The ICurry Format

Sergio Antoy

May 25, 2016

Abstract

This document describes the ICurry format, a representation of Curry programs developed for a Curry compiler that takes a Curry source program and generates the corresponding imperative and/or object-oriented executable code. ICurry is an intermediate abstract representation of Curry program between the source and the executable formats.

Contents

1 Introduction

The ICurry Format defines various concepts that can be found in Curry programs, e.g., module, functions, expressions, etc. Below we define these concepts. There is no type information in ICurry because only well typed Curry modules are translated and type information is not used during the execution of a program.

In the examples that follow ICurry code is represented in an informal format called ".read" that eases understanting the structure of the code. The Curry definition of the ICurry format is in a file called ICurry.curry. The following Curry module is our running example.

Ex:

```
module bintree where
data Bintree x = Leaf | Branch x (Bintree x) (Bintree x)
isin _ Leaf = False
isin x (Branch y l r) = x==y || isin x l || isin x r
```
(1)

2 Elements

In this section we define the elements of the ICurry format.

2.1 module

A complete ICurry unit is a module. An ICurry module is a representation of a Curry module. An ICurry module defines:

1. the module name, a string
2. the imported modules, a list of strings
3. the data types declared by the Curry module, a list of *data* types
4. the functions defined by the Curry module, a list of *function*

Ex:

```
module "bintree"
  import
    "Prelude"
  data ...
  function ...
```
(2)

2.2 name

In ICurry a name represents a qualified identifier in Curry. It is a pair of string with a dot in between.

Ex:

```
"bintree.Bintree"
```
(3)

2.3 data

A data declaration declares a data type and its data constructor. The data type is a name. A data constructor is a name plus its arity and an index. Data constructors of a data type are indexed sequentially starting from 0. As we said earlier, no types and no type variables are present in ICurry.

Ex:

```
data "bintree.Bintree"
  constructor "bintree.Leaf" 0 0
  constructor "bintree.Branch" 3 1
```
(4)

2.4 path

Curry manipulates graphs, which later we call expressions. A *path* allows us to identify a subexpression in an expression. For example, consider the following fragment of code:

```
let x = Branch 11 (Branch 12 Leaf y) Leaf
    y = Branch 13 (Branch 14 x Leaf) Leaf
in x
```
(5)

In the binding of x, y is the 3rd argument of the second argument and the nodes of the parents are both labeled by the *Branch* symbol. Assuming that x is understood, this path is represented by:

```
"bintree.Branch":2 "bintree.Branch":3
```
(6)

2.5 variable

In ICurry a variable identifies a graph node, i.e., an expression. Variables are declared in the code of functions. They are idenfied by an index in $0, 1, 2, \ldots$

unique within a function. There are several kinds of variables:

1. *ILhs* (n, i): identifies the i^{th} argument of an expression rooted by a node labeled by n and matched by the left-hand side of some rule.
2. *ICase*: identifies an expression used as the selector of an *ATable* or *BTable*.
3. *IVar* j (n, i): identifies the i^{th} argument of an expression rooted by a node labeled by n and bound to variable j.
4. *IBind*: identifies a variable bound by a let block.
5. *IFree*: identifies a free variable.

Examples of variables are in the body of functions.

2.6 expression

The objects of a Curry computation are graphs, also called expressions. Informally, an expression is a variable or the application of a symbol to a sequence of arguments. There a handful of particular expressions such as integers and characters. There are several kinds of expressions:

1. *exempt*: an expression representing "no expression".
2. *reference_var* i: the variable identified by integer i.
3. *int* i: the builtin integer i.
4. *char* c: the builtin character c.
5. *Node* n $(l_1, \ldots l_k)$: application of the symbol whose name is n to the sequence of expressions $l_1, \ldots l_k$
6. *partial* i *exp*: *exp* is a partial application missing i arguments.

For example, given the program:

```
main xs = map (2 +) xs
```
(7)

the expression representing the left-hand side of *main* is:

```
Node "Prelude.map" (
  partial 1 (
    Node "Prelude.+" (
      int 2 ) ) ,
  reference_var 1 )
```
(8)

where variable 1 is bound to *xs*.

2.7 statement

The body of a function is a sequence of statements. There are several kinds of statements:

1. *Comment* s: where s is a string providing information generated by the translator, not a comment in the source Curry program. The execution should ignore comments.
2. *Declare* i v: i is an integer uniquely identifying variable v within a function. The execution declares the variable.
3. *Assign* i *exp*: i is the identifier of a variable and *exp* an expression. The execution binds *exp* to the variable.
4. *Fill* i *path* j: integers i and j are variable identifiers, *path* is a list of (n, k) pairs, where n is a name, the label of a node, and and k is an integer, the index of an argument in the node. *path* identifies a node in the binding of i. The execution sets the node to the binding of j.
5. *Return exp*: The execution returns expression *exp*.
6. *ATable* k c b i *branches*: k is an integer uniquely identifying the table within a function, c is the number of branches in the table, b is either *flex* or *rigid* telling whether or not to narrow the inductive variable, i is the inductive variable, *branches* is a list of $(sym, stmt)$ pairs where sym is a constructor symbol and *stmt* a list of statements. If the root of the binding of i is a constructor c, then there exists a branch (c, s) and the execution continues with statements s.

2.8 function

In ICurry a function holds executable code. Each function of a source Curry program is translated into an ICurry a function. Some ICurry functions originate from other constructs such as case and let blocks.

An ICurry function holds 3 pieces of information: n the function name, i the function arity, *stmt* a sequence of one or more executable statements the last of which returns a value.

Ex:

```
function "bintree.isin" 2
  code
    declare_var 1 (ILhs (("bintree","isin"),1))
    declare_var 2 (ILhs (("bintree","isin"),2))
    declare_var 6 ICase
    assign 6
      reference_var 2
    ATable 6 2 flex
      reference_var 6
      "bintree.Leaf" =>
        return
          Node "Prelude.False"
      "bintree.Branch" =>
        declare_var 3 (IVar 6 (("bintree","Branch"),1))
        declare_var 4 (IVar 6 (("bintree","Branch"),2))
        declare_var 5 (IVar 6 (("bintree","Branch"),3))
        return
          Node "Prelude.||" (
            Node "Prelude.==" (
              reference_var 1 ,
              reference_var 3 ) ,
            Node "Prelude.||" (
              Node "bintree.isin" (
                reference_var 1 ,
                reference_var 4 ) ,
              Node "bintree.isin" (
                reference_var 1 ,
                reference_var 5 ) ) )
```
(9)

Index