ART memo 4 (version 0)
Problems with numerics in tracing with nhc98

Malcolm Wallace
University of York, UK

15th June 2000

1 The Float/Double problem

The tracing Prelude/Libraries do not support any operations on `Float`s or `Double`s. (The appropriate classes (`Real`, `Floating`...) are defined, but all instances have null implementations.)

It follows that literal `Float`s and `Double`s are also poorly supported for tracing. Remember however that non-integral literals are defined as being sugar for `(fromRational (n%d))`, where the chosen rational number preserves full precision of the value as a `Ratio Integer`. Thus, provided that `Integer`s and the `fromRational` method are supported, literal rationals are straightforwardly also supported - provided only that you can't do anything useful with them yet.

See the Proposals section for how the situation might be rectified.

2 The literal Integer problem

At present the nhc98comp tracing compiler treats `Integer`s specially, not in full accordance with the language specification. This section is an investigation of the current situation, identifying problems and proposing solutions.

When the tracing compiler was first written, integral numbers defaulted to fixed-width `Int`s in the absence of any further type information. Since then, the newer Haskell'98 Report mandates that integral numbers should default to multi-precision `Integer`s. This means that where previously most programs using integral numbers worked, most now fail.

There are two parts to the problem – one is the runtime representation of `Integer`s and the implementation of primitive operations on `Integer`s; the other is the treatment of literal `Integer`s as they appear in the user's source code.

2.1 Compiler

The compiler source files relevant to this investigation are

- STGBuild.hs
- STGGcode.hs
- FixSyntax.hs

These are easily identified from the Makefile as being the only files which differ (via CPP magic) between the tracing compiler and the normal compiler.

STGBuild.hs

```
#ifdef DBGTRANS
buildExp pu (PosInteger pos i) = oneHeap True pu (HEAP_INT (fromInteger i))
#else
buildExp pu (PosInteger pos i) = oneHeap True pu (HEAP_INTEGER i)
#endif
```

We can see that literal `Integer`s are down-cast to fixed-width `Int`s before code generation.

STGGcode.hs

```
#ifdef DBGTRANS
gAtom (PosInteger pos i) = incDepth >>> unitS [PUSH_INT (fromInteger i)]
#else
gAtom (PosInteger pos i) = incDepth >>> unitS [PUSH_INTEGER i]
#endif
```

Likewise here.

FixSyntax.hs

The following section of code is part of the normal compiler. It finds an expression of the form `(fromInteger {dict} x)` for a literal `x` and, depending on the context and type of the argument `x`, simplifies the whole expression to a simple literal of the correct type. If that is not possible, it passes the expression on (almost) unchanged.

```
--- fromInteger {Int Integer Float Double} constant
fsExp exp@(ExpApplication pos [v@(ExpVar _ qfromInteger)
                              ,(ExpDict v2@(Exp2 _ qNum qType))
                              ,l@(ExpLit pl (LitInteger b i))]) =
  fsState >>>= \ state ->
  if tidIS state qfromInteger == tfromInteger && tidIS state qNum == tNum
  then
    let id       = tidIS state qType
        return x = unitS (ExpLit pl x)
    in   if id == tInt        then return (LitInt b (fromInteger i))
    else if id == tIntHash    then return (LitInt UnBoxed (fromInteger i))
    else if id == tInteger    then unitS l
    else if id == tFloat      then return (litFloat Integer b i)
    else if id == tFloatHash  then return (litFloat Integer UnBoxed i)
    else if id == tDouble     then return (LitDouble b (fromInteger i))
    else if id == tDoubleHash then return (LitDouble UnBoxed (fromInteger i))
    else if id == tRational   then return (LitRational b (fromInteger i))
    else fsExp (ExpApplication pos [v,ExpDict (ExpApplication pos [v2]),l])
  -- Match (sel (class.type dicts) args)
  else fsExp (ExpApplication pos [v,ExpDict (ExpApplication pos [v2]),l])
```

Because the tracing compiler transforms the types of all expressions so that they calculate a trace in addition to a value, this piece of code is partially overridden by a preceding clause which is present only when tracing.

```
#ifdef DBGTRANS
fsExp exp@(ExpApplication p [ExpVar _ fci
                            ,ExpDict dict@(Exp2 _ qNum qType)
                            ,sr
                            ,t
                            ,l@(ExpLit p2 (LitInteger b i))]) =
  fsState >>>= \state ->
  if tidIS state fci == t_fromConInteger && tidIS state qNum == tNum then
      fsTidFun >>>= \tidFun ->
      if tidIS state qType == tInt then
          unitS (ExpApplication p [ExpVar p (tidFun (t_conInt, Var)), sr, t,
                                   ExpLit p (LitInt b (fromInteger i))])
      else if tidIS state qType == tInteger then
          unitS (ExpApplication p [ExpVar p (tidFun (t_conInteger, Var))
                                  , sr, t, l])
      else error ("fsExp: strange expr(1) at "++ strPos p)
  else if tidIS state fci == t_patFromConInteger then
      fsTidFun >>>= \tidFun ->
      if tidIS state qType == tInt then
          unitS (ExpApplication p [ExpCon p (tidFun (tR, Con)),
```

```
                                ExpLit p (LitInt b (fromInteger i)), t])
      else if tidIS state qType == tInteger then
          unitS (ExpApplication p [ExpCon p (tidFun (tR, Con)), l, t])
      else error ("fsExp: strange expr(5) at "++ strPos p)
    else error ("fsExp: strange expr(2) at " ++ strPos p ++
                "\nfci=" ++ show t_fromConInteger ++
                "\npfci=" ++ show t_patFromConInteger ++
                "\n?=" ++ show(tidIS state fci))
fsExp exp@(ExpApplication p [ExpVar _ fci
                             ,ExpDict dict@(ExpVar _ _)
                             ,sr
                             ,t
                             ,l@(ExpLit p2 (LitInteger b i))]) =
    fsState >>>= \state ->
    if tidIS state fci == t_fromConInteger then
        fsTidFun >>>= \tidFun ->
        unitS (ExpApplication p [ExpVar p (tidFun (t_ap 1, Var)),
                                 sr, t, ExpApplication p [ExpVar p (tidFun (tfromInteger, Var)),
                                                          dict, sr, t],
                                 ExpApplication p [ExpVar p (tidFun (t_conInteger, Var)),
                                                   sr, t, l]])
    else if tidIS state fci == t_patFromConInteger then error ("fsExp: strange expr(4) at "++strPos p)
    else error ("fsExp: strange expr(3) at "++ strPos p)
#endif
```

Comparing this with the normal code, it now matters whether we are looking at a *constructive* expression of the form (`fromConInteger {dict} SR T x`) or a *pattern* expression of the form (`patFromConInteger {dict} SR T x`).

In the constructive case, provided the context is `Num` and the type of the literal is either `Integer` or `Int`, a translation to a direct expression is given.

In the pattern case, the context is ignored and once again if the type is either `Int` or `Integer` a translation is given.

In the case of a constructive pattern with a definite (i.e. not variable) but non-`Num` context, we end up with an error rather than keeping the original expression (as one might have expected from the non-tracing case).

Looking at the second clause, when the context is non-definite, i.e. variable, it seems that we can translate constructive expressions to an application of the `t_ap1` tracing combinator, but we cannot deal with pattern expressions.

2.2 Runtime system

In the normal runtime system, numeric operations on almost all types are implemented as bytecode instructions in the virtual machine. There is special magic inside the compiler to recognise certain function applications and emit the appropriate bytecode instruction.

However for tracing it is not possible to take advantage of this mechanism because we want to use the standard virtual machine, yet the traced numeric ops have totally different types.

So, in the tracing runtime system, the file

- getconstr.c

holds extra tracing versions of most of the numeric primitives. These are written by hand. Here we see that

```
/* Num Integer -- Not correct!!! */
PRIM_INT_OP2(_tprim_IntegerLt,<,mkBool,mkNmBool)
PRIM_INT_OP2(_tprim_IntegerPlus,+,mkInt,mkNmInt)
PRIM_INT_OP2(_tprim_IntegerMinus,-,mkInt,mkNmInt)
PRIM_INT_OP2(_tprim_IntegerTimes,*,mkInt,mkNmInt)
```

etc. The macro `PRIM_INT_OP2` defines the primitive Haskell operations by application of the C operator to two fixed-width `Int`s. Clearly the comment is accurate in recognising that this is not adequate for multi-precision `Integer`s. We also note the following code:

```
C_HEADER(_tprim_FromInteger)
{
  fprintf(stderr, "_fromInteger not yet implemented.\n");
  exit(1);
}
```

which indicates that a runtime conversion of an Integer to another Integral type won't work in the general case, although it is possible that a more specific conversion function might be applied instead. As an example of the latter,

```
C_HEADER(_tprim_IntFromInteger)
```

implements the down-casting (truncation) of multi-precision `Integer`s to fixed-width `Int`s. However,

```
PRIM_NOT_IMPLEMENTED(DoubleFromInteger)
PRIM_NOT_IMPLEMENTED(FloatFromInteger)
```

shows that there are other numeric conversions still lacking.

3 Proposed solution

We pre-suppose that there exists a correct scheme for transforming primitive FFI functions to tracing variants thereof. (See separate memo.)

Floats and Doubles

- Whereas `Float` and `Double` operations are currently not defined at all in the tracing runtime system,
- we propose to use the primitive FFI as an interface to the standard C Float and Double operations.

Integers

- Whereas `Integer` operations are currently defined by hand, in C, in part of the tracing runtime system,
- and whereas such definitions actually implement fixed-width `Int`s not multiple-precision `Integer`s,
- we propose to convert to using the primitive FFI as an interface to the standard m-p `Integer` library.

Integers

- Whereas the FixSyntax phase currently has an incomplete set of conditions for patching literal `Integer`s in expressions to the actual type used,
- and whereas the default action here, in the absence of a matching simplification, is to fail rather than to continue without simplification,
- we propose to fill out the remaining useful conditions under which a simplification is possible,
- and to default to continuing with an unsimplified expression where simplification was not possible.