Implementation of Overloading [110.97 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 AS exp_pa
```

where `exp_pa` is an "and"-separated sequence of expressions, which will all be simple path expressions such as `Int.+`.

Each element of the path expression list `exp_pa` specifies one of the *variants* of the overloaded identifier. Each will be associated with a different primitive type, called the *indicator* type. These primitive types are defined in `compiler/ElabData/types/basictypes.{sig,sml}`. The 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, for instance, `realTy` = `real64Ty` and `real32Ty`. This would lead to additional operators (e.g. `/` for reals, `@` for strings) being added to the set of overloaded operators.

Overloadings

```
files: system/smlnj/init/pervasive.sml  (global overloadind declarations)
       compiler/ElabData/types/types.{sig,sml}  (OVLDV tyKind)
compiler/ElabData/syntax/varcon.{sig,sml} (datatype var, VALvar, OVLDvar)
```

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

```
overload ~
   as  Int.~ and I32.~ and I64.~ and CII.~
   and Word.~ and W8.~ and W32.~ and W64.~
    and R64.~
overload +
  as  Int.+ and I32.+ and I64.+ and CII.+
  and Word.+ and W8.+ and W32.+ and W64.+
  and R64.+
overload -
  as  Int.- and I32.- and I64.- and CII.-
  and Word.- and W8.- and W32.- and W64.-
  and R64.-
overload *
  as  Int.* and I32.* and I64.* and CII.*
  and Word.* and W8.* and W32.* and W64.*
  and R64.*
overload div
  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
  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 <
  as  Int.< and I32.< and I64.< and CII.<
  and Word.< and W8.< and W32.< and W64.<
  and R64.<
  and InlineT.Char.<
  and stringlt
overload <=
  as  Int.<= and I32.<= and I64.<= and CII.<=
  and Word.<= and W8.<= and W32.<= and W64.<=
  and R64.<=
  and InlineT.Char.<=
  and stringle
overload >
  as  Int.> and I32.> and I64.> and CII.>
  and Word.> and W8.> and W32.> and W64.>
  and R64.>
  and InlineT.Char.>
  and stringgt
overload >=
  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
  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 (currently) 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 (derived from the variant paths) is fixed. For instance, the indicator type for integer addition is intTy and it will be associated with the VALvar derived by elaborating the path expression `Int.+`.

Overloading Classes

```
files: compiler/Elaborator/types/overloadclasses.sml (type class, class defs)
       compiler/ElabData/types/types.{sig,sml} (OVLDV tvKind)
```

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}        (intClass)
Word = {wordTy, word8Ty, word32Ty, word64Ty}     (wordClass)
Real = {realTy}                                  (realClass)
Char = {charTy}
String = {StringTy}
IntWord = Int U Word   (U = union)               (int_wordClass)
IntReal = Int U Real                             (int_realClass)
Num = IntWord U Real                             (numClass)
Text = Char U String                             (textClass)
NumText = Num U Text                             (num_textClass)
```

The classes associated with the overloaded operators are:

```
~, +, - * : Num
div, mod : IntWord
<, <=, >, >= : NumText
abs : RealInt
```

So only Num, IntWord, NumText, IntReal are (currently) associated with overloaded operators.

For a given applied occurrence of an overloaded identifier, resolution is based on the indicator type determined by the type checker.

Each overloaded identifier has a overloading *type scheme* which characterizes the types of its variants. For instance, the type scheme for identifier `+` is 'a * 'a → 'a. During type checking, the initial type assigned to an occurrence of the overloaded identifier is a fresh instantiation of its type scheme replacing the scheme variable 'a with an overloading type metavariable (or univariable) of the form

```
tv = ref(OVLDV{eq=false,variants=<variant VALvars>)} : tyvar
```

The variants field will be invariant and the variants are determined by the overload declaration for the identifier.

The indicator type is derived from the type of the occurrence inferred ty the type checker. That type will be either:

1. an instantiation of the tyvar tv, which will be compared with the members of the identifier’s overload class, and if it does not match any of them, produces a type error, or
2. if tv remains uninstantiated or is instantiated to another OVLDV tyvar, and thus does not determine a unique indicator type, the default indicator is used.

By convention, the default indicator is the "first" element of the identifier’s overloading class, represented as a list.

```
default(IntWord) : intTy = hd(int_wordClass)
default(IntReal) : intTy = hd(int_realClass)
default(Num)     : intTy = hd(numClass)
default(NumText) : intTy = hd(num_textClass)
```

Note that, currently, none of the four "relevant" overloading classes is disjoint from the others. Indeed, `intTy` is the intersection of them all. This would change if additional real (float) types are added (e.g. real32 or float32) with division "/" being overloaded for all the real types. Then the overloading class for "/" would be disjoint from WordInt, and would have a different default type.

Semantic representations

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

```
and tvKind
  = ...
  | OVLDV of
    {eq: bool,  (* equality attribute, may be set by unification *)
     sources: varSource list} (* names and locations of overloaded variables *)
     (* used to instantiate overloaded operator type scheme,
      * representing one of a finite set of possible ground types used as
      * "indicator" types to resolve the overloading *)
  | OVLDI of litSource list  (* overloaded integer literal *)
  | OVLDW of litSource list  (* overloaded word literal *)
  ...
withtype tyvar = tvKind ref
```

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 unified with the OVLD tyvar for "**". This unified OVLDV tvKind will have two sources: the occurrences of `<` and** in the expression.

The sources actually play no essential role, but provide information that can be used in diagnostic messages for type errors.

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,exps),env,rpath,region) =
    (* exps are simple variable paths, with monomorphic types that
     * are ground instances of the known typeScheme for id *)
    let fun getVar exp =
        (case exp
           of VARexp(ref(v),_) => v
            | MARKexp(e,_) => getVar e
            | _ => bug "evalOVERLOADdec.getVar")
        val val_vars = map (fn exp => getVar(#1(elabExp(exp,env,region)))) exps
        val ovldvar = OVLDvar{name = id, variants = val_vars}
    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 (e.g. `Int.+`) 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,           (* name of the overloaded operator *)
       variants : var list}       (* variant variables (VALvars) *)
  | ERRORvar                      (* error variables *)
```

So in elabOVERLOADdec, the overloaded variable (id) is bound to an OVLDvar structure, which contains the identifier (name) and the list of VALvars (:VarCon.var) obtained by elaborating the paths for each variant.

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. 16]:

```
 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 OVLDV 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. The OVLD tyvar is instantiated to a type [instTyvar, l. 427], which will be used during the overloading resolution phase of type checking to either resolve the overloading to a single variant, or signal a type error, if the type is not in the overloading class of the overloaded identifier.
2. The OVLDV tyvar is unified with another tyvar, in which case:
    a. If the other tyvar is also an OVLDV, then the two OVLDV tyvars are merged into a single OVLDV whose sources are the concatenation of the original sources, and the equality attribute is propagated. [Here a check could be implemented as to whether the sources were "compatible", meaning their overloading classes overlapped. Currently this would have no effect, since all relevant classes overlap. It would also be redundant, since any incompatibility would show up during resolution.]
    b. If the other tyvar is an OPEN, then it is instantiated to to the OVLDV type with propagation of the eq attribute from the two unified tyvars (i.e. eq = eq1 orelse eq2).
    c. If the other tyvar is an OVLDI or OVLDW (integer or word literal), the OVLDV variable is instantiated to it (i.e. OVLDI and OVLDW *have precedence* over OVLDV tyvars. This is because literal overloading resolution is performed before variable overloading resolution.

Overloading resolution

```
files: compiler/Elaborator/types/typecheck.sml
       compiler/Elaborator/types/overloadclasses.sml (OverloadClasses)
     compiler/Elaborator/types/overloadvar.sml (OverloadVar)
       compiler/Elaborator/types/overload.sml (Overload)
```

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 in its variant list, to effect overloading resolution.

During typechecking [typecheck.sml, l. 526], the VarCon.OVLDvar ref is pushed (olv_push) 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'll return to overloaded literals below. `ovl_push` returns a type that is an instantitation of the overload type scheme for the variable with a new `VARty(ref(OVLDV{…})`. The scheme for the variable is obtained [overload.sml, pushvar] by calling OverloadVar.symToScheme to the variable name. This function is defined in OverloadVar [overloadvar.sml], and it is based on a table relating each overloaded symbol to its type scheme and its overloading class (defined in OverloadClasses, [overloadclasses.sml])

After the first phase of type checking is completed, ol_resolve is called to perform the overloading resolution phase. This function iterates through the pushed var refs (in reverse order, so the order in which they were pushed), and attempts to resolve and replace each one. This is performed by resolveOVLDvar, using the context type (the original VARty(tyvar) returned for this variable by pushvar). If the context type is still an uninstantiated OVLD tyvar, the indicator is the default type from the class, otherwise, it is a head-reduced version of the instantiation. This indicator is used to lookup the corresponding variant through the function OverloadVar.resolveVar. This function scans the elements of the class in parallel with the variants, looking for a match for the indicator and returning the corresponding variant. This works because the class list and the variant list are coordinated, meaning the nth element of the class is the indicator type for the nth variant. If the indicator type is not found in the class, a type error is reported.

The type scheme and overloading class for each overloaded identifier are available through a table, `overloadTable` defined internally in OverloadVar. It is a list of 11 triples of type

```
type entry = S.symbol * T.tyfun * OLC.class
```

essentially an association list mapping symbols to their scheme and class. This information is fixed, and so does not need to be extracted from the overload declarations. The only essential information in an overload declaration for an identifier is the VALvar's for the variants. These cannot be precomputed because they involve dynamic access information generated during the bootstrapping of the compiler.

Resolving overloaded literals

Overloading literals is simpler because the only information needed is the context type, which must be a member of the appropriate overloading class for a given literal. For instance, and int expression like `3` must be resolved to one of the possible integer types in class Int.

Note: There can be interaction between the resolving of one overloaded operator and another, if their OVLD tyvar is shared. If the shared OVLD tyvar is not instantiated before resolution, the first operator resolved will instantiate it to the default indicator type. This may cause a clash with the resolution of the second operator if it is not compatible with the default type for the first operator. For instance, if the two operators are `mod` (default `intTy`) and `/` (assuming it is overloaded at multiple real types), then the resolution of `mod` will instantiate the shared tyvar to `intTy`, which will not be a valid option for resolving the `/` operator.

Adding new overloaded identifiers

```
files : compiler/Elaborator/types/overloadclasses.sml
        compiler/Elaborator/types/overloadvar.sml
        compiler/Elaborator/types/unify.sml
```

Adding new overloaded identifiers that fit in with the existing overloadings, i.e. by having one of the existing overloading classes, is straightforward. One only needs to add 3-tuples (rows) for them to the overloadTable in OverloadVar and make sure that the order in which variants are specified in the overload declaration is consistent with the order to their indicator types in their overloading class.

Let us suppose we want to add an overload operator that has a new overloading class. For instance, we might overload the division operator / on multiple real types. The new overloading class would be

```
Real' = {realTy, real32Ty}
```

and the overload declaration would look like:

```
overload / as Real./ and Real32./
```

indicating that the operation defaults to Real./. The type scheme would be the existing binaryScheme defined in OverloadVar.

Note that the existing overloading would presumably be extended to include Real32 variants (e.g. Real32.~, Real32.+, etc.).

Overloading literals for this class would involve adding a new constructor to Types.tvKind:

```
and tvKind
  = INSTANTIATED of ty (* instantiation of an OPEN *)
  | OPEN of
     {depth: int, eq: bool, kind: openTvKind}
  | UBOUND of (* explicit type variables *)
     {depth: int, eq: bool, name: S.symbol}
  | OVLDV of
    {eq: bool,  (* equality attribute, may be set by unification *)
     sources: varSource list} (* names and locations of overloaded
  | OVLDI of litSource list  (* overloaded integer literal *)
  | OVLDW of litSource list  (* overloaded word literal *)
  | OVLDR of realLitSource list (* new overloaded real literal *)
  | LBOUND of {depth: int, eq: bool, index: int}
```

where `realLitSource` would be a type for appropriatly describing source occurrences of the real literals.

The treatment of OVLDR tyvars would be analagous to the treatment of the existing OVLDI and OVLDW varieties (in Unify), except that these tyvars would be assumed to not have the equality attribute and there would be additional tests in Unify to detect equality clashes (e.g. where an OPEN tyvar with eq = true is unified with an OVLDR tyvar).

Printing type metavariables

A new way of printing type metavariables has been introduced. Metavariables are denoted by square brackets "[..]" with descriptive information between the brackets. The descriptive information indicates the form of the metavariable, the equality attribute (if true), and an internally generated "name" (Z,Y,X,...) to help identify multiple occurrences of the same metavariable.

Here are some examples:

```
OPEN{eq=false,...}: [Z]  (META or FLEX)
```

```
OPEN{eq=true,...} : [Z:eq]
```

```
OVLDI :  [Z:INT]   (representing int literal overloading)
```

```
OVLDW :  [Z:WORD]  (representing int literal overloading)
```

```
OVLDV :  [Z:OL(+,*)] or [Z:OL(+,*):eq], where +,* are the operators involved
```

```
LBOUND : [Z:LB] or [Z:LB:eq]
```

```
UBOUND : □a  or □□a  (where □a□ is the name of the explicitly bound type variable)
```

Additional information, e.g. depth, is printed in debugging mode. For OVLDI and OVLDW, the value of the literal(s) could be added.