Meta-Variant Translators for Icon

Ralph E. Griswold

Department of Computer Science, The University of Arizona

Variant translators provide a system for constructing robust preprocessors for Icon programs [1]. The variant translator specification system makes it easy to specify changes to programs, such as the one for modeling string scanning [2]:

expr1 ? *expr2* ➛ Escan(Bscan(*expr1*), *expr2*)

A single specification does the trick and works regardless of the complexity of the expressions involved:

```
Bques(x, y, z)   "Escan(Bscan("   x "),"   z   ")"
```

Writing such specifications is fairly easy, once you learn a few rules. However, if a variant translation is complicated, its specification may be tedious to construct and prone to error. Furthermore, such specifications aren't feasible for specialized variant translations, such as for translating one procedure name differently from all other procedure names. Such translations can be accomplished by using C functions, but the kind of code that is needed is tedious to write, hard to modify, and requires proficiency in C.

This report describes a higher-level approach for producing variant translators, called meta-variant translators, that allows variant translations to be written in Icon instead of C.

An ordinary variant translator translates an Icon program into another Icon program, as shown in Figure 1. To change the translation, it is necessary to change the translation specifications and build a new variant translator, vt, as indicated by the asterisk.

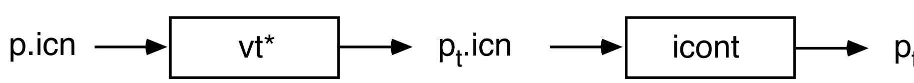

Figure 1. Variant Translation

A meta-variant translator translates an Icon program into another Icon program, which is translated and linked with a library of code-generation procedures, gen.icn, to produce the final Icon program, as shown in Figure 2.

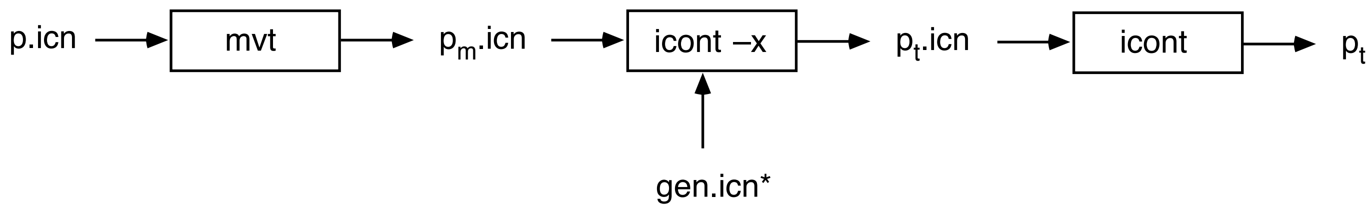

Figure 2. Meta-variant translation

The standard version of gen.icn contains procedures that perform an "identity" translation, so that the output is same to the input except for layout. A variant translation is accomplished by making changes to the code-generation procedures in gen.icn, rather than by changing the variant translator — that is, by using a variant gen.icn. Note that changing the translation does not require changing mvt.

As an example of identity translation, consider the simple program

```
procedure main()
  while line := read() do
    line ? process()
end
```

The output of mvt for this program is a procedure tp_() (for target program):

```
procedure tp_()
Proc_("main",)
Reduce_(While_Do_(Asgnop_(":=",Var_("line"),
Invoke_(Var_("read"),Null_())),Scan_(Var_("line"),
Invoke_(Var_("process"),Null_()))),)
End_()
end
```

The procedure calls correspond to syntactic components of the original Icon program. For example, Proc_() is a procedure corresponding to a procedure declaration in the original program, and While_Do_() is a procedure corresponding to the while-do loop. (The procedure Reduce_() writes the result returned by its argument.)

The code for these procedures is contained in gen.icn. For example, the code for the identity translation of while-do is

```
procedure While_Do_(e1, e2)
  return "while " || e1 || " do " || e2
end
```

The complete identity translator is listed in the appendix.

The result of linking gen.icn with the code generated by mvt and executing the result is

```
procedure main()
while (line := read()) do (line ? process())
end
```

This is equivalent to the original program; only the layout has been changed.

To see how meta-variant translators can be done, consider the string-scanning translation given at the beginning of this article. The identify translation for string scanning is

```
procedure Scan_(e1, e2)
  return "(" || e1 || " ? " || e2 || ")"
end
```

To get the variant translation for modeling string scanning, it's only necessary to change this procedure to

```
procedure Scan_(e1, e2)
  return "Escan(Bscan(" || e1 || ")," || e2 || ")"
end
```

As another example, suppose you want a variant translation to convert calls of map() to calls of Map() so that you can trace the function map() by providing a procedure Map() that does the same thing. The declaration for Map() might be

```
procedure Map(s1, s2, s3)
  return map(s1, s2, s3)
end
```

The variant translation can be accomplished by adding the line

```
if e0 == "map" then e0 := "Map"
```

at the beginning of the procedure Invoke_() as shown in the appendix.

It's also necessary to get the procedure declaration for Map() into the final program. This can be done by adding the following lines to the beginning or end of main() in gen.icn:

```
write("procedure Map(s1, s2, s3)")
write("   return map(s1, s2, s3)")
write("end")
```

An alternative approach, which is more desirable in the case of more elaborate variant translations of this type, is to add

```
write("link libe")
```

at the beginning or end of main() in gen.icn and provide the code to be linked with the final program in libe.icn.

If you want to trace a generator, be sure to use suspend instead of return; otherwise your procedure won't produce all the results produced by the generator. For example, for seq(), the procedure would be

```
procedure Seq(i1, i2)
  suspend seq(i1, i2)
end
```

In fact, it doesn't hurt to use suspend for functions that aren't generators.

Conclusions

Meta-variant translators allow you to write variant translators in Icon. The job is so easy that all kinds of things are worth doing that you'd probably not consider with standard variant translators and C.

There are, however, a few potential problems with meta-variant translators. Producing a translation, once you have gen.icn the way you want it, is somewhat more complex and slightly slower than for standard variant translators. The complexity can be hidden in a script and the loss in translation speed generally is insignificant, given the amount of *programming* time that it takes to craft a standard variant translator, as opposed to a meta-variant one.

A more serious problem is the amount of memory required to build the Icon program that produces the final translation. As illustrated above, the output of mvt is considerably larger than the input to mvt. If the input to mvt is a large program, the output is a huge one. It's also necessary to link gen.icn with the output of mvt, adding to the size of the intermediate program. The memory needed usually is not a problem on platforms in the workstation class, but it certainly can be on personal computers.

As mentioned above, it's not necessary to change mvt (a standard variant translator) to change the variant translation. This is true as long as the input language is standard Icon. If the input language is different from Icon, as say, in the variant translator Seque for [3], then a different version of mvt is needed. If the output language is different from Icon, as it is in the translation of Rebus to SNOBOL4 [4], then a considerably different version of gen.icn may be needed.

Getting Meta-Variant Translators

The meta-variant translator system described here is contained in the UNIX distribution of Version 9 of Icon. See /icon/tests/vtran/meta. Meta-variant translators also are available by anonymous FTP to cs.arizona.edu; cd /icon/meta and get READ.ME to see what to do next.

References

1. *Variant Translators for Version 9.0 of Icon*, Ralph E. Griswold, Icon Project document IPD245, Department of Computer Science, The University of Arizona, 1994.

2. "Modeling String Scanning", *Icon Analyst* 6, pp. 1-2.

3. "Lost Languages — Seque", *Icon Analyst* 19, pp. 1-4.

4. "Lost Languages — Rebus", *Icon Analyst* 18, pp. 1-4.

Appendix — Identity Meta–Variant Translator

```
#  main() calls tp_(), which is produced by the meta–variant
#  translation.

procedure main()
  tp_()
end

procedure Alt_(e1, e2)                      # e1 | e2
  return "(" || e1 || "|" || e2 || ")"
end

procedure Apply_(e1, e2)                    # e1 ! e2
  return "(" || e1 || "!" || e2 || ")"
end

procedure Arg_(e)
  return e
end

procedure Asgnop_(op, e1, e2)               # e1 op e2
  return "(" || e1 || " " || op || " " || | e2 || ")"
end

procedure Augscan_(e1, e2)                  # e1 ?:= e2
  return "(" || e1 || " ?:= " || e2 || ")"
end

procedure Bamper_(e1, e2)                   # e1 & e2
  return "(" || e1 || " & " || e2 || ")"
end

procedure Binop_(op, e1, e2)                # e1 op e2
  return "(" || e1 || " " ||  op || " " ||  e2 || ")"
end

procedure Break_(e)                         # break e
  return "break " || e
end
```

```
procedure Case_(e, clist)                # case e of { caselist }
  return "case " || e || " of {" || clist || "}"
end

procedure Cclause_(e1, e2)               # e1 : e2
  return e1 || " : " || e2 || "\n"
end

procedure Clist_(e1, e2)                 # e1 ; e2 in case list
  return e1 || ";" || e2
end

procedure Clit_(e)                       # 's'
  return "'" || e || "'"
end

procedure Compound_(es[])                # { e1; e2; ... }
  local result
  if *es = 0 then return "{}\n"
  result := "{\n"
  every result ||:= !es || "\n"
  return result || "}\n"
end

procedure Create_(e)                     # create e
  return "create " || e
end

procedure Default_(e)                    # default: e
  return "default: " || e
end

procedure End_()                         # end
  write("end")
  return
end

procedure Every_(e)                      # every e
  return "every " || e
end

procedure Every_Do_(e1, e2)              # every e1 do e2
```

```
  return "every " || e1 || " do " || e2
end

procedure Fail_()                    # fail
  return "fail"
end

procedure Field_(e1, e2)             # e . f
  return "(" || e1 || "." || e2 || ")"
end

procedure Global_(vs[])              # global v1, v2, ...
  local result
  result := ""
  every result ||:= !vs || ", "
  write("global ", result[1:– 2])
  return
end

procedure If_(e1, e2)                # if e1 then e2
  return "if " || e1 || " then " || e2
end

procedure If_Else_(e1, e2, e3)       # if e1 then e2 else e3
  return "if " || e1 || " then " || e2 || " else " || e3
end

procedure Ilit_(e)                   # i
  return e
end

procedure Initial_(s)                # initial e
  write("initial ", s)
  return
end

procedure Invocable_(es[])           # invocable ... (not handled properly in mvt)
  if \es then write("invocable all")
  else write("invocable ", es)
  return
end
```

```
procedure Invoke_(e0, es[])                    # e0(e1, e2, ...)
  local result

  if *es = 0 then return e0 || "()"

  result := ""
  every result ||:= !es || ", "

  return e0 || "(" || result[1:-2] || ")"

end

procedure Key_(s)                              # &s

  return "&" || s

end

procedure Limit_(e1, e2)                       # e1 \ e2

  return "(" || e1 || "\\" || e2 || ")"

end

procedure Link_(vs[])                          # link "v1, v2, ..."

  local result

  result := ""
  every result ||:= !vs || ", "

  write("link ", result[1:-2])

  return

end

procedure List_(es[])                          # [e1, e2, ... ]
  local result

  if *es = 0 then return "[]"

  result := ""
  every result ||:= !es || ", "

  return "[" || result[1:-2] || "]"

end

procedure Local_(vs[])                         # local v1, v2, ...
  local result

  result := ""
  every result ||:= !vs || ", "

  write("local ", result[1:-2])

  return

end

procedure Next_()                              # next
```

```
  return "next"

end

procedure Not_(e)                        # not e

  return "not(" || e || ")"

end

procedure Null_()                        # &null

  return ""

end

procedure Paren_(es[])                   # (e1, e2, ... )
  local result

  if *es = 0 then return "()"

  result := ""
  every result ||:= !es || ", "

  return "(" || result[1:-2] || ")"

end

procedure Pdco_(e0, es[])                # e0{e1, e2, ... }
  local result

  if *es = 0 then return e0 || "{}"

  result := ""
  every result ||:= !es || ", "

  return e0 || "{" || result[1:-2] || "}"

end

procedure Proc_(s, es[])                 # procedure s(v1, v2, ...)
  local result, e

  if *es = 0 then write("procedure ", s, "()")

  result := ""
  every e := !es do
    if \e == "[]" then result[-2:0] := e || ", "
    else result ||:= (\e | "") || ", "

  write("procedure ", s, "(", result[1:-2], ")")

  return

end

procedure Record_(s, es[])               # record s(v1, v2, ...)
  local result, field

  if *es = 0 then write("record ", s, "()")

  result := ""
```

```
  every field := !es do
    result ||:= (\field | "") || ", "
  write("record ", s, "(", result[1:– 2], ")")
  return
end

procedure Reduce_(s[])                    # used in code generation
  every write(!s)
  return
end

procedure Repeat_(e)                      # repeat e
  return "repeat " || e
end

procedure Return_(e)                      # return e
  return "return " || e
end

procedure Rlit_(e)
  return e
end

procedure Scan_(e1, e2)                   # e1 ? e2
  return "(" || e1  || " ? " || e2 || ")"
end

procedure Section_(op, e1, e2, e3)        # e1[e2 op  e3]
  return e1 || "[" || e2 || op || e3 || "]"
end

procedure Slit_(s)                        # "s"
  return image(s)
end

procedure Static_(ev[])                   # static v1, v2, ..
  local result
  result := ""
  every result ||:= !ev || ", "
  write("static ", result[1:– 2])
  return
end
```

```
procedure Subscript_(e1, e2)       # e1[e2]
  return e1 || "[" || e2 || "]"
end

procedure Suspend_(e)              # suspend e
  return "suspend " || e
end

procedure Suspend_Do_(e1, e2)      # suspend e1 do e2
  return "suspend " || e1 || " do " || e2
end

procedure To_(e1, e2)              # e1 to e2
  return "(" || e1 || " to " || e2 || ")"
end

procedure To_By_(e1, e2, e3)       # e1 to e2 by e3
  return "(" || e1 || " to " || e2 || " by " || e3 || ")"
end

procedure Repalt_(e)               # |e
  return "(|" || e || ")"
end

procedure Unop_(op, e)             # op e
  return "(" || op || e || ")"
end

procedure Until_(e)                # until e
  return "until " || e
end

procedure Until_Do_(e1, e2)        # until e1 do e2
  return "until " || e1 || " do " || e2
end

procedure Var_(s)                  # v
  return s
end

procedure While_(e)                # while e
```

```
  return "while " || e
end

procedure While_Do_(e1, e2)                # while e1 do e2
  return "while " || e1 || " do " || e2
end
```