Implementation of Overloading [110.96 version]

Introduction

This document describes the implementation of overloading and overloading resolution in SML/NJ 110.97 (revised from 110.96). Initially the description will use the 110.96 representations, but these will evolve into modified 110.97 versions.

File paths are relative to base/.

Terminology

An overloaded operator is a variable bound in the pervasive environment to an OVLDvar structure. A primitive type, called an *indicator*, that is inferred for a given occurrence of an operator, will determine which *variant* of that operator is used at that occurrence (e.g. Int.+ for an occurrence of the overloaded "+" operator). If a primitive type is not inferred for an occurrence of an operator, the default variant will used. This process of determining which definite operator replaces an occurrence of an overloaded operator variable is called *overloading resolution*.

Declaring overloading

```
files: compiler/Parse/parse/sml.grm
```

The declaration form for overloadings is:

```
OVERLOAD id COLON ty AS exp_pa
```

where `exp_pa` is an "and"-separated sequence of expressions, but these will only be simple path expressions (e.g. `Int.+`).

Each element of the expression (path) list `exp_pa` determines a unique indicator type, which can be reduced to one of the primitive type constructors defined in compiler/ElabData/types/basictypes.{sig,sml}. These primitive types serving as "indicators" for resolving overloadings are:

```
intTy/intTycon
int32Ty/int32Tycon
int64Ty/int64Tycon
intinfTy/intinfTycon
wordTy/wordTycon
word8Ty/word8Tycon
word32Ty/word32Tycon
word64Ty/word64Tycon
realTy/realTycon  (= R64.real)
charTy/charTycon
stringTy/stringTycon
```

`realTy`, `charTy`, `stringTy` could split into multiple cases in the future.

Overloadings

```
files: system/smlnj/init/pervasive.sml
```

This file defines the pervasive environment, including all overload bindings.

```
overload ~ :   ('a -> 'a)
   as  Int.~ and I32.~ and I64.~ and CII.~
   and Word.~ and W8.~ and W32.~ and W64.~
    and R64.~
overload + :   ('a * 'a -> 'a)
  as  Int.+ and I32.+ and I64.+ and CII.+
  and Word.+ and W8.+ and W32.+ and W64.+
  and R64.+
overload - :   ('a * 'a -> 'a)
  as  Int.- and I32.- and I64.- and CII.-
  and Word.- and W8.- and W32.- and W64.-
  and R64.-
overload * :   ('a * 'a -> 'a)
  as  Int.* and I32.* and I64.* and CII.*
  and Word.* and W8.* and W32.* and W64.*
  and R64.*
(*
overload / : ('a * 'a -> 'a)
  as R64./
*)
val op / = R64./        (* temporary hack around overloading bug *)
overload div : ('a * 'a -> 'a)
  as  Int.div and I32.div and I64.div and CII.div
  and Word.div and W8.div and W32.div and W64.div
overload mod : ('a * 'a -> 'a)
  as  Int.mod and I32.mod and I64.mod and CII.mod
  and Word.mod and W8.mod and W32.mod and W64.mod
```

```
overload < :   ('a * 'a -> bool)
  as  Int.< and I32.< and I64.< and CII.<
  and Word.< and W8.< and W32.< and W64.<
  and R64.<
  and InlineT.Char.<
  and stringlt
overload <= :   ('a * 'a -> bool)
  as  Int.<= and I32.<= and I64.<= and CII.<=
  and Word.<= and W8.<= and W32.<= and W64.<=
  and R64.<=
  and InlineT.Char.<=
  and stringle
overload > :   ('a * 'a -> bool)
  as  Int.> and I32.> and I64.> and CII.>
  and Word.> and W8.> and W32.> and W64.>
  and R64.>
  and InlineT.Char.>
  and stringgt
overload >= :   ('a * 'a -> bool)
  as  Int.>= and I32.>= and I64.>= and CII.>=
  and Word.>= and W8.>= and W32.>= and W64.>=
  and R64.>=
  and InlineT.Char.>=
  and stringge
overload abs : ('a -> 'a)
  as Int.abs and I32.abs and I64.abs and CII.abs and R64.abs
```

The set of overloaded operators is fixed, though implementation dependent. This means that these are the only overoadings that will exist, and there is nothing changing in the derived information. The set of operators is fixed, their ground types are fixed, and the mapping from indicator types to the corresponding VALvars is fixed.

Overloading Classes

As defined in DefnRev, Appendix E.1, overloaded operators can be associated with "overloading classes" which are sets of primitive types that determine the different "resolutions" of the operator

```
Int = {intTy, int32Ty, int64Ty, intinfTy}
Word = {wordTy, word8Ty, word32Ty, word64Ty}
Real = {realTy}
Char = {charTy}
String = {StringTy}
WordInt = Word U Int   (U = union)
RealInt = Real U Int
Num = Word U Int U Real
NumTxt = Num U {charTy, stringTy}
```

The association of classes with overloaded operators is:

```
~, +, - * : Num
div, mod : WordInt
<, <=, >, >= : NumTxt
abs : RealInt
```

So only Num, WordInt, NumTxt, RealInt are associated with overloaded operators (so far).

For a given applied occurrence of an overloaded operator, resolution is based on a single primitive type determined by the type checker. This "indicator" type will be a member of the operator's overloading class. In cases where the type checker does not produce a unique indicator (the occurrence remains undetermined) then a "default" member of its overloading class is used to resolve the operator. The defaults for the relevant classes are all the same, namely intTy; intTy is the universal default indicator.

```
WordInt : intTy
RealInt : intTy
Num : intTy
NumTxt : intTy
```

The overload classes actually play no role in the overloading resolution algorithm. They are just a potentially useful notion in the "metatheory" of overloading resolution.

Note that none of the four "relevant" overloading classes is disjoint from the others. Indeed, Int is the intersection of them all.

Semantic representations

```
files: compiler/ElabData/types.{sig,sml}
```

```
and ovldSource
  = OVAR of S.symbol * SourceMap.region     (* overloaded variable *)
  | OINT of IntInf.int * SourceMap.region   (* overloaded int literal *)
  | OWORD of IntInf.int * SourceMap.region  (* overloaded word literal *)
  (* in future, may need to add real, char, string literals as sources *)

and tvKind
  = ...
  | OVLD of (* overloaded operator type scheme variable,
         * representing one of a finite set of ground type options *)
     {sources: ovldSource list,   (* name of overloaded variable or literal value *)
      options: ty list} (* potential resolution types *)
```

Concentrating on the overloaded variable (operator) case, the source identifier and its source region are recorded as arguments of the OVAR constructor of ovldSource. There can be multiple

sources as the overload tvars for multiple overloaded identifiers get unified. Example

```
(fn x => x < x; x * x)
```

where the type checker will unify the OVLD tyvar introduced for "<" with the tyvar for "x" and then that will get unififed with the OVLD tyvar for "*".

When two such OVLD tyvars are unified, the options are narrowed by intersecting the options of each tyvar, and the new sources are the union (concatenation) of the sources of the two variables.

The options actually play no essential role. In the end, the type inferred for an overloaded operator occurrence will either be a single primitive type, which will be used to *resolve* the operator, or it will remain an OVLD tyvar, in which case the operator will be resolved by its default type (which is always intTy).

When overloading resolution is performed the resulting primitive type (if unique; i.e. if a single element of the options matches) is used to resolve the operators at each of the sources. ** Actually, the options field of OVLD is irrelevant. If

Elaboration of overload declarations

```
files: compiler/Elaborator/elaborate/elabcore.sml
       compiler/ElabData/syntax/varcon.{sig,sml} [VALvar, OVLDvar]
       compiler/ElabData/types/types.{sig,sml}  [TYFUN]
       compiler/Elaborator/types/overload.sml  [Overload.matchScheme]
```

```
    and elabOVERLOADdec((id,typeScheme,exps),env,rpath,region) =
    (* exps are simple variables or paths, with monomorphic types;
     * typescheme is a type scheme with a single type variable parameter,
     * which matches the type of each exp *)
    let val (body,tyvars) = ET.elabType(typeScheme,env,error,region)
        val tvs = TS.elements tyvars (* ASSERT: length tyvars = 1 *)
        val scheme = (TU.bindTyvars tvs; TU.compressTy body;
              TYFUN{arity=length tvs, body=body})
        fun option (MARKexp(e,_)) = option e
          | option (VARexp(ref (v as VALvar{typ,...}),_)) =
          {indicator = Overload.matchScheme(scheme,!typ), variant = v}
          | option _ = bug "evalOVERLOADdec.option"
        val options =
        map (fn exp => option(#1(elabExp(exp,env,region)))) exps
        val ovldvar = OVLDvar{name=id,scheme=scheme,
                  options=options}
    in
        (OVLDdec ovldvar, SE.bind(id,B.VALbind ovldvar,SE.empty),
             TS.empty, no_updt)
    end
```

where

```
   val elabExp : Ast.exp * SE.staticEnv * region
                 -> Absyn.exp * TS.tyvarset * tyvUpdate
```

and for the paths occuring in overload decl the resulting Absyn.exp will be of the form VARexp(ref v, []), were v is the VarCon.VALvar obtained by looking up the path in the environment. From varcon.sml (structure VarCon):

```
(* from varcon.sml *)
datatype var
  = VALvar of                       (* ordinary variables *)
      {path : SP.path,
       typ : T.ty ref,
       btvs : T.tyvar list ref,
       access : A.access,
       prim : PrimopId.prim_id}
  | OVLDvar of                          (* overloaded identifier *)
      {name : S.symbol,
       options: {indicator: T.ty, variant: var} list,
       scheme: T.tyfun}
  | ERRORvar                            (* error variables *)
```

So the overloaded variable (id) is bound to an OVLDvar structure, which contains the mapping from indicator types to the VALvar (:VarCon.vars) obtained by elaborating the paths for each option. The overload class of the variable is just the domain of the options mapping.

The typeScheme part of the overload declaration is translated into a TYFUN (always of arity 1). This is then "matched" using Overload.matchScheme with the type of each option VALvar to return an "indicator" type, which will be a primitive type — one of the types in the maximal overload class NumTex.

Type inference process

```
files: compiler/Elaborator/types/overload.sml
       compiler/Elaborator/types/typecheck.sml
       compiler/Elaborator/types/unify.sml
```

```
       | VARexp(refvar as ref(OVLDvar _),_) =>
         (exp, olv_push (refvar, region, err region))
```

When the type checker encounters a VARexp whose variable is an OVLDvar, it calls olv_push to push the refvar onto the overloading stack for this call of the top-level TypeCheck.decType. olv_push is defined by

```
(* setup for recording and resolving overloaded variables and literals *)
val { pushv = olv_push, pushl = oll_push, resolve = ol_resolve } = Overload.new ()
```

The type of olv_push is given by [overload.sml, l. 19]:

```
pushv : VarCon.var ref * SourceMap.region * ErrorMsg.complainer -> Types.ty,
```

The type returned by pushv/olv_push is the type scheme for the overloaded variable instantiated with a fresh OVLD tyvar. This OVLD tyvar contains the source info for this particular VARexp (variable name and region) and a copy of the instance -→ variant mapping for this overloaded operator (which is the same for each occurrence, hence does not need to be reconstructed for each new OVLD tyvar!).

When unification involving this OVLD tyvar occurs, two things can happen:

1. If the OVLD tyvar is instantiated to a type [instTyvar, l. 427], the typechecker checks whether the instantiating type (ty') is in the options list of the tyvar, and if not immediately raises a Unify exception (which generates a type error message). The options list at this point contains a (potentially proper) subset of the overloading class of the operator that generated the OVLD tyvar (one of the sources). Only the first source, if there are multiple ones, is mentioned in the printed metatypevariable [BUG].
2. If the OVLD tyvar is being unified with another tyvar:
    a. if the other tyvar is also an OVLD, then new options (options') are defined as the intersection of the options of the old (i1) and new (i2) OVLD tyvars. If this intersection is nil [**IMPOSSIBLE!**], then Unify is raised and there is a type error reported. Otherwise a new OVLD is defined with the concatenation of the sources and the (non-nil) intersection of the options, and both the tyvars are assigned this new OVLD value.
    b. if the other tyvar is an OPEN, then its eq attribute is used to filter out non-equality elements of options (i.e. realTy), thus indirectly disqualifying realTy as an valid indicator and thereby possibly causing a type error later on when the tyvar is instantiated. If after filter out realTy, options are nil [**IMPOSSIBLE!**], Unify is raised, otherwise a new OVLD is constructed with the same sources and the filtered options and assigned to the old and new tyvar refs.

If an OPEN tyvar is instantiated to the OVLD tyvar, then subcase OVLD [unify.sml, l. 184] of the iter function is invoked. This "propagates" (incorrectly) a true equality type attribute of the instantiated OPEN tyvar to the OVLD tyvar by filtering out non-equality types (which could only be realTy) from the options of the OVLD tyvar. This will cause a type error if later the OVLD tyvar is instantiated to realTy (or any other random type not in the relevant overloading class, but the type error will not be properly blamed on the equality mismatch [**BUG**].

Side Note: The `CONty(DEFtyc⋯)` case of `iter` [unify.sml, l. 196] is bogus and needs to be rewritten to expand definitions incrementally as the instantiating type is traversed, in order to deal with nonstrict (irrelevant) arguments and imbedded OBJ constructors properly.

Overloading resolution

```
files: compiler/Elaborator/types/typecheck.sml
       compiler/Elaborator/types/overload.sml
```

An ordinary variable expression elaborates to a VARexp(ref v, []) where v is the variable representation returned by looking up the variable path in the static environment. If v is an OVLDvar, the ref allows it to be replaced by the resolved variant, a VALvar, to effect overloading resolution.

During typechecking [typecheck.sml, l. 526], the VarCon.OVLDvar ref is pushed onto a overloaded variable stack created by the call to Overload.new [typecheck.sml, l. 75] for that TypeCheck.decType call. A similar thing happens for overloaded literals (oll_push), but we are not concerned with that at the moment.

After the normal type checking is completed, ol_resolve is called and it iterates through the pushed var refs and attempts to resolve and replace each one, based on an indicator type extracted to the (possibly) instantiated type of the VARexp determined by the type checker. Note that OVLD tyvars may have been unified, so more than one of the overloaded operators may be resolved by the same type instantiation.

There are three cases for OVLD tyvar.

1. The OVLD tyvars may have been instantiated to a primitive type belonging to the appropriate class, in which case overloading is resolved and the VARexp ref is updated with the corresponding VALvar.
2. The OVLD tyvar is instantiated to a type that is not in the overloading class for an operator, in which case resolution is unsuccessful and there is a type error (but reported during resolution, or earlier?).
3. The OVLD tyvar is not instantiated (but may have been modified by unification with other variables. In this case, the operator is resolved to the default variant, which will *always* be intTy, which is an equality type. Thus default resolution will always be possible, even if the OVLD tyvar's equality attribute has been turned on during unification! Default resolution cannot cause a equality mismatch type error **[INVARIANT]**.