name_1, name_2)$ is an element of the type hierarchy $\mathcal{H}^*$ and $(name_2, name_1)$ must not be in $\mathcal{H}^*$. Due to the fact that $\mathcal{H}^*$ is transitive and reflexive, it also contains $(name_2, name_2)$, $(name_1, name_1)$, and $(name_1, p)$ for all supertemplates p of $name_1$.
- For checking the types of the methods, we also allow that a method M_i is assigned to some supertemplate p_i (note that p_i is the current template $name_1$ or one of its supertemplates). This is necessary if the method is redefined. Note, however, that methods redefined in subtemplates must have the same type as in supertemplates. This is reasonable since, due to the logic features of Curry, arguments of a method can be used as value parameters as well as result parameters so that a contra- or covariance restriction on arguments cannot be clearly required.

[Template] and [Subtemplate] use the rules of Fig. 3 which we discuss next. The rule [Assignment$_1$] ensures that in an assignment of the form x `:=` E the type of x is the same as the type of the expression E. [Assignment$_2$] handles the special case of an empty list of assignments. The rule [Init] checks the type of a constructor function where the name of the template must be provided as an extra argument. [Init'] additionally checks if E is a valid call of the constructor function of the parent. For this purpose, we also need the name of the parent ($name_2$). The rule [Method] types a method with subtyping the result type as discussed in Sect. 5.1. It checks whether the expression E of a method $E \Rightarrow A$ is a constraint (with the type `Success`) and collects the resulting constraints.

Due to the fact that we need lambda abstraction over initial assignments I or I' and methods M, we introduce a generic rule [Abstraction$_\mathcal{X}$]. $\mathcal{X}$ can be I, I' or M. The rule is similar to the common rule for abstraction.

5.4 Type Inference

We have also developed a type inferencer for our modified type system. Due to lack of space we can not present it here but refer to [12] which contains the complete description of the type inferencer and its implementation. The algorithm is based on the algorithm $\mathcal{D}$ of Kaes [10]. However, our inference algorithm is simpler because we allow subtyping only for objects and messages. The algorithm unifies type expressions in the same way as standard type inference algorithms [2] but additionally collects the subtype constraints. The resulting set of subtype constraints is then checked for satisfiability with a simple test procedure.

Our implementation of the type checker for ObjectCurry is based on Mark Jones' "Typing Haskell in Haskell" [9] which we adapted to Curry. The implementation of the ObjectCurry compiler together with the type inferencer is freely available from the authors.

6 Related Work

In this section we compare ObjectCurry with some other approaches for the object-oriented extension of functional (logic) languages.

Oz [17] is a concurrent constraint programming language with a particular syntax for object-oriented programming, thus, offering similar features as ObjectCurry. The main differences between ObjectCurry and Oz are the type system and the operational semantics. Oz is untyped and supports no detection of type errors at compile time in contrast to ObjectCurry. Furthermore, the operational model of ObjectCurry is based on Curry's computation model [4] which combines an optimal lazy evaluation strategy [1] for the functional (logic) parts of a program with the concurrent evaluation of constraints. In particular, we consider objects as functions consuming the stream of incoming messages where the state is passed as an argument between the different function calls. In contrast, Oz evaluates functions in an eager manner and implement stateful objects via a specific cell store.

Haskell++ [7] extends Haskell's type classes to object classes. It provides a limited form of multiple inheritance and virtual methods but does not provide subtype polymorphism. For instance, it is not possible to create a list with elements of different instances of one object class. The main goal in the development of Haskell++ was a minimal extension to Haskell which supports the inheritance of functions. Objects in Haskell++ contain only methods but no states. On the other hand, ObjectCurry provides real objects with states in the sense of object-oriented programming. It combines the flexibility of conventional object-oriented languages with the features of functional logic programming.

O'Haskell [13, 14] provides an extension for full object-oriented programming with states and subtype polymorphism. It uses monads for the implementation of concurrent objects and states. The main advantage of our implementation, which uses the concurrent and logical features of Curry, is the opportunity to combine this with Curry's port concept [5] for distributed programming. In contrast to O'Haskell, objects in ObjectCurry can also be executed in a distributed setting. This is supported by a function `newNamedObject` which is similar to `new` but makes the new object accessible from other machines in the network with a unique port identifier (see [5] for more details). The implementation of objects remains unchanged. Furthermore, the logical variables in Curry can be exploited as answer channels since the receiver of a message can bind the logical variables in the message to send answers back to the sender.

Finally, Objective Caml [11] is an object-oriented extension of ML. Objective Caml inherits the strict evaluation strategy of ML and subtype polymorphism can only be programmed with explicit coercions in contrast to ObjectCurry which is lazy and provides subtype polymorphism without any annotations since all types can be automatically inferred.

7 Conclusions

We presented the language ObjectCurry as an extension of Curry to allow a convenient definition of objects via templates. Templates play the role of classes in conventional object-oriented languages. A template defines the attributes and methods of an object. Methods are used to determine the reactions to incoming messages where reactions can be the change of the object's state or a constraint to send messages to other objects. Assignments are used to express a transformation on the local state of an object. Templates can also inherit attributes and methods from other templates and inherited methods can be redefined.

We proposed a direct translation of templates into pure Curry but translated target programs using more than one template are not type safe in the sense of traditional typed object-oriented languages. Therefore, we developed a new type system which uses subtype constraints in the types of objects, messages and functions which use objects or messages. We implemented a compiler which translates ObjectCurry programs into Curry and a type checker which also infers types of expressions without explicit type annotations.

Acknowledgements. The authors are grateful to the anonymous referees for their helpful remarks to improve the final version of this paper.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
2. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.
3. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
4. M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 80–93, 1997.
5. 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.
6. M. Hanus. Curry: An Integrated Functional Logic Language, 2000. `http://www.informatik.uni-kiel.de/~curry/`
7. J. Hughes and J. Sparud. Haskell++: An object-oriented extension of Haskell. In *Proceedings of the Workshop on Haskell*, La Jolla, California, YALE Research Report DCS/RR-1075, 1995.
8. S. Janson, J. Montelius, and S. Haridi. Ports for objects in concurrent logic programs. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 211–231. MIT Press, London, 1993.
9. M.P. Jones. Typing Haskell in Haskell, 1999. In *Proceedings of the Workshop on Haskell*, Paris, France, Technical Report UU-CS-1999-28, University of Utrecht, 1999.
10. S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *1992 ACM Conference on Lisp and Functional Programming*, pages 193–204. ACM, ACM, August 1992.
11. X. Leroy. The Objective Caml system. Technical report, 1996. `http://pauillac.inria.fr/ocaml/`.
12. P. Niederau. Object-oriented extension of a declarative language (in german). Master's thesis, RWTH Aachen, 2000.
13. J. Nordlander. Rationale for O'Haskell, August 1999. `http://www.cs.chalmers.se/~nordland/ohaskell/rationale.html`
14. J. Nordlander. *Reactive Objects and Functional Programming.* PhD thesis, Chalmers Göteborg University, May 1999.
15. J. Peterson et al. Haskell: A non-strict, purely functional language (version 1.4). Technical report, Yale University, Yale, 1997.
16. E. Shapiro and A. Takeuchi. Object oriented programming in Concurent Prolog. *New Generation Computing*, 1:25–48, 1983.
17. 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.