Implementation Guide

Marc André Wittorf

Institut für Informatik, CAU Kiel, Germany

November 24, 2023

Contents

This document shall give as much help as possible to implement a new backend on top of the ICurry intermediate format. It includes descriptions for all necessary functions that make up a unoptimized and slow but still complete runtime system, a description on how to translate ICurry constructs and how to implement some of the more complex external functions.

1 Runtime System

The runtime system's purpose is to coordinate the execution of functions with respect to their laziness and non-determinism. It shall track all branches of computation, cycle through them to enable some form of concurrent evaluation and dispatch the correct functions.

The runtime system is an implementation of *The Fair Scheme* [1]. It uses a queue of stacks, each representing a branch in the non-deterministic computation.

In this document, we give an abstract API, which can hopefully serve as a guide on how this runtime system could be implemented in any language. We try to keep it language-agnostic, favoring (semi-)formal or informal descriptions over constructs which may be influenced by some programming languages being absent from others.

1.1 Data Structures

The complete program state is one big, directed graph. Many operations deal with a single node and its direct children, so the chosen data structure should allow to access this kind of information cheaply. There are no operations, which deal with all nodes or all edges at once, so a direct implementation of the usual mathematical definition of a graph using a set of vertices and a set of edges is not recommended. A faster approach would likely be to encode all outgoing edges directly into each node. While the number of children theoretically is not limited, a suitably high maximum number of children may be selected. This is reasonable as there will never be more children than the arity of a function or constructor. A number of arguments that high is not expected in any program.

This section highlights the important parts of this graph and some structures which hold information used to correctly and efficiently transform this

[1] Sergio Antoy and Andy Jost: Compiling a Functional Logic Language: The Fair Scheme. In: Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers. DOI 10.1007/978-3-319-14125-1_12

graph.

1.1.1 Node

A node provides the identity for a (sub-)term. This means that evaluating its represented term will not create a new node, but rather update this existing one. Hence, a node needs to be mutable. In addition to its children, it needs to encode a label. This label determines what kind of term this node represents. It can be one of the following:

1. A failure.
2. A function call.
3. A constructor call.
4. A literal.
5. A choice.

Function calls, constructor calls and literals also need to encode, which function or constructor is called or which literal is meant. For functions, this can be achieved by either adding a reference to the function, using a *function pointer*, *callable object* or similar mechanism, or by using a custom dispatcher and some function identifiers. Constructors need to include their type-unique identifier. If the node is labeled to contain a literal, this literal is to be included. Nodes containing a choice additionally need a mechanism to match multiple choice nodes representing the same choice. This will further be explained in section 1.1.3.

The runtime system needs to be able to differentiate between a function call and a constructor. However, literals can be seen as a set of constructors and thus a differentiation between a constructor call and a literal is not strictly needed, but may be convenient in many places.

1.1.2 Edge

Edges need to encode a source and a target node. Also, edges are sorted as seen from the outgoing node. There are no labels or additional information. Thus, a useful representation for edges is to include a list of all child nodes into each node.

1.1.3 Matching Choices

When doing a *pull-tab* step in the presence of shared nodes, single nodes have their label copied. While this is not a problem for the other types of labels, it eventually causes all choice nodes to be copied into completely independent nodes. This manifests as a run-time choice semantic, which is not desired. The solution is to mark all copies of a choice as belonging together and to make the same decision for each of these nodes.

A simple mechanism for doing this is to include an incrementing integer as *choice identifier* into a choice-labeled node. This choice identifier is equal for all nodes representing the same choice and different between any two nodes belonging to different choices. However, it may be convenient to assign these identifiers lazily. By that we avoid having to pass a the mechanism for obtaining new identifiers to all parts of the program but rather lets a few functions in the runtime system deal with organizing these identifiers. Furthermore, lazy assignments reduce wasting identifiers on choices which will never be evaluated. These identifiers are often a finite resource, so avoiding waste is desirable.

1.1.4 Stack

The stack, or rather *a* stack, is a classic stack over references to nodes. It is not necessary for correctly implementing the runtime system, as its information can staightforwardly be computed by walking the graph. However, it is one of the most important parts for making the runtime system efficient.

1.1.5 Map of Decisions

As hinted at in section 1.1.3, for nodes representing the same choice the same decision has to be made. Thus, we need to keep the information which way each choice was decided. If the matching is done using the proposed *choice identifiers*, this information can be held as a set of $\{(c, n) \mid c \in \mathbb{N}, n \in \mathbb{N}\}$, where c shall be the choice identifier and n the position of the chosen child. Preferably, this data structure shall allow fast lookup by the *choice id*, so a structure called *map* in many languages is a good fit.

1.1.6 Queue

The queue, being the main component of a breadth-first search in the graph is the core mechanism to achieve computational completeness within non-deterministic programs.

It shall contain pairs of a stack (see 1.1.4) and a decision map (see 1.1.5). As this structure is a queue, taking from one end and inserting at the other end should be fast. Other operations are not needed to implement the core of the runtime system. Some more advanced mechanisms like the *concurrent and* (Curry: `(&) :: Bool -> Bool -> Bool`) or encapsulated search may want to have finer control of the queue. However, this is currently not in the scope of this document.

1.2 Memory Management

The runtime system constantly updates a graph by adding new nodes, creating links between existing nodes and breaking these links. As it does not define, when a node may be discarded, there needs to be a system that can detect whether a node is still in use or can be freed from memory. Simple reference counting is not enough for this task, as the graph is allowed to contain cycles which can be discarded at once, and there is no point where these cycles could be broken using weak references.

Rather, a complete system for garbage collection is needed. If the target language itself relies on a garbage collector, it is wise to utilize this for our data structures. If the target language requires explicit memory management, a full garbage collector must to be implemented for nodes. It needs to be able to detect stale cycles and free them.

1.3 Procedures

Now we will give some procedures that form the runtime system when implemented as described. Depending on the target language, some of these are just different names for built-in functions dealing with data structures.

All procedures are given as `name :: arguments -> return-type`, with `argument` being a comma-separated list of argument definitions. An argument is given as `name:type`. A type can be one of the data structures described above, a base type such as `Int` or `Bool`, a tuple of two types, given as `(type1, type2)`, a list of a type, denoted as `[type]`, or a type with a hint that data shall be given by reference `type*`.

Depending on the procedure, being marked as pass-by-reference has at least one of two implications for an argument or return value. Modifying the argument shall modify the whole state (sometimes called an *out*-argument). Or some data shall be referenced from somewhere else (*sharing*). Thus, all reference-marked types shall adhere to some reference-semantics.

Not being marked as reference, however, does not require passing by value. Those pieces of data may be passed by reference as well, if this holds

out the prospect of faster execution.

For brevity, the decision map will be abbreviated as `DM` in type signatures.

1.3.1 `updateNode :: n:Node*, l:Label, c:[Node*] -> Void`

`updateNode` shall write the label `l` into the node `n`, disconnect all child nodes from `n` and then make all nodes in `c` children of `n`.

1.3.2 `push :: s:Stack*, n:Node* -> Void`, `pop :: s:Stack* -> Node*`, `peek :: s:Stack* -> Node*`

`push`, `pop` and `peek` shall be standard stack operations.

1.3.3 `next :: q:Queue* -> (Stack*, DM*)`, `enqueue :: q:Queue*, (Stack*, DM*) -> Void`

`next` and `enqueue` shall return the next element at one end of a queue, while removing it from the queue, or respectively insert an element at the other end of a queue.

1.3.4 `ensureHasChoiceId :: n:Node*`

`ensureHasChoiceId` uses a global supply of identifiers to assign one to a *choice*-labeled node, if this node does not yet have an identifier.
It does not need to check if the passed node is a choice, as it will only be called on choices. It shall execute the following:

if $n.choice\_id$ *is* $<unset>$ **then**
  $n.choice\_id \leftarrow nextId$;
  $nextId \leftarrow nextId + 1$;
end

1.3.5 `pull :: n:Node*, p:Int -> Void`

`pull` executes a *pull-tab* step. This is an operation that pulls a choice towards the root of an expression. Instead of passing the choice-labeled node to this procedure, its arguments are a parent node and a position `p`. A node can be referenced from multiple positions, so a position is strictly required to correctly determine, over which edge this pull-tab step shall be executed over.
`pull` shall execute the following:

$s \leftarrow n.label$;
$cs \leftarrow n.children$;
$c \leftarrow cs[p]$;
$ensureHasChoiceId(c)$;
$cid \leftarrow c.choice\_id$;
$n.label \leftarrow <choice>$;
$n.choice\_id \leftarrow cid$;
$n.children \leftarrow []$;
forall $i \leftarrow c.children$ **do**
    $n' \leftarrow$ **new** $Node$;
    $n'.label \leftarrow s$;
    $n'.children \leftarrow$ **copy of** cs;
    $n'.children[p] \leftarrow i$;
    $n.children \leftarrow n.children + +[n']$;
end

1.3.6 `step :: s:Stack* -> Bool`

`step` does a single step towards the *Head Normal Form*.
It shall execute the following:

```
n ← pop(s);
if n.label is constructor or n.label is literal then
    return |s| > 0;
end
if n.label is <fail> then
    return False;
end
if n.label is function then
    f ← n.label;
    r ← f(n);
    if r is <no_argument_needed> then
        push(s, n);
        return True;
    end
    if n.children[r] is <choice> then
        pull(n, r);
        push(s, n);
    end
    if n.children[r] is function then
        push(s, n);
        push(s, n.children[r]);
    end
    return True;
end
if n.label is <choice> then
    return True;
end
return False;
```

`step`'s return value indicates whether the runtime system shall enqueue this nondeterministic branch again. A return value of `False` shows that this branch requires the head-normal form of a node that is a failure, and thus is a failure itself.

1.3.7 `dispatch :: q:Queue* -> Void`

`dispatch` is the main procedure coordinating the execution.
It shall execute the following:

```
(s, m) ← next(q);
n ← peek(s);
if n.label is <choice> and |s| = 1 then
    ensureHasChoiceId(n);
    b ← m[n.choice_id];
    if b is <not_found> then
        for (i, n') ← zip([0..], n.children) do
            m' ← copy of m;
            m'[n.choice_id] ← i;
            s' ← new Stack;
            push(s', n');
            enqueue(q, (s', m'));
        end
    end
    else
        updateNode(n, n.children[b]);
        enqueue(q, (s, m));
    end
end
else
    r ← step(s);
    if r then
        enqueue(q, (s, m));
    end
end
```

1.3.8 `run :: q:Queue* -> Void`

`run` shall repeatedly call `dispatch` (see 1.3.7) as long as the queue is non-empty:

```
while |q| > 0 do
    dispatch(q);
end
```

2 The Backend

Most of the work necessary for translating a Curry program to a new target language is already done by the *curry-frontend* and the transformations from the *icurry* package. The emitted *ICurry* format is designed to be structured imperatively and shall enable a sufficiently easy translation to the desired

target language by requiring only slightly more logic than what is needed for a pretty printer. Furthermore, the *Extended ICurry* format executes some more transformations on top of the *ICurry* representation, which only make implementing the actual backend a little more straightforward.

2.1 (Extended) ICurry Structure

An (Extended) ICurry program consists of four parts: a module name, a list of imported modules, a list of declared data types and a list of declared functions.

Apart from the module name, all these pieces of information should be needed to emit a complete program. The module name, however, may be required for some target languages to help naming the module or single functions.

The import list is a comprehensive list of all modules used in this module. Every function or datatype that is referenced in this module is defined either in this module or in one of the modules in this list. Still, this list may contain unused modules, as there is currently no mechanism to prune imported but unused modules.

2.2 Data Types

Data type declarations have a name, a number of type variables and a number of constructors. Every data type shall be compiled into a generator function.

A generator function looks just like any other function and it is used exactly the same. Its arguments arise from the data type's type arguments. They are used to parameterize this generator by passing an appropriate generator for every type argument.

The generator never requires the head-normal form of any argument. It shall only set its node to a choice. This choice shall contain one constructor call for every constructor of this type. The constructors' arguments are determined as follows:

If a type variable is referenced in the constructor's signature, a function call to the `Prelude.unshare` function, with the variable as only argument, shall be passed. `unshare` will be explained in section 4.1. If a type constructor is given instead, a function call to this data type's generator is inserted. Its arguments are found by doing this recursively.

2.3 Functions

A function is either compiled or defined externally. If a function is defined externally, the external implementation shall be called and no further action is necessary.

Translating a function body is the more interesting case. All specified arguments shall be unpacked from the node the function operates on. Names for variables in the target program can directly be derived from each `IVarIndex`, for example by converting the index to a string and prepending a letter. They are meant to be sufficiently unique.

Then, the block defines the actual logic happening in the function. All blocks carry a number of local variables and a number of assignments. Before dealing with the logic specific to each different block, all these local variables shall be declared and defined to an empty node. This is needed because these new variables can be used before they are assigned, for example to allow cyclic data structures. Then it shall process each assignment by assigning the structure arising from the given expression to the given variable.

A simple block shall now set the contents of the current node to the graph structure constructed from the expression.

A case block shall examine the specified variable. If it is not in head-normal form, the function aborts by returning the position of this variable in the function's argument list. The variable is guaranteed to always be a function argument. If it is in head-normal form, it shall find the correct branch based on the constructor in this variable. Then it shall process the block given in this branch.

For a case differentiation over literals it may happen that no branch matches. In this case the node shall be set to a failure.

For a differentiation over constructors of a data type, this can never happen. All cases resulting in a failure are given explicitly. As constructors can have arguments, these must be unpacked to the variables by their position just like function arguments, before the block specified in this branch is processed.

Evaluating expressions builds a subgraph. `ILit`, `IFCall`, `ICCall` and `IOr` (respectively their `IE`-counterparts) generate new nodes labeled accordingly. `IVar` (respectively `IEVar`) just gives an existing node which is referenced by a variable.

3 Input/Output

As the described method of evaluation is lazy, the order of evaluation of every subterm is not set by the order of their appearance in the source program. While this is not a problem for pure functions, IO requires the ability to specify the order of execution. This is solved using the `IO` monad, which enforces every action to be executed in order. To do this, an implicit *world* object is passed between IO actions that are being executed. This world could be seen as actually containing all data for input and output, allowing to see IO actions as pure functions for theoretical arguments. In practice, however, this world object contains much less, nothing in fact. It is a mere dummy for guarding the access to the actual world (the user, file system, network, etc.) and hence, can be represented by the smallest applicable type.

An action in the source language with type `IO a` can be implemented as a function `() -> (a,())`, with the world being represented through a unit type. It is important to only execute the action and produce a result, once the world argument actually is evaluated to head-normal form. This small detail ensures the correct evaluation order.

As IO actions are composed using `Prelude.>>=$ :: IO a -> (a -> IO b) -> IO b`, this function has to properly pass the world between the two actions. A usual implementation would be to pass the world to the first action, wait for its evaluation to head-normal form, extract the world from the action's result, pass this world to the second action and then make this second action's result the whole result.

Although at first glance this may look like an additional translation step, all basic IO actions are defined as external functions and thus are not translated at all. IO actions defined in the source program are always composed from these basic actions using the previously explained bind operator, which takes care of handling the world.

3.1 `catch` and the World

In practice, IO actions can throw errors. In the source language, these errors have to be handled using the function `catch :: IO a -> (IOError -> IO a) -> IO a`. To properly allow this, it may be customary to have the world slightly more complex than the unit type. An equivalent of `data IOWorld = WorldOK | WorldError IOError` allows carrying an additional error.

`>>=$` then has to immediately return this error when receiving a `WorldError` from the first action. `catch` basically does the opposite of `>>=$`: It immediately returns if no error is seen and prepares and starts the second action (the error handler) if it receives an error from the action.

3.2 Starting an IO Action

Starting an IO action is required if the main function is an IO action or when invoking `unsafePerformIO`. Then, the action shall be copied, this copy shall receive a fresh world object and afterwards it can be assigned to a node, ready to be evaluated. In the case of `unsafePerformIO`, the action may not be directly assigned to the node that previously contained the call to this unsafe function. Instead it must be wrapped in another function that will extract the actual result from the world structure. Copying the action is necessary, as an action may be shared and every call needs to attach its own world object. Modifying the shared action duplicates a world and thus loses the enforcement of evaluation order.

4 External Functions

Many functions that need a native implementation in the target language are not very complex. Just by looking at their signature and maybe their documentation, one should be able to immediately have an idea how to implement them.

As there are many externally defined functions in the *curry-base*, this section will only focus on a few more complicated functions.

Most externally defined functions do not need to handle unevaluated nodes. Small stubs in the curry libraries ensure that these external implementations are only called on nodes which have already been evaluated to (head-)normal form.

4.1 `Prelude.unshare :: a -> a`

`unshare` is the only function added by the translation to *ICurry*. As there is no external documentation for this function, it is included in this document.

Its purpose is to separate a choice node from all other nodes with its choice identity. This is necessary to enable proper semantics for choices in the presence of generators. Without splitting the identities between a passed generator and its use, non-deterministic branches between *different* free variables would be shared. Splitting these identities reduces these sharings to exactly those expected by call-time semantics.

Usually, `unshare` will create a shallow copy of the node which is the only argument that `unshare` is called on. The copy will reference the same nodes as children and no children are changed or copied. `unshare` will reset the copy's choice tag (see 1.1.3), so it will be given a new one on occasion. This copy is the result of this function.

4.2 `Prelude.apply :: (a -> b) -> a -> b` and similar

`apply` looks like the identity function on a functional type. It is used, however, to construct a (potentially partial) function/constructor call from a partial function/constructor call and an argument.

The partial application needs to be copied to a new node. This new node can now be modified by adding a new child reference to the new argument. Copying is necessary, as a partial application may be used multiple times with different arguments, so the original partial application must be preserved.

Several other functions use (partial) applications in their mechanism (for example `$!`, `$#`). They can either use the `apply` function or just replicate this process in place.

4.3 `(Prelude.=:=) :: a -> a -> Bool`

`(=:=)` does a few things at the same time. Not only does it compare two data terms, but it also takes shortcuts by binding free variables and thus avoids lots of useless work.

The comparison recursively descends both arguments. This, however, may not be done strictly in the function body of `(=:=)`, as one of the data terms may be infinite or may contain unevaluated nodes. Strictly evaluating an inifinite structure would lead to an infinite computation in this function. This would immediately break the completeness of the Fair Scheme. An unevaluated node cannot be compared but must be evaluated first. Since a function can only request to evaluate those nodes which are function arguments, this would be a problem.

Instead, this function only compares both data terms' constructors, failing on inequality, and then returns a new subgraph, which represents the conjunction of the structural equalities of each argument. This conjunction shall preferably be realized with a mechanism like the one used in the function `&`, so the comparison is more like a breadth-first search than a depth-first search.

Binding a free variable is unique to this function. If one of this function's arguments is a generator, the generator's node can unconditionally be updated so that it bears the same label and children as the other argument. This requires the ability to distinguish a simple *choice* from a *generator*, as generators are choices as well. In this, generators are special choices and could (only for use in this function) be marked to be suitable for unification.

4.4 `(Prelude.$!!) :: (a -> b) -> a -> b` and `toNF :: a -> a`

`($!!)` evaluates the argument to normal form before passing it to the function. This can easily be implemented by using a helper function `toNF`, which is not part of a Curry library. `f $!! x` then only has to request the evaluation of `toNF x` to head-normal form, which can easily be achieved by passing this term as an argument to a helper function, which then can request the evaluation to head-normal form.

`toNF` then has to ensure that it only ever produces a head-normal form, if the result is also in normal form. This is achieved by recursively using this function and waiting for *all* arguments to be processed before constructing the result. In other words, this function can be seen to emulate a strict identity function in a lazy runtime system.

4.5 `Prelude.readFile :: String -> IO String`

While `writeFile` is easy to implement, because it only receives completely evaluated arguments and may do all the work in a single go, `readFile` should read lazily.

This can be done by obtaining a handle from opening the file, and then returning an applied function call to a `read :: Handle -> String` function as IO result. This is still a correct implementation, as the mechanism of sharing avoids trying to read the same position multiple times, although the function modifies the handle and, thus, is not purely functional. `read` then can be implemented as the equivalent of

if *handle is eof* **then**
  close handle;
  return [];
end
else
  $c \leftarrow$ *readChar handle*;
  return c : *read handle*;
end

close and *readChar* denote the native functions to close a file handle and to get the next character from an open file handle.

4.6 The `Global` Module

The module `Global` allows to define and allow constructs that behave like global variables in imperative languages. While their usage is safe as reading

and writing are performed in IO actions, global variables are defined using the pure function `global`. This is a problem, because global definitions are never shared in Curry [2]. No naive implementation will allow effective writing to a global variable, as a new instance is created everytime the global variable is referenced, which makes it impossible to read a modified value. Also there is no argument which would allow the `global` function to distinguish between different global variables.

To solve this, the translation can introduce a special handling for functions which are a mere call to the `global` function. These functions shall not be translated to a single function, but rather into two parts. The first part is an explicit subgraph containing the starting value (or data read from persistent storage), saved in the equivalent of a global variable. The second part is a function which simply returns a reference to this global structure.

This way, every reference to the global variable's definition will eventually be evaluated to the same structure in memory. A write to this variable can then be reflected in later reads from this global variable.

4.7 `IO.Handle`

The `IO` module defines a data type `Handle`. It also introduces functions to access files and file-like objects making use of this data type.

Other modules, especially the `Socket` module, however, also use handles. Consequently the `IO` functions working on a `Handle` not only need to be able to access files, but also other resources which may require different low-level interfaces. Thus, the `Handle` likely needs a mechanism to dispatch the correct functions for dealing with a handle at runtime.

[2] Michael Hanus (ed.): Curry: An Integrated Functional Logic Language (Vers. 0.9.0). 2016. Available at `http://www.curry-language.org`, last accessed: 2018-09-16