MLton Guide (20061107)

MLton Guide

This is the guide for MLton, an open-source, whole-program, optimizing Standard ML compiler.

This guide was generated automatically from the MLton wiki, available online at http://mlton.org. It is up to date for MLton 20061107.

Home

What is MLton?

MLton is an open-source, whole-program, optimizing Standard ML compiler.

What's new?

- See the slides from the talk on MLton given at the ML Workshop on September 16, 2006.

Next steps

- Read about MLton's Features.
- Look at Documentation.
- See some Users of MLton.
- Download MLton.
- Meet the MLton Developers.
- Get involved with MLton Development.
- User-maintained FAQ.
- Contact us.

Last edited on 2006-09-26 19:14:28 by StephenWeeks.

Index

There are 314 pages.

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | R | S | T | U | V | W | X | Z

A

AccessControl
AdmitsEquality
Alice
AllocateRegisters
AndreiFormiga
ArrayLiteral
AST

B

BasisLibrary
Bug
Bugs20041109
Bugs20051202

C

CallGraph
CallingFromCToSML
CallingFromSMLToC
CallingFromSMLToCFunctionPointer
ChrisClearwater
Chunkify
CKitLibrary
Closure
ClosureConvert
CMinusMinus
CommonArg
CommonBlock
CommonSubexp
CompilationManager
CompilerOverview
CompileTimeOptions
CompilingWithSMLNJ
ConcurrentML
ConcurrentMLImplementation
ConstantPropagation
Contact
Contify
CoreML

CoreMLSimplify
CreatingPages
Credits
CrossCompiling

D

DeadCode
DeepFlatten
DefineTypeBeforeUse
DefinitionOfStandardML
Defunctorize
Developers
Development
Documentation
Drawbacks

E

Eclipse
EditingPages
Elaborate
Emacs
Enscript
EqualityType
EqualityTypeVariable
EtaExpansion
eXene
Experimental

F

FAQ
Features
FirstClassPolymorphism
Fixpoints
Flatten
Fold
Fold01N
ForeignFunctionInterface
ForeignFunctionInterfaceSyntax
ForeignFunctionInterfaceTypes
ForLoops
FrontEnd
FunctionalRecordUpdate
fxp

G

GarbageCollection
GenerativeDatatype
GenerativeException
Glade
Globalize
GnuMP

H

HaMLet
HenryCejtin
History
Home
HowProfilingWorks

I

Identifier
Immutable
ImperativeTypeVariable
ImplementExceptions
ImplementHandlers
ImplementProfiling
ImplementSuffix
Index
InfixingOperators
Inline
InsertLimitChecks
InsertSignalChecks
Installation
IntermediateLanguage
IntroduceLoops

J

JesperLouisAndersen
JohnnyAndersen

K

KnownCase

L

LambdaCalculus
LambdaFree
LanguageChanges
Lazy

Libraries
License
LineDirective
LLVM
LocalFlatten
LocalRef
LoopInvariant

M

Machine
ManualPage
MatchCompilation
MatchCompile
MatthewFluet
mGTK
MichaelNorrish
MikeThomas
ML
MLBasis
MLBasisAnnotationExamples
MLBasisAnnotations
MLBasisAvailableLibraries
MLBasisExamples
MLBasisPathMap
MLBasisSyntaxAndSemantics
MLj
MLKit
MLmon
MLNLFFI
MLNLFFIImplementation
MLRISCLibrary
MLtonArray
MLtonBinIO
MLtonCont
MLtonExn
MLtonFinalizable
MLtonGC
MLtonIntInf
MLtonIO
MLtonItimer
MLtonPlatform
MLtonPointer
MLtonProcEnv
MLtonProcess
MLtonProfile
MLtonRandom
MLtonRlimit
MLtonRusage
MLtonSignal
MLtonSocket

MLtonStructure
MLtonSyslog
MLtonTextIO
MLtonThread
MLtonVector
MLtonWeak
MLtonWord
MLtonWorld
MoinMoin
Monomorphise
MoscowML
Multi
Mutable

N

NumericLiteral

O

ObjectOrientedProgramming
OCaml
OpenGL
OperatorPrecedence
OptionalArguments
OrphanedPages
OtherSites
Overloading

P

PackedRepresentation
PageSize
ParallelMove
Performance
PhantomType
PlatformSpecificNotes
PolyEqual
PolyML
PolymorphicEquality
Polyvariance
Poplog
PortingMLton
PortStatus
PrecedenceParse
Printf
PrintfGentle
ProductType
Profiling
ProfilingAllocation

ProfilingCounts
ProfilingTheStack
ProfilingTime
Projects
Pronounce
PropertyList

R

RayRacine
Reachability
Redundant
RedundantTests
References
RefFlatten
Regions
ReleaseChecklist
RemoveUnused
Restore
Roadmap
RSSA
RSSAShrink
RSSASimplify
RunningOnCygwin
RunningOnDarwin
RunningOnFreeBSD
RunningOnLinux
RunningOnMinGW
RunningOnNetBSD
RunningOnOpenBSD
RunningOnPowerPC
RunningOnSolaris
RunningOnSparc
RunTimeOptions

S

ScopeInference
SelfCompiling
Serialization
ShowBasis
ShowProf
Shrink
SimplifyTypes
SMLNET
SMLNJ
SMLNJDeviations
SMLNJLibrary
SMLofNJStructure
Sources

SpaceSafety
SSA
SSA2
SSA2Simplify
SSASimplify
Stabilizers
StandardML
StandardMLBooks
StandardMLGotchas
StandardMLHistory
StandardMLImplementations
StandardMLPortability
StandardMLTutorials
StephenWeeks
StyleGuide
Subversion
SureshJagannathan
Survey
SurveyDone
Swerve
SXML
SXMLShrink
SXMLSimplify
SyntacticConventions
SystemInfo

T

Talk
TalkDiveIn
TalkFolkLore
TalkFromSMLTo
TalkHowHigherOrder
TalkHowModules
TalkHowPolymorphism
TalkMLtonApproach
TalkMLtonFeatures
TalkMLtonHistory
TalkStandardML
TalkWholeProgram
TILT
ToMachine
TomMurphy
ToRSSA
ToSSA2
TrustedGroup
TypeChecking
TypeConstructor
TypeIndexedValues
TypeVariableScope

U

Unicode
UniversalType
UnresolvedBugs
UnsafeStructure
Useless
Users
Utilities

V

ValueRestriction
VariableArityPolymorphism
Variant
VesaKarvonen

W

WantedPages
WebSite
WesleyTerpstra
WholeProgramOptimization
WikiMacros
WikiName
WikiTool

X

XML
XMLShrink
XMLSimplify
XMLSimplifyTypes

Z

Zone
ZZZOrphanedPages

AST

AST is the IntermediateLanguage produced by the FrontEnd and translated by Elaborate to CoreML.

Description

The abstract syntax tree produced by the FrontEnd.

Implementation

ast-programs.sig ast-programs.fun
ast-modules.sig ast-modules.fun
ast-core.sig ast-core.fun
ast

Type Checking

The AST IntermediateLanguage has no independent type checker. Type inference is performed on an AST program as part of Elaborate.

Details and Notes

Last edited on 2005-11-30 19:55:04 by StephenWeeks.

AccessControl

MoinMoin supports a lot of access control features.

Because people download binaries from the MLton web site, and we are worried about malicious users either changing those binaries, or changing the links that should point at those binaries, we allow editing of some pages (in particular, Download, Home, and Experimental) only by TrustedGroup members.

All other pages are freely editable by any user with an account.

Last edited on 2005-11-30 19:54:23 by StephenWeeks.

AdmitsEquality

A TypeConstructor admits equality if whenever it is applied to equality types, the result is an EqualityType. This notion enables one to determine whether a type constructor application yields an equality type solely from the application, without looking at the definition of the type constructor. It helps to ensure that PolymorphicEquality is only applied to sensible values.

The definition of admits equality depends on whether the type constructor was declared by a `type` definition or a `datatype` declaration.

Type definitions

For type definition

```
type ('a1, ..., 'an) t = ...
```

type constructor `t` admits equality if the right-hand side of the definition is an equality type after replacing `'a1`, ..., `'an` by equality types (it doesn't matter which equality types are chosen).

For a nullary type definition, this amounts to the right-hand side being an equality type. For example, after the definition

```
type t = bool * int
```

type constructor `t` admits equality because `bool * int` is an equality type. On the other hand, after the definition

```
type t = bool * int * real
```

type constructor `t` does not admit equality, because `real` is not an equality type.

For another example, after the definition

```
type 'a t = bool * 'a
```

type constructor `t` admits equality because `bool * int` is an equality type (we could have chosen any equality type other than `int`).

On the other hand, after the definition

```
type 'a t = real * 'a
```

type constructor `t` does not admit equality because `real * int` is not equality type.

We can check that a type constructor admits equality using an `eqtype` specification.

```
structure Ok: sig eqtype 'a t end =
   struct
      type 'a t = bool * 'a
   end
```

```
structure Bad: sig eqtype 'a t end =
   struct
      type 'a t = real * int * 'a
   end
```

On `structure Bad`, MLton reports the following error.

```
Type t admits equality in signature but not in structure.
  not equality: [real] * _ * _
```

The `not equality` section provides an explanation of why the type did not admit equality, highlighting the problematic component (`real`).

Datatype declarations

For a type constructor declared by a datatype declaration to admit equality, every variant of the datatype must admit equality. For example, the following datatype admits equality because `bool` and `char * int` are equality types.

```
datatype t = A of bool | B of char * int
```

Nullary constructors trivially admit equality, so that the following datatype admits equality.

```
datatype t = A | B | C
```

For a parameterized datatype constructor to admit equality, we consider each variant as a type definition, and require that the definition admit equality. For example, for the datatype

```
datatype 'a t = A of bool * 'a | B of 'a
```

the type definitions

```
type 'a tA = bool * 'a
type 'a tB = 'a
```

both admit equality. Thus, type constructor `t` admits equality.

On the other hand, the following datatype does not admit equality.

```
datatype 'a t = A of bool * 'a | B of real * 'a
```

As with type definitions, we can check using an `eqtype` specification.

```
structure Bad: sig eqtype 'a t end =
   struct
      datatype 'a t = A of bool * 'a | B of real * 'a
   end
```

MLton reports the following error.

```
Type t admits equality in signature but not in structure.
  not equality: B of [real] * _
```

MLton indicates the problematic constructor (B), as well as the problematic component of the constructor's argument.

Recursive datatypes

A recursive datatype like

```
datatype t = A | B of int * t
```

introduces a new problem, since in order to decide whether t admits equality, we need to know for the B variant whether t admits equality. The Definition answers this question by requiring a type constructor to admit equality if it is consistent to do so. So, in our above example, if we assume that t admits equality, then the variant B of int * t admits equality. Then, since the A variant trivially admits equality, so does the type constructor t. Thus, it was consistent to assume that t admits equality, and so, t does admit equality.

On the other hand, in the following declaration

```
datatype t = A | B of real * t
```

if we assume that t admits equality, then the B variant does not admit equality. Hence, the type constructor t does not admit equality, and our assumption was inconsistent. Hence, t does not admit equality.

The same kind of reasoning applies to mutually recursive datatypes as well. For example, the following defines both t and u to admit equality.

```
datatype t = A | B of u
and u = C | D of t
```

But the following defines neither t nor u to admit equality.

```
datatype t = A | B of u * real
and u = C | D of t
```

As always, we can check whether a type admits equality using an eqtype specification.

```
structure Bad: sig eqtype t eqtype u end =
   struct
      datatype t = A | B of u * real
      and u = C | D of t
   end
```

MLton reports the following error.

```
Error: z.sml 1.16.
  Type t admits equality in signature but not in structure.
    not equality: B of [u] * [real]
Error: z.sml 1.16.
  Type u admits equality in signature but not in structure.
    not equality: D of [t]
```

Last edited on 2005-12-02 06:44:43 by StephenWeeks.

Alice

Alice is an extension of SML with concurrency, distribution, and constraint solving.

Last edited on 2004-12-28 19:46:32 by StephenWeeks.

AllocateRegisters

AllocateRegisters is an analysis pass for the RSSA IntermediateLanguage, invoked from ToMachine.

Description

Computes an allocation of RSSA variables as Machine register or stack operands.

Implementation

allocate-registers.sig allocate-registers.fun

Details and Notes

Last edited on 2005-11-30 19:54:55 by StephenWeeks.

AndreiFormiga

I'm a graduate student just back in academia. I study concurrent and parallel systems, with a great deal of interest in programming languages (theory, design, implementation). I happen to like functional languages.

I use the nickname tautologico on #sml and my email is andrei DOT formiga AT gmail DOT com.

Last edited on 2004-11-20 18:17:19 by AndreiFormiga.

ArrayLiteral

Standard ML does not have a syntax for array literals or vector literals. The only way to write down an array is like

```
Array.fromList [w, x, y, z]
```

No SML compiler produces efficient code for the above expression. The generated code allocates a list and then converts it to an array. To alleviate this, one could write down the same array using `Array.tabulate`, or even using `Array.array` and `Array.update`, but that is syntactically unwieldy.

Fortunately, using Fold, it is possible to define constants `A`, and `` ` `` so that one can write down an array like:

```
A `w `x `y `z $
```

This is as syntactically concice as the `fromList` expression. Furthermore, MLton, at least, will generate the efficient code as if one had written down a use of `Array.array` followed by four uses of `Array.update`.

Along with `A` and `` ` ``, one can define a constant `V` that makes it possible to define vector literals with the same syntax, e.g.,

```
V `w `x `y `z $
```

Note that the same element indicator, `` ` ``, serves for both array and vector literals. Of course, the `$` is the end-of-arguments marker always used with Fold. The only difference between an array literal and vector literal is the `A` or `V` at the beginning.

Here is the implementation of `A`, `V`, and `` ` ``. We place them in a structure and use signature abstraction to hide the type of the accumulator. See Fold for more on this technique.

```
structure Literal:>
   sig
      type 'a z
      val A: ('a z, 'a z, 'a array, 'd) Fold.t
      val V: ('a z, 'a z, 'a vector, 'd) Fold.t
      val ` : ('a, 'a z, 'a z, 'b, 'c, 'd) Fold.step1
   end =
   struct
      type 'a z = int * 'a option * ('a array -> unit)

      val A =
         fn z =>
         Fold.fold
         ((0, NONE, ignore),
          fn (n, opt, fill) =>
          case opt of
             NONE =>
                Array.tabulate (0, fn _ => raise Fail "array0")
           | SOME x =>
                let
                   val a = Array.array (n, x)
                   val () = fill a
                in
                   a
```

```
              end)
         z

      val V = fn z => Fold.post (A, Array.vector) z

      val ` =
         fn z =>
         Fold.step1
         (fn (x, (i, opt, fill)) =>
          (i + 1,
           SOME x,
           fn a => (Array.update (a, i, x); fill a)))
         z
   end
```

The idea of the code is for the fold to accumulate a count of the number of elements, a sample element, and a function that fills in all the elements. When the fold is complete, the finishing function allocates the array, applies the fill function, and returns the array. The only difference between `A` and `V` is at the very end; `A` just returns the array, while `V` converts it to a vector using post-composition, which is further described on the Fold page.

Last edited on 2006-03-21 22:05:32 by StephenWeeks.

BasisLibrary

The Standard ML Basis Library is a collection of modules dealing with basic types, input/output, OS interfaces, and simple datatypes. It is intended as a portable library usable across all implementations of SML. The official online version of the Basis Library specification is at http://www.standardml.org/Basis/. We keep a copy at http://mlton.org/basis/. There is a book that includes all of the online version and more. For a reverse chronological list of changes to the specification, see http://www.standardml.org/Basis/history.html.

MLton implements all of the required portions of the Basis Library. MLton also implements many of the optional structures. You can obtain a complete and current list of what's available using `mlton -show-basis` (see ShowBasis). By default, MLton makes the Basis Library available to user programs. You can also access the Basis Library from ML Basis files.

Below is a complete list of what MLton implements.

1. Top-level types and constructors
2. Top-level exception constructors
3. Top-level values
4. Overloaded identifiers
5. Top-level signatures
6. Top-level structures
7. Type equivalences
8. Real and Math functions
9. Top-level functors

Top-level types and constructors

```
eqtype 'a array
datatype bool = false | true
eqtype char
type exn
eqtype int
datatype 'a list = nil | :: of ('a * 'a list)
datatype 'a option = NONE | SOME of 'a
datatype order = EQUAL | GREATER | LESS
type real
datatype 'a ref = ref of 'a
eqtype string
type substring
eqtype unit
eqtype 'a vector
eqtype word
```

Top-level exception constructors

```
Bind
Chr
Div
```

```
Domain
Empty
Fail of string
Match
Option
Overflow
Size
Span
Subscript
```

Top-level values

MLton does not implement the optional top-level value `use: string -> unit`, which conflicts with whole-program compilation because it allows new code to be loaded dynamically. MLton implements all other top-level values:

!, :=, <>, =, @, ^, app, before, ceil, chr, concat, exnMessage, exnName, explode, floor, foldl, foldr, getOpt, hd, ignore, implode, isSome, length, map, not, null, o, ord, print, real, rev, round, size, str, substring, tl, trunc, valOf, vector.

Overloaded identifiers

*, +, -, /, <, <=, >, >=, ~, abs, div, mod.

Top-level signatures

```
ARRAY
ARRAY2
ARRAY_SLICE
BIN_IO
BIT_FLAGS
BOOL
BYTE
CHAR
COMMAND_LINE
DATE
GENERAL
GENERIC_SOCK
IEEE_REAL
IMPERATIVE_IO
INET_SOCK
INTEGER
INT_INF
IO
LIST
LIST_PAIR
MATH
MONO_ARRAY
MONO_ARRAY2
```

```
MONO_ARRAY_SLICE
MONO_VECTOR
MONO_VECTOR_SLICE
NET_HOST_DB
NET_PROT_DB
NET_SERV_DB
OPTION
OS
OS_FILE_SYS
OS_IO
OS_PATH
OS_PROCESS
PACK_REAL
PACK_WORD
POSIX
POSIX_ERROR
POSIX_FILE_SYS
POSIX_IO
POSIX_PROCESS
POSIX_PROC_ENV
POSIX_SIGNAL
POSIX_SYS_DB
POSIX_TTY
PRIM_IO
REAL
SOCKET
STREAM_IO
STRING
STRING_CVT
SUBSTRING
TEXT
TEXT_IO
TEXT_STREAM_IO
TIME
TIMER
UNIX
UNIX_SOCK
VECTOR
VECTOR_SLICE
WORD
```

Top-level structures

```
structure Array: ARRAY
structure Array2: ARRAY2
structure ArraySlice: ARRAY_SLICE
structure BinIO: BIN_IO
structure BinPrimIO: PRIM_IO
structure Bool: BOOL
structure BoolArray: MONO_ARRAY
```

```
structure BoolArray2: MONO_ARRAY2
structure BoolArraySlice: MONO_ARRAY_SLICE
structure BoolVector: MONO_VECTOR
structure BoolVectorSlice: MONO_VECTOR_SLICE
structure Byte: BYTE
structure Char: CHAR
```

`Char` characters correspond to ISO-8859-1. The `Char` functions do not depend on locale.

```
structure CharArray: MONO_ARRAY
structure CharArray2: MONO_ARRAY2
structure CharArraySlice: MONO_ARRAY_SLICE
structure CharVector: MONO_VECTOR
structure CharVectorSlice: MONO_VECTOR_SLICE
structure CommandLine: COMMAND_LINE
structure Date: DATE
```

`Date.fromString` and `Date.scan` accept a space in addition to a zero for the first character of the day of the month. The Basis Library specification only allows a zero.

```
structure FixedInt: INTEGER
structure General: GENERAL
structure GenericSock: GENERIC_SOCK
structure IEEEReal: IEEE_REAL
structure INetSock: INET_SOCK
structure IO: IO
structure Int: INTEGER
structure Int1: INTEGER
structure Int2: INTEGER
structure Int3: INTEGER
structure Int4: INTEGER
...
structure Int31: INTEGER
structure Int32: INTEGER
structure Int64: INTEGER
structure IntArray: MONO_ARRAY
structure IntArray2: MONO_ARRAY2
structure IntArraySlice: MONO_ARRAY_SLICE
structure IntVector: MONO_VECTOR
structure IntVectorSlice: MONO_VECTOR_SLICE
structure Int8: INTEGER
structure Int8Array: MONO_ARRAY
structure Int8Array2: MONO_ARRAY2
structure Int8ArraySlice: MONO_ARRAY_SLICE
structure Int8Vector: MONO_VECTOR
structure Int8VectorSlice: MONO_VECTOR_SLICE
structure Int16: INTEGER
structure Int16Array: MONO_ARRAY
structure Int16Array2: MONO_ARRAY2
structure Int16ArraySlice: MONO_ARRAY_SLICE
structure Int16Vector: MONO_VECTOR
```

```
structure Int16VectorSlice: MONO_VECTOR_SLICE
structure Int32: INTEGER
structure Int32Array: MONO_ARRAY
structure Int32Array2: MONO_ARRAY2
structure Int32ArraySlice: MONO_ARRAY_SLICE
structure Int32Vector: MONO_VECTOR
structure Int32VectorSlice: MONO_VECTOR_SLICE
structure Int64Array: MONO_ARRAY
structure Int64Array2: MONO_ARRAY2
structure Int64ArraySlice: MONO_ARRAY_SLICE
structure Int64Vector: MONO_VECTOR
structure Int64VectorSlice: MONO_VECTOR_SLICE
structure IntInf: INT_INF
structure LargeInt: INTEGER
structure LargeIntArray: MONO_ARRAY
structure LargeIntArray2: MONO_ARRAY2
structure LargeIntArraySlice: MONO_ARRAY_SLICE
structure LargeIntVector: MONO_VECTOR
structure LargeIntVectorSlice: MONO_VECTOR_SLICE
structure LargeReal: REAL
structure LargeRealArray: MONO_ARRAY
structure LargeRealArray2: MONO_ARRAY2
structure LargeRealArraySlice: MONO_ARRAY_SLICE
structure LargeRealVector: MONO_VECTOR
structure LargeRealVectorSlice: MONO_VECTOR_SLICE
structure LargeWord: WORD
structure LargeWordArray: MONO_ARRAY
structure LargeWordArray2: MONO_ARRAY2
structure LargeWordArraySlice: MONO_ARRAY_SLICE
structure LargeWordVector: MONO_VECTOR
structure LargeWordVectorSlice: MONO_VECTOR_SLICE
structure List: LIST
structure ListPair: LIST_PAIR
structure Math: MATH
structure NetHostDB: NET_HOST_DB
structure NetProtDB: NET_PROT_DB
structure NetServDB: NET_SERV_DB
structure OS: OS
structure Option: OPTION
structure PackReal32Big: PACK_REAL
structure PackReal32Little: PACK_REAL
structure PackReal64Big: PACK_REAL
structure PackReal64Little: PACK_REAL
structure PackRealBig: PACK_REAL
structure PackRealLittle: PACK_REAL
structure PackWord32Big: PACK_WORD
structure PackWord32Little: PACK_WORD
structure Position: INTEGER
structure Posix: POSIX
structure Real: REAL
structure RealArray: MONO_ARRAY
```

```
structure RealArray2: MONO_ARRAY2
structure RealArraySlice: MONO_ARRAY_SLICE
structure RealVector: MONO_VECTOR
structure RealVectorSlice: MONO_VECTOR_SLICE
structure Real32: REAL
structure Real32Array: MONO_ARRAY
structure Real32Array2: MONO_ARRAY2
structure Real32ArraySlice: MONO_ARRAY_SLICE
structure Real32Vector: MONO_VECTOR
structure Real32VectorSlice: MONO_VECTOR_SLICE
structure Real64: REAL
structure Real64Array: MONO_ARRAY
structure Real64Array2: MONO_ARRAY2
structure Real64ArraySlice: MONO_ARRAY_SLICE
structure Real64Vector: MONO_VECTOR
structure Real64VectorSlice: MONO_VECTOR_SLICE
structure Socket: SOCKET
```

The Basis Library specification requires functions like `Socket.sendVec` to raise an exception if they fail. However, on some platforms, sending to a socket that hasn't yet been connected causes a `SIGPIPE` signal, which invokes the default signal handler for `SIGPIPE` and causes the program to terminate. If you want the exception to be raised, you can ignore `SIGPIPE` by adding the following to your program.

```
let
   open MLton.Signal
in
   setHandler (Posix.Signal.pipe, Handler.ignore)
end
```

```
structure String: STRING
```

The `String` functions do not depend on locale.

```
structure StringCvt: STRING_CVT
structure Substring: SUBSTRING
structure SysWord: WORD
structure Text: TEXT
structure TextIO: TEXT_IO
structure TextPrimIO: PRIM_IO
structure Time: TIME
structure Timer: TIMER
structure Unix: UNIX
structure UnixSock: UNIX_SOCK
structure Vector: VECTOR
structure VectorSlice: VECTOR_SLICE
structure Word: WORD
structure Word1: WORD
structure Word2: WORD
structure Word3: WORD
structure Word4: WORD
...
```

```
structure Word31: WORD
structure Word32: WORD
structure Word64: WORD
structure WordArray: MONO_ARRAY
structure WordArray2: MONO_ARRAY2
structure WordArraySlice: MONO_ARRAY_SLICE
structure WordVectorSlice: MONO_VECTOR_SLICE
structure WordVector: MONO_VECTOR
structure Word8Array: MONO_ARRAY
structure Word8Array2: MONO_ARRAY2
structure Word8ArraySlice: MONO_ARRAY_SLICE
structure Word8Vector: MONO_VECTOR
structure Word8VectorSlice: MONO_VECTOR_SLICE
structure Word16Array: MONO_ARRAY
structure Word16Array2: MONO_ARRAY2
structure Word16ArraySlice: MONO_ARRAY_SLICE
structure Word16Vector: MONO_VECTOR
structure Word16VectorSlice: MONO_VECTOR_SLICE
structure Word32Array: MONO_ARRAY
structure Word32Array2: MONO_ARRAY2
structure Word32ArraySlice: MONO_ARRAY_SLICE
structure Word32Vector: MONO_VECTOR
structure Word32VectorSlice: MONO_VECTOR_SLICE
structure Word64Array: MONO_ARRAY
structure Word64Array2: MONO_ARRAY2
structure Word64ArraySlice: MONO_ARRAY_SLICE
structure Word64Vector: MONO_VECTOR
structure Word64VectorSlice: MONO_VECTOR_SLICE
```

Type equivalences

The following types are equivalent.

```
Int.int = Int32.int
Int64.int = FixedInt.int = Position.int
IntInf.int = LargeInt.int
Real.real = Real64.real = LargeReal.real
Word.word = Word32.word = SysWord.word
Word64.word = LargeWord.word
```

Real and Math functions

The `Real`, `Real32`, and `Real64` modules are implemented using the `C` math library, so the SML functions will reflect the behavior of the underlying library function. We have made some effort to unify the differences between the math libraries on different platforms, and in particular to handle exceptional cases according to the Basis Library specification. However, there will be differences due to different numerical algorithms and cases we may have missed. Please submit a bug report if you encounter an error in the handling of an exceptional case.

On x86, real arithmetic is implemented internally using 80 bits of precision. Using higher precision for intermediate results in computations can lead to different results than if all the computation is done at 32 or 64 bits. If you require strict IEEE compliance, you can compile with `-ieee-fp true`, which will cause intermediate results to be stored after each operation. This may cause a substantial performance penalty.

Top-level functors

```
ImperativeIO
PrimIO
StreamIO
```

MLton's `StreamIO` functor takes structures `ArraySlice` and `VectorSlice` in addition to the arguments specified in the Basis Library specification.

Last edited on 2005-11-30 23:04:45 by StephenWeeks.

Bug

To report a bug, please send mail to MLton@mlton.org. Please include the complete SML program that caused the problem and a log of a compile of the program with `-verbose 2`. For large messages (over 256K), please send an email containing the discussion text and a link to any large files. You may use our TemporaryUpload page for uploading large files.

There are some UnresolvedBugs that we don't plan to fix.

We also maintain a list of bugs found with each release.

- Bugs20051202
- Bugs20041109

Last edited on 2006-08-10 12:26:17 by VesaKarvonen.

Bugs20041109

Here are the known bugs in MLton 20041109, listed in reverse chronological order of date reported.

* `MLton.Finalizable.touch` doesn't necessarily keep values alive long enough. Our SVN has a patch to the compiler. You must rebuild the compiler in order for the patch to take effect.

  Thanks to Florian Weimer for reporting this bug.

* A bug in an optimization pass may incorrectly transform a program to flatten ref cells into their containing data structure, yielding a type-error in the transformed program. Our CVS has a patch to the compiler. You must rebuild the compiler in order for the patch to take effect.

  Thanks to VesaKarvonen for reporting this bug.

* A bug in the front end mistakenly allows unary constructors to be used without an argument in patterns. For example, the following program is accepted, and triggers a large internal error.

  ```
  fun f x = case x of SOME => true | _ => false
  ```

  We have fixed the problem in our CVS.

  Thanks to William Lovas for reporting this bug.

* A bug in Posix.IO.{getlk,setlk,setlkw} causes a link-time error: `undefined reference to Posix_IO_FLock_typ` Our CVS has a patch to the Basis Library implementation.

  Thanks to Adam Chlipala for reporting this bug.

* A bug can cause programs compiled with `-profile alloc` to segfault. Our CVS has a patch to the compiler. You must rebuild the compiler in order for the patch to take effect.

  Thanks to John Reppy for reporting this bug.

* A bug in an optimization pass may incorrectly flatten ref cells into their containing data structure, breaking the sharing between the cells. Our CVS has a patch to the compiler. You must rebuild the compiler in order for the patch to take effect.

  Thanks to Paul Govereau for reporting this bug.

* Some arrays or vectors, such as `(char * char) vector`, are incorrectly implemented, and will conflate the first and second components of each element. Our CVS has a patch to the compiler. You must rebuild the compiler in order for the patch to take effect.

  Thanks to Scott Cruzen for reporting this bug.

* `Socket.Ctl.getLINGER` and `Socket.Ctl.setLINGER` mistakenly raise `Subscript`. Our CVS has a patch to the Basis Library implementation.

Thanks to Ray Racine for reporting the bug.

- CML `Mailbox.send` makes a call in the wrong atomic context. Our CVS has a patch to the CML implementation.
- `OS.Path.joinDirFile` and `OS.Path.toString` did not raise `InvalidArc` when they were supposed to. They now do. Our CVS has a patch to the Basis Library implementation.

  Thanks to Andreas Rossberg for reporting the bug.
- The front end incorrectly disallows sequences of expressions (separated by semicolons) after a topdec has already been processed. For example, the following is incorrectly rejected.

  ```
  val x = 0;
  ignore x;
  ignore x;
  ```

  We have fixed the problem in our CVS.

  Thanks to Andreas Rossberg for reporting the bug.
- The front end incorrectly disallows expansive `val` declarations that bind a type variable that doesn't occur in the type of the value being bound. For example, the following is incorrectly rejected.

  ```
  val 'a x = let exception E of 'a in () end
  ```

  We have fixed the problem in our CVS.

  Thanks to Andreas Rossberg for reporting this bug.
- The x86 codegen fails to account for the possibility that a 64-bit move could interfere with itself (as simulated by 32-bit moves). We have fixed the problem in our CVS.

  Thanks to Scott Cruzen for reporting this bug.
- `NetHostDB.scan` and `NetHostDB.fromString` incorrectly raise an exception on internet addresses whose last component is a zero, e.g `0.0.0.0`. Our CVS has a patch to the Basis Library implementation.

  Thanks to Scott Cruzen for reporting this bug.
- `StreamIO.inputLine` has an off-by-one error causing it to drop the first character after a newline in some situations. Our CVS has a patch. to the Basis Library implementation.

  Thanks to Scott Cruzen for reporting this bug.
- `BinIO.getInstream` and `TextIO.getInstream` are implemented incorrectly. This also impacts the behavior of `BinIO.scanStream` and `TextIO.scanStream`. If you (directly or indirectly) realize a `TextIO.StreamIO.instream` and do not (directly or indirectly) call `TextIO.setInstream` with a derived stream, you may lose input data. We have fixed the problem in our CVS.

Thanks to WesleyTerpstra for reporting this bug.

- `Posix.ProcEnv.setpgid` doesn't work. If you compile a program that uses it, you will get a link time error

  ```
  undefined reference to `Posix_ProcEnv_setpgid'
  ```

  The bug is due to `Posix_ProcEnv_setpgid` being omitted from the MLton runtime. We fixed the problem in our CVS by adding the following definition to `runtime/Posix/ProcEnv/ProcEnv.c`

  ```
   Int Posix_ProcEnv_setpgid (Pid p, Gid g) {
          return setpgid (p, g);
  }
  ```

  Thanks to Tom Murphy for reporting this bug.

Last edited on 2005-12-01 05:16:27 by StephenWeeks.

Bugs20051202

Here are the known bugs in MLton 20051202, listed in reverse chronological order of date reported.

- The MONO_VECTOR signature of the basis library implementation incorrectly omits the specification of `find`.
- The optimizer reports an internal error ("TypeError") when an imported C function is called but not used. Thanks to `jq` for the bug report.

Last edited on 2006-10-13 11:19:07 by VesaKarvonen.

CKitLibrary

The ckit Library is a C front end written in SML that translates C source code (after preprocessing) into abstract syntax represented as a set of SML datatypes. The ckit Library is distributed with SML/NJ. Due to differences between SML/NJ and MLton, this library will not work out-of the box with MLton.

As of 20050818, MLton includes a port of the ckit Library synchronized with SML/NJ version 110.57.

Usage

- You can import the ckit Library into an MLB file with `$(SML_LIB)/ckit-lib/ckit-lib.mlb`
- If you are porting a project from SML/NJ's CompilationManager to MLton's ML Basis system using `cm2mlb`, note that the following map is included by default:

  ```
  $ckit-lib.cm/ckit-lib.cm   $(SML_LIB)/ckit-lib/ckit-lib.mlb
  ```

  This will automatically convert a `$/ckit-lib.cm` import in an input `.cm` file into a `$(SML_LIB)/ckit-lib/ckit-lib.mlb` import in the output `.mlb` file.

Details

The following changes were made to the ckit Library, in addition to deriving the `.mlb` file from the `.cm` files:

- `parser/parse-tree-sig.sml` (modified): Rewrote use of (sequential) `withtype` in signature.
- `parser/parse-tree.sml` (modified): Rewrote use of (sequential) `withtype`.
- `ast/ast-sig.sml` (modified): Rewrote use of `withtype` in signature.
- `ast/pp/pp-lib.sml` (modified): Rewrote use of *or-patterns*.
- `ast/pp/pp-ast-ext-sig.sml` (modified): Rewrote use of `signature` in `local`.
- `ast/pp/pp-ast-adornment-sig.sml` (modified): Rewrote use of `signature` in `local`.
- `ast/type-util-sig.sml` (modified): Rewrote use of `signature` in `local`.
- `ast/type-util.sml` (modified): Rewrote use of *or-patterns*.
- `ast/sizeof.sml` (modified): Rewrote use of *or-patterns*.
- `ast/initializer-normalizer.sml` (modified): Rewrote use of *or-patterns*.
- `ast/build-ast.sml` (modified): Rewrote use of *or-patterns*.

Patch

- ckit.patch

Last edited on 2006-03-04 17:01:32 by MatthewFluet.

CMinusMinus

C-- is a portable assembly language intended to make it easy for compilers for different high-level languages to share the same backend. An experimental version of MLton has been made to generate C--.

http://mlton.org/pipermail/mlton/2005-March/026850.html

Also see

- LLVM

Last edited on 2006-09-04 20:25:21 by StephenWeeks.

CallGraph

For easier visualization of profiling data, `mlprof` can create a call graph of the program in dot format, from which you can use the graphviz software package to create a postscript graph. For example,

```
mlprof -call-graph foo.dot foo mlmon.out
```

will create `foo.dot` with a complete call graph. For each source function, there will be one node in the graph that contains the function name (and source position with `-show-line true`), as well as the percentage of ticks. If you want to create a call graph for your program without any profiling data, you can simply call `mlprof` without any `mlmon.out` files, as in

```
mlprof -call-graph foo.dot foo
```

Because SML has higher-order functions, the call graph is is dependent on MLton's analysis of which functions call each other. This analysis depends on many implementation details and might display spurious edges that a human could conclude are impossible. However, in practice, the call graphs tend to be very accurate.

Because call graphs can get big, `mlprof` provides the `-keep` option to specify the nodes that you would like to see. This option also controls which functions appear in the table that `mlprof` prints. The argument to `-keep` is an expression describing a set of source functions (i.e. graph nodes). The expression *e* should be of the following form.

- `all`
- `"`*s*`"`
- `(and` *e* ...`)`
- `(from` *e*`)`
- `(not` *e*`)`
- `(or` *e*`)`
- `(pred` *e*`)`
- `(succ` *e*`)`
- `(thresh` *x*`)`
- `(thresh-gc` *x*`)`
- `(thresh-stack` *x*`)`
- `(to` *e*`)`

In the grammar, `all` denotes the set of all nodes. `"`*s*`"` is a regular expression denoting the set of functions whose name (followed by a space and the source position) has a prefix matching the regexp. The `and`, `not`, and `or` expressions denote intersection, complement, and union, respectively. The `pred` and `succ` expressions add the set of immediate predecessors or successors to their argument, respectively. The `from` and `to` expressions denote the set of nodes that have paths from or to the set of nodes denoted by their arguments, respectively. Finally, `thresh`, `thresh-gc`, and `thresh-stack` denote the set of nodes whose percentage of ticks, gc ticks, or stack ticks, respectively, is greater than or equal to the real number *x*.

For example, if you want to see the entire call graph for a program, you can use `-keep all` (this is the default). If you want to see all nodes reachable from function `foo` in your program, you would use `-keep '(from "foo")'`. Or, if you want to see all the functions defined in subdirectory `bar` of your project that used at least 1% of the ticks, you would use

```
-keep '(and ".*/bar/" (thresh 1.0))'
```

To see all functions with ticks above a threshold, you can also use `-thresh x`, which is an abbreviation for `-keep '(thresh x)'`. You can not use multiple `-keep` arguments or both `-keep` and `-thresh`. When you use `-keep` to display a subset of the functions, `mlprof` will add dashed edges to the call graph to indicate a path in the original call graph from one function to another.

When compiling with `-profile-stack true`, you can use `mlprof -gray true` to make the nodes darker or lighter depending on whether their stack percentage is higher or lower.

MLton's optimizer may duplicate source functions for any of a number of reasons (functor duplication, monomorphisation, polyvariance, inlining). By default, all duplicates of a function are treated as one. If you would like to treat the duplicates separately, you can use `mlprof -split` *regexp*, which will cause all duplicates of functions whose name has a prefix matching the regular expression to be treated separately. This can be especially useful for higher-order utility functions like `General.o`.

Caveats

Technically speaking, `mlprof` produces a call-stack graph rather than a call graph, because it describes the set of possible call stacks. The difference is in how tail calls are displayed. For example if `f` nontail calls `g` and `g` tail calls `h`, then the call-stack graph has edges from `f` to `g` and `f` to `h`, while the call graph has edges from `f` to `g` and `g` to `h`. That is, a tail call from `g` to `h` removes `g` from the call stack and replaces it with `h`.

Last edited on 2005-11-30 23:11:25 by StephenWeeks.

CallingFromCToSML

MLton's ForeignFunctionInterface allows programs to *export* SML functions to be called from C. Suppose you would like export from SML a function of type `real * char -> int` as the C function `foo`. MLton extends the syntax of SML to allow expressions like the following:

```
_export "foo": (real * char -> int) -> unit;
```

The above expression exports a C function named `foo`, with prototype

```
Int32 foo (Real64 x0, Char x1);
```

The `_export` expression denotes a function of type `(real * char -> int) -> unit` that when called with a function `f`, arranges for the exported `foo` function to call `f` when `foo` is called. So, for example, the following exports and defines `foo`.

```
val e = _export "foo": (real * char -> int) -> unit;
val _ = e (fn (x, c) => 13 + Real.floor x + Char.ord c)
```

The general form of an `_export` expression is

```
_export "C function name" attr... : cFuncTy -> unit;
```

The type and the semicolon are not optional. As with `_import`, a sequence of attributes may follow the function name.

MLton's `-export-header` option generates a C header file with prototypes for all of the functions exported from SML. Include this header file in your C files to type check calls to functions exported from SML. This header file includes `typedefs` for the types that can be passed between SML and C.

Example

Suppose that `export.sml` is

```
val e = _export "f": (int * real * char -> char) -> unit;
val _ = e (fn (i, r, _) =>
           (print (concat ["i = ", Int.toString i,
                           "  r = ", Real.toString r, "\n"])
            ; #"g"))
val g = _import "g": unit -> unit;
val _ = g ()
val _ = g ()

val e = _export "f2": (Word8.word -> word array) -> unit;
val _ = e (fn w =>
           Array.tabulate (10, fn _ => Word.fromLargeWord (Word8.toLargeWord w)))
val g2 = _import "g2": unit -> word array;
val a = g2 ()
val _ = print (concat ["0wx", Word.toString (Array.sub (a, 0)), "\n"])

val e = _export "f3": (unit -> unit) -> unit;
val _ = e (fn () => print "hello\n");
val g3 = _import "g3": unit -> unit;
val _ = g3 ()
```

```
(* This example demonstrates mutual recursion between C and SML. *)
val e = _export "f4": (int -> unit) -> unit;
val g4 = _import "g4": int -> unit;
val _ = e (fn i => if i = 0 then () else g4 (i - 1))
val _ = g4 13

val (_, zzzSet) = _symbol "zzz" alloc: (unit -> int) * (int -> unit);
val () = zzzSet 42
val g5 = _import "g5": unit -> unit;
val _ = g5 ()

val _ = print "success\n"
```

Create the header file with `-export-header`.

```
% mlton -default-ann 'allowFFI true'    \
        -export-header export.h         \
        -stop tc                        \
        export.sml
```

`export.h` now contains the following C prototypes.

```
Int8 f (Int32 x0, Real64 x1, Int8 x2);
Pointer f2 (Word8 x0);
void f3 ();
void f4 (Int32 x0);
extern Int32 zzz;
```

Use `export.h` in a C program, `ffi-export.c`, as follows.

```
#include <stdio.h>
#include "export.h"

void g () {
        Char c;

        fprintf (stderr, "g starting\n");
        c = f (13, 17.15, 'a');
        fprintf (stderr, "g done  char = %c\n", c);
}

Pointer g2 () {
        Pointer res;
        fprintf (stderr, "g2 starting\n");
        res = f2 (0xFF);
        fprintf (stderr, "g2 done\n");
        return res;
}

void g3 () {
        fprintf (stderr, "g3 starting\n");
        f3 ();
        fprintf (stderr, "g3 done\n");
}

void g4 (Int i) {
        fprintf (stderr, "g4 (%d)\n", i);
```

```
        f4 (i);
}

void g5 () {
        fprintf (stderr, "g5 ()\n");
        fprintf (stderr, "zzz = %i\n", zzz);
        fprintf (stderr, "g5 done\n");
}
```

Compile `ffi-export.c` and `export.sml`.

```
% gcc -c ffi-export.c
% mlton -default-ann 'allowFFI true' \
         export.sml ffi-export.o
```

Finally, run `export`.

```
% ./export
g starting
...
g4 (0)
success
```

Download

- export.sml
- ffi-export.c

Last edited on 2005-11-30 23:11:45 by StephenWeeks.

CallingFromSMLToC

MLton's ForeignFunctionInterface allows an SML program to *import* C functions. Suppose you would like to import from C a function with the following prototype:

```
int foo (double d, char c);
```

MLton extends the syntax of SML to allow expressions like the following:

```
_import "foo": real * char -> int;
```

This expression denotes a function of type `real * char -> int` whose behavior is implemented by calling the C function whose name is `foo`. Thinking in terms of C, imagine that there are C variables `d` of type `double`, `c` of type `unsigned char`, and `i` of type `int`. Then, the C statement `i = foo (d, c)` is executed and `i` is returned.

The general form of an `_import` expression is:

```
_import "C function name" attr... : cFuncTy;
```

The type and the semicolon are not optional.

The function name is followed by a (possibly empty) sequence of attributes, analogous to C `__attribute__` specifiers. For now, the only attributes supported are `cdecl` and `stdcall`. These specify the calling convention of the C function on Cygwin/Windows, and are ignored on all other platforms. The default is `cdecl`. You must use `stdcall` in order to correctly call Windows API functions.

Example

`import.sml` imports the C function `ffi` and the C variable `FFI_INT` as follows.

```
(* main.sml *)

(* Declare ffi to be implemented by calling the C function ffi. *)
val ffi = _import "ffi": real array * int ref * int -> char;
open Array

val size = 10
val a = tabulate (size, fn i => real i)
val r = ref 0
val n = 17

(* Call the C function *)
val c = ffi (a, r, n)

val (nGet, nSet) = _symbol "FFI_INT": (unit -> int) * (int -> unit);

val _ = print (concat [Int.toString (nGet ()), "\n"])

val _ =
   print (if c = #"c" andalso !r = 45
             then "success\n"
          else "fail\n")
```

`ffi-import.c` is

```
#include "platform.h"

Int FFI_INT = 13;
Word FFI_WORD = 0xFF;
Bool FFI_BOOL = TRUE;
Real FFI_REAL = 3.14159;

Char ffi (Pointer a1, Pointer a2, Int n) {
        double *ds = (double*)a1;
        int *p = (int*)a2;
        int i;
        double sum;

        sum = 0.0;
        for (i = 0; i < GC_arrayNumElements (a1); ++i) {
                sum += ds[i];
                ds[i] += n;
        }
        *p = (int)sum;
        return 'c';
}
```

Compile and run the program.

```
% mlton -default-ann 'allowFFI true' import.sml ffi-import.c
% ./import
13
success
```

Download

- import.sml
- ffi-import.c

Next Steps

- CallingFromSMLToCFunctionPointer

Last edited on 2005-12-02 04:17:30 by StephenWeeks.

CallingFromSMLToCFunctionPointer

Just as MLton can directly call C functions, it is possible to make indirect function calls; that is, function calls through a function pointer. MLton extends the syntax of SML to allow expressions like the following:

```
_import * : MLton.Pointer.t -> real * char -> int;
```

This expression denotes a function of type

```
MLton.Pointer.t -> real * char -> int
```

whose behavior is implemented by calling the C function at the address denoted by the `MLton.Pointer.t` argument, and supplying the C function two arguments, a `double` and an `int`. The C function pointer may be obtained, for example, by the dynamic linking loader (`dlopen`, `dlsym`, ...).

The general form of an indirect `_import` expression is:

```
_import * attr... : cPtrTy -> cFuncTy;
```

The type and the semicolon are not optional.

Example

This example uses `dlopen` and friends (imported using normal `_import`) to dynamically load the math library (`libm`) and call the `cos` function. Suppose `iimport.sml` contains the following.

```
signature DYN_LINK =
   sig
      type hndl
      type mode
      type fptr

      val dlopen : string * mode -> hndl
      val dlsym : hndl * string -> fptr
      val dlclose : hndl -> unit

      val RTLD_LAZY : mode
      val RTLD_NOW : mode
   end

structure DynLink :> DYN_LINK =
   struct
      type hndl = MLton.Pointer.t
      type mode = Word32.word
      type fptr = MLton.Pointer.t

      val dlopen =
         _import "dlopen" : string * mode -> hndl;
      val dlerror =
         _import "dlerror": unit -> MLton.Pointer.t;
      val dlsym =
         _import "dlsym" : hndl * string -> fptr;
      val dlclose =
         _import "dlclose" : hndl -> Int32.int;
```

```
val RTLD_LAZY = 0wx00001 (* Lazy function call binding.  *)
val RTLD_NOW  = 0wx00002 (* Immediate function call binding.  *)

val dlerror = fn () =>
   let
      val addr = dlerror ()
   in
      if addr = MLton.Pointer.null
         then NONE
         else let
                 fun loop (index, cs) =
                    let
                       val w = MLton.Pointer.getWord8 (addr, index)
                       val c = Byte.byteToChar w
                    in
                       if c = #"\000"
                          then SOME (implode (rev cs))
                          else loop (index + 1, c::cs)
                    end
              in
                 loop (0, [])
              end
   end

val dlopen = fn (filename, mode) =>
   let
      val filename = filename ^ "\000"
      val hndl = dlopen (filename, mode)
   in
      if hndl = MLton.Pointer.null
         then raise Fail (case dlerror () of
                             NONE => "???"
                           | SOME s => s)
         else hndl
   end

val dlsym = fn (hndl, symbol) =>
   let
      val symbol = symbol ^ "\000"
      val fptr = dlsym (hndl, symbol)
   in
      case dlerror () of
         NONE => fptr
       | SOME s => raise Fail s
   end

val dlclose = fn hndl =>
   if MLton.Platform.OS.host = MLton.Platform.OS.Darwin
      then ()  (* Darwin reports the following error message if you
                * try to close a dynamic library.
                *   "dynamic libraries cannot be closed"
                * So, we disable dlclose on Darwin.
                *)
   else
      let
         val res = dlclose hndl
      in
         if res = 0
            then ()
         else raise Fail (case dlerror () of
                             NONE => "???"
```

```
                                    | SOME s => s)
            end
   end

val dll =
   let
      open MLton.Platform.OS
   in
      case host of
         Cygwin => "cygwin1.dll"
       | Darwin => "libm.dylib"
       | _ => "libm.so"
   end

val hndl = DynLink.dlopen (dll, DynLink.RTLD_LAZY)

local
   val double_to_double =
      _import * : DynLink.fptr -> real -> real;
   val cos_fptr = DynLink.dlsym (hndl, "cos")
in
   val cos = double_to_double cos_fptr
end

val _ = print (concat ["    Math.cos(2.0) = ", Real.toString (Math.cos 2.0), "\n",
                       "libm.so::cos(2.0) = ", Real.toString (cos 2.0), "\n"])

val _ = DynLink.dlclose hndl
```

Compile and run `iimport.sml`.

```
% mlton -default-ann 'allowFFI true'    \
        -target-link-opt linux -ldl     \
        -target-link-opt solaris -ldl   \
         iimport.sml
% iimport
    Math.cos(2.0) = ~0.416146836547
libm.so::cos(2.0) = ~0.416146836547
```

This example also shows the `-target-link-opt` option, which uses the switch when linking only when on the specified platform. Compile with `-verbose 1` to see in more detail what's being passed to `gcc`.

Download

- iimport.sml

Last edited on 2005-11-30 23:18:27 by StephenWeeks.

ChrisClearwater

Last edited on 2005-11-30 23:18:55 by StephenWeeks.

Chunkify

Chunkify is an analysis pass for the RSSA IntermediateLanguage, invoked from ToMachine.

Description

It partitions all the labels (function and block) in an RSSA program into disjoint sets, referred to as chunks.

Implementation

chunkify.sig chunkify.fun

Details and Notes

Breaking large RSSA functions into chunks is necessary for reasonable `gcc` compile times with the CCodegen.

Last edited on 2005-11-30 23:19:46 by StephenWeeks.

Closure

A closure is a data structure that is the run-time representation of a function.

Typical Implementation

In a typical implementation, a closure consists of a *code pointer* (indicating what the function does) and an *environment* containing the values of the free variables of the function. For example, in the expression

```
let
   val x = 5
in
   fn y => x + y
end
```

the closure for `fn y => x + y` contains a pointer to a piece of code that knows to take its argument and add the value of `x` to it, plus the environment recording the value of `x` as `5`.

To call a function, the code pointer is extracted and jumped to, passing in some agreed upon location the environment and the argument.

MLton's Implementation

MLton does not implement closures traditionally. Instead, based on whole-program higher-order control-flow analysis, MLton represents a function as an element of a sum type, where the variant indicates which function it is and carries the free variables as arguments. See ClosureConvert and CejtinEtAl00 for details.

Last edited on 2005-11-30 23:25:36 by StephenWeeks.

ClosureConvert

ClosureConvert is a translation pass from the SXML IntermediateLanguage to the SSA IntermediateLanguage.

Description

It converts an SXML program into an SSA program.

Defunctionalization is the technique used to eliminate Closures (see CejtinEtAl00).

Uses Globalize and LambdaFree analyses.

Implementation

closure-convert.sig closure-convert.fun

Details and Notes

Last edited on 2005-12-02 04:17:57 by StephenWeeks.

CommonArg

CommonArg is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

It optimizes instances of `Goto` transfers that pass the same arguments to the same label; e.g.

```
L_1 ()
  ...
  z1 = ?
  ...
  L_3 (x, y, z1)
L_2 ()
  ...
  z2 = ?
  ...
  L_3 (x, y, z2)
L_3 (a, b, c)
  ...
```

This code can be simplified to:

```
L_1 ()
  ...
  z1 = ?
  ...
  L_3 (z1)
L_2 ()
  ...
  z2 = ?
  ...
  L_3 (z2)
L_3 (c)
  a = x
  b = y
```

which saves a number of resources: time of setting up the arguments for the jump to `L_3`, space (either stack or pseudo-registers) for the arguments of `L_3`, etc. It may also expose some other optimizations, if more information is known about `x` or `y`.

Implementation

common-arg.sig common-arg.fun

Details and Notes

Three analyses were originally proposed to drive the optimization transformation. Only the *Dominator Analysis* is currently implemented. (Implementations of the other analyses are available in the Subversion repository.)

Syntactic Analysis

The simplest analysis I could think of maintains

```
varInfo: Var.t -> Var.t option list ref
```

initialized to `[]`.

For each variable `v` bound in a `Statement.t` or in the `Function.t` args, then `List.push(varInfo v, NONE)`. For each `L (x1, ..., xn)` transfer where `(a1, ..., an)` are the formals of `L`, then `List.push(varInfo ai, SOME xi)`. For each block argument a used in an unknown context (e.g., arguments of blocks used as continuations, handlers, arith success, runtime return, or case switch labels), then `List.push(varInfo a, NONE)`.

Now, any block argument `a` such that `varInfo a = xs`, where all of the elements of `xs` are equal to `SOME x`, can be optimized by setting `a = x` at the beginning of the block and dropping the argument from `{Goto` transfers.

That takes care of the example above. We can clearly do slightly better, by changing the transformation criteria to the following: any block argument a such that `varInfo a = xs`, where all of the elements of `xs` are equal to `SOME x` *or* are equal to `SOME a`, can be optimized by setting `a = x` at the beginning of the block and dropping the argument from `Goto` transfers. This optimizes a case like:

```
L_1 ()
  ... z1 = ? ...
  L_3 (x, y, z1)
L_2 ()
  ... z2 = ? ...
  L_3(x, y, z2)
L_3 (a, b, c)
  ... w = ? ...
  case w of
    true => L_4 | false => L_5
L_4 ()
   ...
   L_3 (a, b, w)
L_5 ()
   ...
```

where a common argument is passed to a loop (and is invariant through the loop). Of course, the LoopInvariant optimization pass would normally introduce a local loop and essentially reduce this to the first example, but I have seen this in practice, which suggests that some optimizations after LoopInvariant do enough simplifications to introduce (new) loop invariant arguments.

Fixpoint Analysis

However, the above analysis and transformation doesn't cover the cases where eliminating one common argument exposes the opportunity to eliminate other common arguments. For example:

```
L_1 ()
  ...
  L_3 (x)
L_2 ()
  ...
  L_3 (x)
```

```
L_3 (a)
  ...
  L_5 (a)
L_4 ()
  ...
  L_5 (x)
L_5 (b)
  ...
```

One pass of analysis and transformation would eliminate the argument to `L_3` and rewrite the `L_5(a)` transfer to `L_5 (x)`, thereby exposing the opportunity to eliminate the common argument to `L_5`.

The interdependency the arguments to `L_3` and `L_5` suggest performing some sort of fixed-point analysis. This analysis is relatively simple; maintain

```
varInfo: Var.t -> VarLattice.t
```

where

```
VarLattice.t ~=~ Bot | Point of Var.t | Top
```

(but as implemented by the FlatLattice functor with a `lessThan` list and `value ref` under the hood), initialized to `Bot`.

For each variable `v` bound in a `Statement.t` or in the `Function.t` args, then `VarLattice.<= (Point v, varInfo v)` For each `L (x1, ..., xn)` transfer where `(a1, ..., an)` are the formals of `L`}, then `VarLattice.<= (varInfo xi, varInfo ai)`. For each block argument a used in an unknown context, then `VarLattice.<= (Point a, varInfo a)`.

Now, any block argument a such that `varInfo a = Point x` can be optimized by setting `a = x` at the beginning of the block and dropping the argument from `Goto` transfers.

Now, with the last example, we introduce the ordering constraints:

```
varInfo x <= varInfo a
varInfo a <= varInfo b
varInfo x <= varInfo b
```

Assuming that `varInfo x = Point x`, then we get `varInfo a = Point x` and `varInfo b = Point x`, and we optimize the example as desired.

But, that is a rather weak assumption. It's quite possible for `varInfo x = Top`. For example, consider:

```
G_1 ()
  ... n = 1 ...
  L_0 (n)
G_2 ()
  ... m = 2 ...
  L_0 (m)
L_0 (x)
  ...
L_1 ()
  ...
  L_3 (x)
L_2 ()
```

```
  ...
  L_3 (x)
L_3 (a)
  ...
  L_5(a)
L_4 ()
  ...
  L_5(x)
L_5 (b)
   ...
```

Now `varInfo x = varInfo a = varInfo b = Top`. What went wrong here? When `varInfo x` went to `Top`, it got propagated all the way through to `a` and `b`, and prevented the elimination of any common arguments. What we'd like to do instead is when `varInfo x` goes to `Top`, propagate on `Point x` -- we have no hope of eliminating `x`, but if we hold `x` constant, then we have a chance of eliminating arguments for which `x` is passed as an actual.

Dominator Analysis

Does anyone see where this is going yet? Pausing for a little thought, MatthewFluet realized that he had once before tried proposing this kind of "fix" to a fixed-point analysis -- when we were first investigating the Contify optimization in light of John Reppy's CWS paper. Of course, that "fix" failed because it defined a non-monotonic function and one couldn't take the fixed point. But, StephenWeeks suggested a dominator based approach, and we were able to show that, indeed, the dominator analysis subsumed both the previous call based analysis and the cont based analysis. And, a moment's reflection reveals further parallels: when `varInfo: Var.t -> Var.t option list ref`, we have something analogous to the call analysis, and when `varInfo: Var.t -> VarLattice.t`, we have something analogous to the cont analysis. Maybe there is something analogous to the dominator approach (and therefore superior to the previous analyses).

And this turns out to be the case. Construct the graph `G` as follows:

```
nodes(G) = {Root} U Var.t
edges(G) = {Root -> v | v bound in a Statement.t or
                        in the Function.t args} U
           {xi -> ai | L(x1, ..., xn) transfer where (a1, ..., an)
                                      are the formals of L} U
           {Root -> a | a is a block argument used in an unknown context}
```

Let `idom(x)` be the immediate dominator of `x` in `G` with root `Root`. Now, any block argument a such that `idom(a) = x <> Root` can be optimized by setting `a = x` at the beginning of the block and dropping the argument from `Goto` transfers.

Furthermore, experimental evidence suggests (and we are confident that a formal presentation could prove) that the dominator analysis subsumes the "syntactic" and "fixpoint" based analyses in this context as well and that the dominator analysis gets "everything" in one go.

Final Thoughts

I must admit, I was rather suprised at this progression and final result. At the outset, I never would have thought of a connection between Contify and CommonArg optimizations. They would seem to be two completely different optimizations. Although, this may not really be the case. As one of the reviewers of the ICFP paper said:

> I understand that such a form of CPS might be convenient in some cases, but when we're talking about analyzing code to detect that some continuation is constant, I think it makes a lot more sense to make all the continuation arguments completely explicit.
>
> I believe that making all the continuation arguments explicit will show that the optimization can be generalized to eliminating constant arguments, whether continuations or not.

What I think the common argument optimization shows is that the dominator analysis does slightly better than the reviewer puts it: we find more than just constant continuations, we find common continuations. And I think this is further justified by the fact that I have observed common argument eliminate some `env_X` arguments which would appear to correspond to determining that while the closure being executed isn't constant it is at least the same as the closure being passed elsewhere.

At first, I was curious whether or not we had missed a bigger picture with the dominator analysis. When we wrote the contification paper, I assumed that the dominator analysis was a specialized solution to a specialized problem; we never suggested that it was a technique suited to a larger class of analyses. After initially finding a connection between Contify and CommonArg (and thinking that the only connection was the technique), I wondered if the dominator technique really was applicable to a larger class of analyses. That is still a question, but after writing up the above, I'm suspecting that the "real story" is that the dominator analysis is a solution to the common argument optimization, and that the Contify optimization is specializing CommonArg to the case of continuation arguments (with a different transformation at the end). (Note, a whole-program, inter-procedural common argument analysis doesn't really make sense (in our SSA IntermediateLanguage), because the only way of passing values between functions is as arguments. (Unless of course in the case that the common argument is also a constant argument, in which case ConstantPropagation could lift it to a global.) The inter-procedural Contify optimization works out because there we move the function to the argument.)

Anyways, it's still unclear to me whether or not the dominator based approach solves other kinds of problems.

Phase Ordering

On the downside, the optimization doesn't have a huge impact on runtime, although it does predictably saved some code size. I stuck it in the optimization sequence after Flatten and (the third round of) LocalFlatten, since it seems to me that we could have cases where some components of a tuple used as an argument are common, but the whole tuple isn't. I think it makes sense to add it after IntroduceLoops and LoopInvariant (even though CommonArg get some things that LoopInvariant gets, it doesn't get all of them). I also think that it makes sense to add it before CommonSubexp, since identifying variables could expose more common subexpressions. I would think a similar thought applies to RedundantTests.

Last edited on 2005-11-30 23:32:23 by StephenWeeks.

CommonBlock

CommonBlock is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

It eliminates equivalent blocks in a SSA function. The equivalence criteria requires blocks to have no arguments or statements and transfer via `Raise`, `Return`, or `Goto` of a single global variable.

Implementation

common-block.sig common-block.fun

Details and Notes

- Rewrites

  ```
  L_X ()
    raise (global_Y)
  ```

  to

  ```
  L_X ()
    L_Y' ()
  ```

  and adds

  ```
  L_Y' ()
    raise (global_Y)
  ```

  to the SSA function.
- Rewrites

  ```
  L_X ()
    return (global_Y)
  ```

  to

  ```
  L_X ()
    L_Y' ()
  ```

  and adds

  ```
  L_Y' ()
    return (global_Y)
  ```

  to the SSA function.
- Rewrites

  ```
  L_X ()
    L_Z (global_Y)
  ```

to

```
L_X ()
  L_Y' ()
```

and adds

```
L_Y' ()
  L_Z (global_Y)
```

to the SSA function.

The Shrink pass rewrites all uses of `L_X` to `L_Y'` and drops `L_X`.

For example, all uncaught `Overflow` exceptions in a SSA function share the same raising block.

Last edited on 2005-11-30 23:33:11 by StephenWeeks.

CommonSubexp

CommonSubexp is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

It eliminates instances of common subexpressions.

Implementation

common-subexp.sig common-subexp.fun

Details and Notes

In addition to getting the usual sorts of things like

- ```
  (w + 0wx1) + (w + 0wx1)
  ```

  rewritten to

  ```
  let val w' = w + 0wx1 in w' + w' end
  ```

it also gets things like

- ```
  val a = Array_array n
  val b = Array_length a
  ```

  rewritten to

  ```
  val a = Array_array n
  val b = n
  ```

`Arith` transfers are handled specially. The *result* of an `Arith` transfer can be used in *common* `Arith` transfers that it dominates:

- ```
  val l = (n + m) + (n + m)

  val k = (l + n) + ((l + m) handle Overflow => ((l + m)
                                handle Overflow => l + n))
  ```

  is rewritten so that `(n + m)` is computed exactly once, as

are `(l + n)` and `(l + m)`.

Last edited on 2005-11-30 23:34:09 by StephenWeeks.
  ```

CompilationManager

The Compilation Manager (CM) is SML/NJ's mechanism for supporting programming-in-the-very-large. To aid in porting code from SML/NJ and in developing code simultaneously with MLton and SML/NJ, MLton supports a very limited subset of CM files. From MLton's point of view, a CM file `foo.cm` defines a list of SML source files. The call

```
mlton foo.cm
```

is equivalent to compiling an SML program consisting of the concatenation of these files. As always with MLton, the concatenation must be the whole program you wish to compile.

In its simplest form, a CM file contains the keywords `Group is` followed by an explicit list of sml files. For example, if `foo.cm` contains

```
Group is
bar.sig
bar.fun
main.sml
```

then a call `mlton foo.cm` is equivalent to concatenating the three files together and calling MLton on that SML file. The list of files defined by a CM file is the same as the order in which the filenames appear in the CM file. Thus, to MLton, order in a CM file matters. In the above example, if `main.sml` refers to a structure defined in `bar.fun`, then `main.sml` must appear after `bar.fun` in the file list.

CM files can also refer to other CM files. A reference to `bar.cm` from within `foo.cm` means to include all of the SML files defined by `bar.cm` before any of the subsequent files in `foo.cm`. For example if `foo.cm` contains

```
Group is
bar.cm
main.sml
```

and `bar.cm` contains

```
Group is
bar.sig
bar.fun
```

then a call to `mlton foo.cm` is equivalent to compiling the concatenation of `bar.sig`, `bar.fun`, and `main.sml`.

CM also has a preprocessor mechanism that allows files to be conditionally included. This can be useful when developing code with SML/NJ and MLton. In SML/NJ, the preprocessor defines the symbol `SMLNJ_VERSION`. In MLton, no symbols are defined. So, to conditionally include `foo.sml` when compiling under SML/NJ, one can use the following pattern.

```
# if (defined(SMLNJ_VERSION))
foo.sml
# endif
```

To conditionally include `foo.sml` when compiling under MLton, one can negate the test.

```
# if (! defined(SMLNJ_VERSION))
foo.sml
# endif
```

The filenames listed in a CM file can be either absolute paths or relative paths, in which case they are interpreted relative to the directory containing the CM file. If a CM file refers either directly or indirectly to an SML source file in more than one way, only the first occurrence of the file is included. Finally, the only valid file suffixes in a CM file are `.cm`, `.fun`, `.sig`, and `.sml`.

Comparison with CM

If you are unfamiliar with CM under SML/NJ, then you can skip this section.

MLton supports the full syntax of CM as of SML/NJ version 110.9.1. Extensions since then are unsupported. Also, many of the syntactic constructs are ignored. The most important difference between the two is that order in CM files matters to MLton but not to SML/NJ, which performs automatic dependency analysis. Also, CM supports export filters, which restricts the visibility of modules. MLton ignores export filters. As a consequence, it is possible that a program that is accepted by SML/NJ's CM might not be accepted by MLton's CM. In this case, you will have to manually reorder the files and possibly rename modules so that the concatenation of the files is the program you intend.

CM performs cutoff recompilation to avoid recompiling the entire program, while MLton always compiles the entire program. CM makes a distinction between groups and libraries, which MLton does not. CM supports other tools like lex and yacc, while MLton does not. MLton relies on traditional makefiles to use other tools.

Porting SML/NJ CM files to MLton

If you have already created large projects using SML/NJ and CM, there may be a large number of file dependencies implicit in your sources that are not reflected in your CM files. Because MLton relies on ordering in CM files, your CM files probably will not work with MLton. To help in porting CM files to MLton, the MLton distribution includes the sources for a utility, `cmcat`, that will print an ordered list of files corresponding to a CM file. See `util/cmcat/cmcat.sml` for details. Building `cmcat` requires that you have already installed a recent version of SML/NJ.

Alternatively, you can convert your CM files to `.mlb` files. The MLton distribution includes the sources for a utility, `cm2mlb`, that will print an ML Basis file with essentially the same semantics as the CM file -- handling the full syntax of CM supported by your installed SML/NJ version and correctly handling export filters. When `cm2mlb` encounters a `.cm` import, it attempts to convert it to a corresponding `.mlb` import. CM anchored paths are translated to paths according to a default configuration file (`cm2mlb-map`). For example, the default configuration includes

```
$basis.cm/basis.cm  $(SML_LIB)/basis/basis.mlb
```

to ensure that a `$/basis.cm` import is translated to a `$(SML_LIB)/basis/basis.mlb` import. See `util/cm2mlb` for details. Building `cm2mlb` requires that you have already installed a recent version of SML/NJ.

Last edited on 2005-11-30 23:40:40 by StephenWeeks.

CompileTimeOptions

MLton's compile-time options control the name of the output file, the verbosity of compile-time messages, and whether or not certain optimizations are performed. They also can specify which intermediate files are saved and can stop the compilation process early, at some intermediate pass, in which case compilation can be resumed by passing the generated files to MLton. MLton uses the input file suffix to determine the type of input program. The possibilities are `.c`, `.cm`, `.mlb`, `.o`, and `.sml`.

With no arguments, MLton prints the version number and exits. For a usage message, run MLton with an invalid switch, e.g. `mlton -z`. In the explanation below and in the usage message, for flags that take a number of choices (e.g. `{true|false}`), the first value listed is the default.

Options

- `-align {4|8}`
  Aligns object sizes and doubles in memory by the specified alignment. The default varies depending on architecture.
- `-as-opt` *option*
  Pass *option* to `gcc` when assembling.
- `-cc-opt` *option*
  Pass *option* to `gcc` when compiling C code.
- `-codegen {native|bytecode|c}`
  Generate native code, byte code, or C code. With `-codegen native`, MLton typically compiles more quickly and generates better code.
- `-const '`*name value*`'`
  Set the value of a compile-time constant. Here is a list of available constants, their default values, and what they control.
    - `Exn.keepHistory {false|true}`
  Enable `MLton.Exn.history`. See MLtonExn for details. There is a performance cost to setting this to `true`, both in memory usage of exceptions and in run time, because of additional work that must be performed at each exception construction, raise, and handle.
- `-default-ann` *ann*
  Specify default ML Basis annotations. For example, `-default-ann 'warnUnused true'` causes unused variable warnings to be enabled by default. A default is overridden by the corresponding annotation in an ML Basis file.
- `-disable-ann` *ann*
  Ignore the specified ML Basis annotation in every ML Basis file. For example, to see *all* match and unused warnings, compile with

```
-default-ann 'warnUnused true'
-disable-ann forceUsed
-disable-ann nonexhaustiveMatch
-disable-ann redundantMatch
-disable-ann warnUnused
```

- `-export-header` *file*
  Write to *file* C prototypes for all of the functions in the program exported from SML to C.
- `-ieee-fp {false|true}`
  Cause the native code generator to be pedantic about following the IEEE floating point standard. By default, it is not, because of the performance cost. This only has an effect with `-codegen native`.

- `-inline n`
  Set the inlining threshold used in the optimizer. The threshold is an approximate measure of code size of a procedure. The default is `320`.
- `-keep {g|o|sml}`
  Save intermediate files. If no `-keep` argument is given, then only the output file is saved.

`g`	generated `.S` and `.c` files passed to gcc and the assembler
`o`	object (`.o`) files
`sml`	SML file

- `-link-opt option`
  Pass *option* to `gcc` when linking. You can use this to specify library search paths, e.g. `-link-opt -Lpath`, and libraries to link with, e.g. `-link-opt -lfoo`, or even both at the same time, e.g. `-link-opt '-Lpath -lfoo'`. If you wish to pass an option to the linker, you must use `gcc`'s `-Wl,` syntax, e.g., `-link-opt '-Wl,--export-dynamic'`.
- `-mlb-path-map file`
  Use *file* as an ML Basis path map to define additional MLB path variables. Multiple uses of `-mlb-path-map` are allowed, with variable definitions in later path maps taking precedence over earlier ones.
- `-output file`
  Specify the name of the final output file. The default name is the input file name with its suffix removed and an appropriate, possibly empty, suffix added.
- `-profile {no|alloc|count|time}`
  Produce an executable that gathers Profiling data. When such an executable is run, it produces an `mlmon.out` file.
- `-profile-branch {false|true}`
  If true, the profiler will separately gather profiling data for each branch of a function definition, `case` expression, and `if` expression.
- `-profile-stack {false|true}`
  If `true`, the executable will gather profiling data for all functions on the stack, not just the currently executing function. See ProfilingTheStack.
- `-runtime arg`
  Pass argument to the runtime system via `@MLton`. See RunTimeOptions. The argument will be processed before other `@MLton` command line switches. Multiple uses of `-runtime` are allowed, and will pass all the arguments in order. If the same runtime switch occurs more than once, then the last setting will take effect. There is no need to supply the leading `@MLton` or the trailing `--`; these will be supplied automatically.

  An argument to `-runtime` may contain spaces, which will cause the argument to be treated as a sequence of words by the runtime. For example the command line:

  ```
  mlton -runtime 'ram-slop 0.4' foo.sml
  ```

  will cause `foo` to run as if it had been called like:

  ```
  foo @MLton ram-slop 0.4 --
  ```

  An executable created with `-runtime stop` doesn't process any `@MLton` arguments. This is useful to create an executable, e.g. `echo`, that must treat `@MLton` like any other command-line argument.

  ```
  % mlton -runtime stop echo.sml
  ```

```
% echo @MLton --
@MLton --
```

- `-show-basis` *file*
  Pretty print to *file* the basis defined by the input program. See ShowBasis.
- `-show-def-use` *file*
  Output def-use information to *file*. Each identifier that is defined appears on a line, followed on subsequent lines by the position of each use.
- `-stop {f|g|o|sml|tc}`
  Specify when to stop.

`f`	list of files on stdout (only makes sense when input is `foo.cm` or `foo.mlb`)
`g`	generated `.S` and `.c` files
`o`	object (`.o`) files
`sml`	SML file (only makes sense when input is `foo.cm` or `foo.mlb`)
`tc`	after type checking

  If you compile with `-stop g` or `-stop o`, you can resume compilation by running MLton on the generated `.c` and `.S` or `.o` files.

- `-target {self|...}`
  Generate an executable that runs on the specified platform. The default is `self`, which means to compile for the machine that MLton is running on. To use any other target, you must first install a cross compiler.
- `-target-as-opt` *target option*
  Like `-as-opt`, this passes *option* to `gcc` when assembling, except it only passes *option* when the target architecture or operating system is *target*. Valid values for *target* are: `hppa`, `powerpc`, `sparc`, `x86`, `cygwin`, `darwin`, `freebsd`, `linux`, `mingw`, `netbsd`, `openbsd`, `solaris`.
- `-target-cc-opt` *target option*
  Like `-cc-opt`, this passes *option* to `gcc` when compiling C code, except it only passes *option* when the target architecture or operating system is *target*. Valid values for *target* are as for `-target-as-opt`.
- `-target-link-opt` *target option*
  Like `-link-opt`, this passes *option* to `gcc` when linking, except it only passes *option* when the target architecture or operating system is *target*. Valid values for *target* are as for `-target-as-opt`.
- `-verbose {0|1|2|3}`
  How verbose to be about what passes are running. The default is `0`.

`0`	silent
`1`	calls to compiler, assembler, and linker
`2`	1, plus intermediate compiler passes
`3`	2, plus some data structure sizes

Last edited on 2005-12-02 06:14:04 by StephenWeeks.

CompilerOverview

The following table shows the overall structure of the compiler. IntermediateLanguages are shown in the center column. The names of compiler passes are listed in the left and right columns.

Compiler Overview

Translation Passes	IntermediateLanguage	*Optimization Passes*
	Source	
FrontEnd		
	AST	
Elaborate		
	CoreML	CoreMLSimplify
Defunctorize		
	XML	XMLSimplify
Monomorphise		
	SXML	SXMLSimplify
ClosureConvert		
	SSA	SSASimplify
ToSSA2		
	SSA2	SSA2Simplify
ToRSSA		
	RSSA	RSSASimplify
ToMachine		
	Machine	

The `Compile` functor (compile.sig , compile.fun), controls the high-level view of the compiler passes, from FrontEnd to code generation.

Last edited on 2005-08-19 15:41:28 by MatthewFluet.

CompilerPassTemplate

An analysis pass for the ZZZ IntermediateLanguage, invoked from ZZZOtherPass. An implementation pass for the ZZZ IntermediateLanguage, invoked from ZZZSimplify. An optimization pass for the ZZZ IntermediateLanguage, invoked from ZZZSimplify. A rewrite pass for the ZZZ IntermediateLanguage, invoked from ZZZOtherPass. A translation pass from the ZZA IntermediateLanguage to the ZZB IntermediateLanguage.

Description

A short description of the pass.

Implementation

ZZZ.sig ZZZ.fun

Details and Notes

Relevant details and notes.

Last edited on 2005-12-02 04:18:14 by StephenWeeks.

CompilingWithSMLNJ

You can compile MLton with SML/NJ, however the resulting compiler will run much more slowly than MLton compiled by itself. We don't recommend using SML/NJ as a means of porting MLton to a new platform or bootstrapping on a new platform.

If you do want to build MLton with SML/NJ, it is best to have a binary MLton package installed. If you don't, here are some issues you may encounter when you run `make nj-mlton`.

You will get (many copies of) the error message:

```
/bin/sh: mlton: not found
```

The `Makefile` calls `mlton` to determine dependencies, and can proceed in spite of this error.

If you don't have a `mlton` executable, you will get the error message:

```
Error: cannot upgrade basis because the compiler doesn't work
```

We use `upgrade-basis.sml` to work around basis library differences, allowing a version of MLton written for a newer basis library to be compiled by an older version of MLton. The file isn't necessary when building with SML/NJ, but is listed in `$(SOURCES)`, so the `Makefile` is attempting to build it. Building `upgrade-basis.sml` requires the old version of MLton to be around so that the right stubs can be built.

To work around this problem, do one of the following.

- Manually tweak sources to remove $(UP) until you're finished building MLton with SML/NJ and have a working MLton.
- Build `upgrade-basis.sml` on some other machine with a working MLton and copy it over.

If you don't have an `mllex` executable, you will get the error message:

```
mllex: Command not found
```

Building MLton requires `mllex` and `mlyacc` executables, which are distributed with a binary package of MLton. The easiest solution is to copy the front-end lexer/parser files from a different machine (`ml.grm.sml`, `ml.grm.sig`, `ml.lex.sml`, `mlb.grm.sig`, `mlb.grm.sml`).

Last edited on 2006-10-19 17:06:01 by StephenWeeks.

ConcurrentML

Concurrent ML is an SML concurrency library based on synchronous message passing. MLton has an initial port of CML from SML/NJ, but is missing a thread-safe wrapper around the Basis Library and event-based equivalents to `IO` and `OS` functions.

All of the core CML functionality is present.

```
structure CML: CML
structure SyncVar: SYNC_VAR
structure Mailbox: MAILBOX
structure Multicast: MULTICAST
structure SimpleRPC: SIMPLE_RPC
structure RunCML: RUN_CML
```

The `RUN_CML` signature is minimal.

```
signature RUN_CML =
   sig
      val isRunning: unit -> bool
      val doit: (unit -> unit) * Time.time option -> OS.Process.status
      val shutdown: OS.Process.status -> 'a
   end
```

MLton's `RunCML` structure does not include all of the cleanup and logging operations of SML/NJ's `RunCML` structure. However, the implementation does include the `CML.timeOutEvt` and `CML.atTimeEvt` functions, and a preemptive scheduler that knows to sleep when there are no ready threads and some threads blocked on time events.

Because MLton does not wrap the Basis Library for CML, the "right" way to call a Basis Library function that is stateful is to wrap the call with `MLton.Thread.atomically`.

Usage

- You can import the CML Library into an MLB file with `$(SML_LIB)/cml/cml.mlb`
- If you are porting a project from SML/NJ's CompilationManager to MLton's ML Basis system using `cm2mlb`, note that the following map is included by default:

  ```
  $cml/cml.cm  $(SML_LIB)/cml/cml.mlb
  ```

  This will automatically convert a `$cml/cml.cm` import in an input `.cm` file into a `$(SML_LIB)/cml/cml.mlb` import in the output `.mlb` file.

Also see

- ConcurrentMLImplementation
- eXene

Last edited on 2005-12-02 03:33:39 by StephenWeeks.

ConcurrentMLImplementation

Here are some notes on MLton's implementation of ConcurrentML.

Concurrent ML was originally implemented for SML/NJ. It was ported to MLton in the summer of 2004. The main difference between the implementations is that SML/NJ uses continuations to implement CML threads, while MLton uses its underlying thread package. Presently, MLton's threads are a little more heavyweight than SML/NJ's continuations, but it's pretty clear that there is some fat there that could be trimmed.

The implementation of CML in SML/NJ is built upon the first-class continuations of the `SMLofNJ.Cont` module.

```
type 'a cont
val callcc: ('a cont -> 'a) -> 'a
val isolate: ('a -> unit) -> 'a cont
val throw: 'a cont -> 'a -> 'b
```

The implementation of CML in MLton is built upon the first-class threads of the MLtonThread module.

```
type 'a t
val new: ('a -> unit) -> 'a t
val prepare: 'a t * 'a -> Runnable.t
val switch: ('a t -> Runnable.t) -> 'a
```

The port is relatively straightforward, because CML always throws to a continuation at most once. Hence, an "abstract" implementation of CML could be built upon first-class one-shot continuations, which map equally well to SML/NJ's continuations and MLton's threads.

The "essence" of the port is to transform:

```
callcc (fn k => ... throw k' v')
```

to

```
switch (fn t => ... prepare (t', v'))
```

which suffices for the vast majority of the CML implementation.

There was only one complicated transformation: blocking multiple base events. In SML/NJ CML, the representation of base events is given by:

```
datatype 'a event_status
  = ENABLED of {prio: int, doFn: unit -> 'a}
  | BLOCKED of {
        transId: trans_id ref,
        cleanUp: unit -> unit,
        next: unit -> unit
      } -> 'a
type 'a base_evt = unit -> 'a event_status
```

When synchronizing on a set of base events, which are all blocked, we must invoke each `BLOCKED` function with the same transId and cleanUp (the transId is (checked and) set to CANCEL by the cleanUp function, which is invoked by the first enabled event; this "fizzles" every other event in the synchronization group that later becomes enabled). However, each `BLOCKED` function is implemented by a callcc, so that when the event is enabled, it throws back to the point of synchronization. Hence, the next function (which doesn't return) is invoked by the `BLOCKED` function to escape the callcc and continue in the thread performing the synchronization. In SML/NJ this is implemented as follows:

```
fun ext ([], blockFns) = callcc (fn k => let
      val throw = throw k
      val (transId, setFlg) = mkFlg()
      fun log [] = S.atomicDispatch ()
        | log (blockFn:: r) =
            throw (blockFn {
                transId = transId,
                cleanUp = setFlg,
                next = fn () => log r
              })
      in
        log blockFns; error "[log]"
      end)
```

(Note that `S.atomicDispatch` invokes the continuation of the next continuation on the ready queue.) This doesn't map well to the MLton thread model. Although it follows the

```
callcc (fn k => ... throw k v)
```

model, the fact that blockFn will also attempt to do

```
callcc (fn k' => ... next ())
```

means that the naive transformation will result in nested switch-es.

We need to think a little more about what this code is trying to do. Essentially, each `blockFn` wants to capture this continuation, hold on to it until the event is enabled, and continue with next; when the event is enabled, before invoking the continuation and returning to the synchronization point, the cleanUp and other event specific operations are performed.

To accomplish the same effect in the MLton thread implementation, we have the following:

```
datatype 'a status =
   ENABLED of {prio: int, doitFn: unit -> 'a}
 | BLOCKED of {transId: trans_id,
               cleanUp: unit -> unit,
               next: unit -> rdy_thread} -> 'a

type 'a base = unit -> 'a status

fun ext ([], blockFns): 'a =
     S.atomicSwitch
     (fn (t: 'a S.thread) =>
      let
         val (transId, cleanUp) = TransID.mkFlg ()
         fun log blockFns: S.rdy_thread =
            case blockFns of
               [] => S.next ()
             | blockFn::blockFns =>
                  (S.prep o S.new)
                  (fn _ => fn () =>
                   let
```

```
                    val () = S.atomicBegin ()
                    val x = blockFn {transId = transId,
                                     cleanUp = cleanUp,
                                     next = fn () => log blockFns}
                 in S.switch(fn _ => S.prepVal (t, x))
                 end)
   in
      log blockFns
   end)
```

To avoid the nested switch-es, I run the blockFn in it's own thread, whose only purpose is to return to the synchronization point. This corresponds to the `throw (blockFn {...})` in the SML/NJ implementation. I'm worried that this implementation might be a little expensive, starting a new thread for each blocked event (when there are only multiple blocked events in a synchronization group). But, I don't see another way of implementing this behavior in the MLton thread model.

Note that another way of thinking about what is going on is to consider each `blockFn` as prepending a different set of actions to the thread `t`. It might be possible to give a `MLton.Thread.unsafePrepend`.

```
fun unsafePrepend (T r: 'a t, f: 'b -> 'a): 'b t =
   let
      val t =
         case !r of
            Dead => raise Fail "prepend to a Dead thread"
          | New g => New (g o f)
          | Paused (g, t) => Paused (fn h => g (f o h), t)
   in (* r := Dead; *)
      T (ref t)
   end
```

I have commented out the `r := Dead`, which would allow multiple prepends to the same thread (i.e., not destroying the original thread in the process). Of course, only one of the threads could be run: if the original thread were in the `Paused` state, then multiple threads would share the underlying runtime/primitive thread. Now, this matches the "one-shot" nature of CML continuations/threads, but I'm not comfortable with extending `MLton.Thread` with such an unsafe operation.

Other than this complication with blocking multiple base events, the port was quite routine. (As a very pleasant surprise, the CML implementation in SML/NJ doesn't use any SML/NJ-isms.) There is a slight difference in the way in which critical sections are handled in SML/NJ and MLton; since `MLton.Thread.switch` _always_ leaves a critical section, it is sometimes necessary to add additional `atomicBegin`/`Ends` to ensure that we remain in a critical section after a thread switch.

While looking at virtually every file in the core CML implementation, I took the liberty of simplifying things where it seemed possible; in terms of style, the implementation is about half-way between Reppy's original and MLton's.

Some changes of note:

- `util/` contains all pertinent data-structures: (functional and imperative) queues, (functional) priority queues. Hence, it should be easier to switch in more efficient or real-time implementations.
- `core-cml/scheduler.sml`: in both implementations, this is where most of the interesting action takes place. I've made the connection between `MLton.Thread.ts` and `ThreadId.thread_ids` more abstract than it is in the SML/NJ implementation, and encapsulated all of the `MLton.Thread` operations in this module.

- eliminated all of the "by hand" inlining

Future Extensions

The CML documentation says the following:

```
CML.joinEvt: thread_id -> unit event
```

> `joinEvt tid`

creates an event value for synchronizing on the termination of the thread with the ID tid. There are three ways that a thread may terminate: the function that was passed to spawn (or spawnc) may return; it may call the exit function, or it may have an uncaught exception. Note that `joinEvt` does not distinguish between these cases; it also does not become enabled if the named thread deadlocks (even if it is garbage collected).

I believe that the `MLton.Finalizable` might be able to relax that last restriction. Upon the creation of a `'a Scheduler.thread`, we could attach a finalizer to the underlying `'a MLton.Thread.t` that enables the `joinEvt` (in the associated `ThreadID.thread_id`) when the `'a MLton.Thread.t` becomes unreachable.

I don't know why CML doesn't have

```
CML.kill: thread_id -> unit
```

which has a fairly simple implementation -- setting a kill flag in the `thread_id` and adjusting the scheduler to discard any killed threads that it takes off the ready queue. The fairness of the scheduler ensures that a killed thread will eventually be discarded. The semantics are little murky for blocked threads that are killed, though. For example, consider a thread blocked on `SyncVar.mTake mv` and a thread blocked on `SyncVar.mGet mv`. If the first thread is killed while blocked, and a third thread does `SyncVar.mPut (mv, x)`, then we might expect that we'll enable the second thread, and never the first. But, when only the ready queue is able to discard killed threads, then the `SyncVar.mPut` could enable the first thread (putting it on the ready queue, from which it will be discarded) and leave the second thread blocked. We could solve this by adjusting the `TransID.trans_id types` and the "cleaner" functions to look for both canceled transactions and transactions on killed threads.

John Reppy says that MarlowEtAl01 and FlattFindler04 explain why `CML.kill` would be a bad idea.

Between `CML.timeOutEvt` and `CML.kill`, one could give an efficient solution to the recent comp.lang.ml post about terminating a function that doesn't complete in a given time.

```
fun timeOut (f: unit -> 'a, t: Time.time): 'a option =
  let
     val iv = SyncVar.iVar ()
     val tid = CML.spawn (fn () => SyncVar.iPut (iv, f ()))
  in
     CML.select
     [CML.wrap (CML.timeOutEvt t, fn () => (CML.kill tid; NONE)),
      CML.wrap (SyncVar.iGetEvt iv, fn x => SOME x)]
  end
```

Space Safety

There are some CML related posts on the MLton mailing list

http://mlton.org/pipermail/mlton/2004-May/

that discuss concerns that SML/NJ's implementation is not space efficient, because multi-shot continuations can be held indefinitely on event queues. MLton is better off because of the one-shot nature -- when an event enables a thread, all other copies of the thread waiting in other event queues get turned into dead threads (of zero size).

Last edited on 2005-12-02 04:18:52 by StephenWeeks.

ConstantPropagation

Constant propagation is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This is whole-program constant propagation, even through data structures. It also performs globalization of (small) values computed once.

Uses Multi.

Implementation

constant-propagation.sig constant-propagation.fun

Details and Notes

Last edited on 2005-12-01 02:56:45 by StephenWeeks.

Contact

Mailing lists

There are two mailing lists available.

- MLton@mlton.org (subscribe, archive) MLton developers
- MLton-user@mlton.org (subscribe, archive) MLton user community

In addition to the pipermail archive at mlton.org, there are archives of both MLton and MLton-user that use Lurker.

Mailing list policy

- Both mailing lists are unmoderated. However, we use a whitelist to prevent spam. So, the first time you send to the list, your mail will be delayed until we add you to the whitelist.
- Large messages (over 256K) should not be sent. Rather, please send an email containing the discussion text and a link to any large files. You may use our TemporaryUpload page for uploading these files.
- Very active `MLton@mlton.org` list members who might otherwise be expected to provide a fast response should send a message when they will be offline for more than a few days. The convention is to put "*userid* offline until *date*" in the subject line to make it easy to scan.

IRC

- Some MLton developers and users are in channel `#sml` on http://freenode.net.

Last edited on 2005-12-01 02:58:05 by StephenWeeks.

Contify

Contify is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

Contification is a compiler optimization that turns a function that always returns to the same place into a continuation. This exposes control-flow information that is required by many optimizations, including traditional loop optimizations.

Implementation

contify.sig contify.fun

Details and Notes

See Contification Using Dominators. The intermediate language described in that paper has since evolved to the SSA IntermediateLanguage; hence, the complication described in Section 6.1 is no longer relevant.

Last edited on 2005-12-01 02:59:40 by StephenWeeks.

CoreML

Core ML is an IntermediateLanguage, translated from AST by Elaborate, optimized by CoreMLSimplify, and translated by Defunctorize to XML.

Description

CoreML is polymorphic, higher-order, and has nested patterns.

Implementation

core-ml.sig core-ml.fun

Type Checking

The CoreML IntermediateLanguage has no independent type checker.

Details and Notes

Last edited on 2005-12-01 03:00:12 by StephenWeeks.

CoreMLSimplify

The single optimization pass for the CoreML IntermediateLanguage is controlled by the `Compile` functor (compile.fun).

The following optimization pass is implemented:

- DeadCode

Last edited on 2005-08-19 15:40:09 by MatthewFluet.

CreatingPages

To create a page on this WebSite, edit an existing page, and add the name of the new page, like `FooBar`, to the page contents. When you view the new version of the existing page, a link will have been automatically created, and if you click on it, you will be given the option to create the new page.

You can also go directly to a new page by entering the page name as a URL into your browser, like `http://mlton.org/FooBar`.

You can also type in the page name here to go directly to that page.

Last edited on 2005-12-01 03:02:19 by StephenWeeks.

Credits

MLton was designed and implemented by HenryCejtin, MatthewFluet, SureshJagannathan, and StephenWeeks.

- HenryCejtin wrote the `IntInf` implementation, the original profiler, the original man pages, the `.spec` files for the RPMs, and lots of little hacks to speed stuff up.
- MatthewFluet implemented the X86 native code generator, ported `mlprof` to work with the native code generator, did a lot of work on the SSA optimizer, both adding new optimizations and improving or porting existing optimizations, updated the Basis Library implementation, ported ConcurrentML and ML-NLFFI to MLton, and implemented the ML Basis system.
- SureshJagannathan implemented some early inlining and uncurrying optimizations.
- StephenWeeks implemented most of the original version of MLton, and continues to keep his fingers in most every part.

Many people have helped us over the years. Here is an alphabetical list.

- JesperLouisAndersen sent several patches to improve the runtime on FreeBSD and ported MLton to run on NetBSD and OpenBSD.
- JohnnyAndersen implemented `BinIO`, modified MLton so it could cross compile to MinGW, and provided useful discussion about cross-compilation.
- Alain Deutsch and PolySpace Technologies provided many bug fixes and runtime system improvements, code to help the Sparc/Solaris port, and funded a number of improvements to MLton.
- Martin Elsman provided helpful discussions in the development of the ML Basis system.
- Brent Fulgham ported MLton most of the way to MinGW.
- Adam Goode provided the script to build the PDF MLton Guide.
- Simon Helsen provided bug reports, suggestions, and helpful discussions.
- Joe Hurd provided useful discussion and feedback on source-level profiling.
- VesaKarvonen contributed `esml-mode.el` (see Emacs) and patches for improving match warnings.
- Richard Kelsey provided helpful discussions.
- Ville Laurikari ported MLton to HPPA/HP-UX.
- Geoffrey Mainland helped with FreeBSD packaging.
- Eric McCorkle ported MLton to Intel Mac.
- TomMurphy wrote the original version of `MLton.Syslog` as part of his `mlftpd` project, and has sent many useful bug reports and suggestions.
- Michael Neumann helped to patch the runtime to compile under FreeBSD.
- Barak Pearlmutter built the original Debian package for MLton, and helped us to take over the process.
- Filip Pizlo ported MLton to Darwin.
- Sam Rushing ported MLton to FreeBSD.
- Jeffrey Mark Siskind provided helpful discussions and inspiration with his Stalin Scheme compiler.
- WesleyTerpstra added support for `MLton.Process.create`, made a number of contributions to the ForeignFunctionInterface, and contributed a number of other runtime system patches.
- Luke Ziarek assisted in porting MLton to Darwin.

We have also benefited from other software development tools and used code from other sources.

- MLton was developed using Standard ML of New Jersey and the Compilation Manager (CM)
- MLton's lexer (`mlton/frontend/ml.lex`), parser (`mlton/frontend/ml.grm`), and precedence-parser (`mlton/elaborate/precedence-parse.fun`) are modified versions of

code from SML/NJ.

- The MLton Basis Library implementation of conversions between binary and decimal representations of reals uses David Gay's gdtoa library.
- The MLton Basis Library implementation uses modified versions of portions of the the SML/NJ Basis Library implementation modules `OS.IO`, `Posix.IO`, `Process`, and `Unix`.
- The MLton Basis Library implementation uses modified versions of portions of the ML Kit Version 4.1.4 Basis Library implementation modules `Path`, `Time`, and `Date`.
- Many of the benchmarks come from the SML/NJ benchmark suite.
- Many of the regression tests come from the ML Kit Version 4.1.4 distribution, which borrowed them from the Moscow ML distribution.
- MLton uses the [http://www.gnu.org/software/gmp/gmp.html GNU multiprecision library] for its implementation of `IntInf`.
- MLton's implementation of mllex, mlyacc, the ckit Library, Concurrent ML, and ML-NLFFI are modified versions of code from SML/NJ.

Last edited on 2006-10-11 18:32:42 by StephenWeeks.

CrossCompiling

MLton's `-target` flag directs MLton to cross compile an application for another platform. By default, MLton is only able to compile for the machine it is running on. In order to use MLton as a cross compiler, you need to do two things.

1. Install the GCC cross-compiler tools on the host so that GCC can compile to the target.
2. Cross compile the MLton runtime system to build the runtime libraries for the target.

To make the terminology clear, we refer to the *host* as the machine MLton is running on and the *target* as the machine that MLton is compiling for.

To build a GCC cross-compiler toolset on the host, you can use the script `bin/build-cross-gcc`, available in the MLton sources, as a template. The value of the `target` variable in that script is important, since that is what you will pass to MLton's `-target` flag. Once you have the toolset built, you should be able to test it by cross compiling a simple hello world program on your host machine.

```
% gcc -b i386-pc-cygwin -o hello-world hello-world.c
```

You should now be able to run `hello-world` on the target machine, in this case, a Cygwin machine.

Next, you must cross compile the MLton runtime system and inform MLton of the availability of the new target. The script `bin/add-cross` from the MLton sources will help you do this. Please read the comments at the top of the script. Here is a sample run adding a Solaris cross compiler.

```
% add-cross sparc-sun-solaris sun blade
Making runtime.
Building print-constants executable.
Running print-constants on blade.
```

Running `add-cross` uses `ssh` to compile the runtime on the target machine and to create `print-constants`, which prints out all of the constants that MLton needs in order to implement the Basis Library. The script runs `print-constants` on the target machine (`blade` in this case), and saves the output.

Once you have done all this, you should be able to cross compile SML applications. For example,

```
mlton -target i386-pc-cygwin hello-world.sml
```

will create `hello-world`, which you should be able to run from a Cygwin shell on your Windows machine.

Cross-compiling alternatives

Building and maintaining cross-compiling `gcc`'s is complex. You may find it simpler to use `mlton -keep g` to generate the files on the host, then copy the files to the target, and then use `gcc` or `mlton` on the target to compile the files.

Last edited on 2005-12-02 04:19:16 by StephenWeeks.

DeadCode

Dead-code elimination is an optimization pass for the CoreML IntermediateLanguage, invoked from CoreMLSimplify.

Description

This pass eliminates declarations from the Basis Library not needed by the user program.

Implementation

dead-code.sig dead-code.fun

Details and Notes

In order to compile small programs rapidly, a pass of dead code elimination is run in order to eliminate as much of the Basis Library as possible. The dead code elimination algorithm used is not safe in general, and only works because the Basis Library implementation has special properties:

- it terminates
- it performs no I/O

The dead code elimination includes the minimal set of declarations from the Basis Library so that there are no free variables in the user program (or remaining Basis Library implementation). It has a special hack to include all bindings of the form:

```
val _ = ...
```

There is an ML Basis annotation, `deadCode true`, that governs which code is subject to this unsafe dead-code elimination.

Last edited on 2005-12-01 03:28:11 by StephenWeeks.

DeepFlatten

Deep flatten is an optimization pass for the SSA2 IntermediateLanguage, invoked from SSA2Simplify.

Description

This pass flattens into mutable fields of objects and into vectors.

For example, an `(int * int) ref` is represented by a 2 word object, and an `(int * int) array` contains pairs of `int`s, rather than pointers to pairs of `int`s.

Implementation

deep-flatten.sig deep-flatten.fun

Details and Notes

Last edited on 2005-12-01 03:29:16 by StephenWeeks.

DefineTypeBeforeUse

Standard ML requires types to be defined before they are used. Because of type inference, the use of a type can be implicit; hence, this requirement is more subtle than it might appear. For example, the following program is not type correct, because the type of `r` is `t option ref`, but `t` is defined after `r`.

```
val r = ref NONE
datatype t = A | B
val () = r := SOME A
```

MLton reports the following error, indicating that the type defined on line 2 is used on line 1.

```
Error: z.sml 1.1.
  Type escapes the scope of its definition at z.sml 2.10.
    type: t
    in: val r = ref NONE
```

While the above example is benign, the following example shows how to cast an integer to a function by (implicitly) using a type before it is defined. In the example, the ref cell `r` is of type `t option ref`, where `t` is defined *after* `r`, as a parameter to functor `F`. This example causes PolyML 4.1.3 to seg fault.

```
val r = ref NONE
functor F (type t
           val x: t) =
   struct
      val () = r := SOME x
      fun get () = valOf (!r)
   end
structure S1 = F (type t = unit -> unit
                  val x = fn () => ())
structure S2 = F (type t = int
                  val x = 13)
val () = S1.get () ()
```

MLton reports the following error.

```
Warning: z.sml 1.1.
  Unable to locally determine type of variable: r.
    type: ??? option ref
    in: val r = ref NONE
Error: z.sml 1.1.
  Type escapes the scope of its definition at z.sml 2.17.
    type: t
    in: val r = ref NONE
```

Last edited on 2005-12-01 03:38:39 by StephenWeeks.

DefinitionOfStandardML

The Definition of Standard ML (Revised) is a terse and formal specification of Standard ML's syntax and semantics. The language specified by this book is often referred to as SML 97.

There is an older version of the definition, published in 1990, which has an accompanying commentary that introduces and explains the notation and approach. The same notation is used in the SML 97 definition, so it is worth purchasing the older definition and commentary if you intend a close study of the definition.

Last edited on 2004-12-28 19:55:24 by StephenWeeks.

Defunctorize

Defunctorize is a translation pass from the CoreML IntermediateLanguage to the XML IntermediateLanguage.

Description

This pass converts a CoreML program to an XML program by performing:

- linearization
- MatchCompile
- LookupConstants
- polymorphic `val` dec expansion
- `datatype` lifting (to the top-level)

Implementation

defunctorize.sig defunctorize.fun

Details and Notes

This pass is grossly misnamed and does not perform defunctorization.

Datatype Lifting

This pass moves all `datatype` declarations to the top level.

Standard ML `datatype` declarations can contain type variables that are not bound in the declaration itself. For example, the following program is valid.

```
fun 'a f (x: 'a) =
   let
      datatype 'b t = T of 'a * 'b
      val y: int t = T (x, 1)
   in
      13
   end
```

Unfortunately, the `datatype` declaration can not be immediately moved to the top level, because that would leave `'a` free.

```
datatype 'b t = T of 'a * 'b
fun 'a f (x: 'a) =
   let
      val y: int t = T (x, 1)
   in
      13
   end
```

In order to safely move `datatype`s, this pass must close them, as well as add any free type variables as extra arguments to the type constructor. For example, the above program would be translated to the following.

```
datatype ('a, 'b) t = T of 'a * 'b
fun 'a f (x: 'a) =
   let
      val y: ('a, int) t = T (x, 1)
   in
      13
   end
```

Historical Notes

The Defunctorize pass originally eliminated Standard ML functors by duplicating their body at each application. These duties have been adopted by the Elaborate pass.

Last edited on 2005-12-02 04:19:26 by StephenWeeks.

Developers

Here is a picture of the MLton team at a meeting in Chicago in August 2003. From left to right we have:

StephenWeeks MatthewFluet HenryCejtin SureshJagannathan

image

Also see the Credits for a list of specific contributions.

Developers list

A number of people read the developers mailing list, MLton@mlton.org, and make contributions there. Here's a list of those who have a page here.

- AndreiFormiga
- JesperLouisAndersen
- JohnnyAndersen
- MichaelNorrish
- MikeThomas
- RayRacine
- WesleyTerpstra

Last edited on 2005-12-01 03:45:09 by StephenWeeks.

Development

This page is the central point for MLton development.

- Access the Sources.
- Ideas for Projects to improve MLton.
- Developers that are or have been involved in the project.
- Help maintain and improve the WebSite.

Notes

- CompilerOverview
- CrossCompiling
- License
- PortingMLton
- ReleaseChecklist
- SelfCompiling

Last edited on 2005-04-22 19:59:46 by StephenWeeks.

Documentation

Documentation is available on the following topics.

- Standard ML
    - Basis Library
    - Additional libraries
- Installing MLton
- Using MLton
    - Foreign function interface (FFI)
    - Manual page (compile-time options run-time options)
    - ML Basis system
    - MLton structure
    - Platform-specific notes
    - Profiling
    - Type checking
- About MLton
    - Credits
    - Drawbacks
    - Features
    - History
    - License
    - Talk
- MLLex pdf
- MLYacc pdf
- References

Last edited on 2005-12-01 19:30:55 by StephenWeeks.

Drawbacks

MLton has several drawbacks due to its use of whole-program compilation.

- Large compile-time memory requirement.
  Because MLton performs whole-program analysis and optimization, compilation requires a large amount of memory. For example, compiling MLton (over 140K lines) requires at least 512M RAM.
- Long compile times.
  Whole-program compilation can take a long time. For example, compiling MLton (over 140K lines) on a 1.6GHz machine takes five to ten minutes.
- No interactive top level.

Because of whole-program compilation, MLton does not provide an interactive top level. In particular, it does not implement the optional Basis Library function `use`.

Last edited on 2005-12-02 04:19:39 by StephenWeeks.

Eclipse

Eclipse is an open, extensible IDE.

ML-Dev is a plug-in for Eclipse, based on SML/NJ.

There has been some talk on the MLton mailing list about adding support to Eclipse for MLton/SML, and in particular, using http://eclipsefp.sourceforge.net/. We are unaware of any progress along those lines.

Last edited on 2006-02-09 19:33:33 by StephenWeeks.

EditingPages

You can help maintain this WebSite and improve a page's contents by using the "Edit" link found at the bottom of that page. Pages are written using MoinMoin's wiki markup language. You can practice editing in the WikiSandBox.

Before you begin editing, you must create a user account. When you do so, please also create a home page (like StephenWeeks) so we know who you are. See our AccessControl policy for who is allowed to edit what.

By contributing to this web site, you agree to dedicate your contribution to the public domain. For more details, please see our License.

Last edited on 2005-12-01 20:16:41 by StephenWeeks.

Elaborate

Elaborate is a translation pass from the AST IntermediateLanguage to the CoreML IntermediateLanguage.

Description

This pass performs type inference and type checking according to the Definition. It also defunctorizes the program, eliminating all module-level constructs.

Implementation

elaborate.sig elaborate.fun
elaborate-env.sig elaborate-env.fun
elaborate-modules.sig elaborate-modules.fun
elaborate-core.sig elaborate-core.fun
elaborate

Details and Notes

At the modules level, the Elaborate pass:

- elaborates signatures with interfaces (see interface.sig and interface.fun).
  The main trick is to use disjoint sets to efficiently handle sharing of tycons and of structures and then to copy signatures as dags rather than as trees.
- checks functors at the point of definition, using functor summaries to speed up checking of functor applications.
  When a functor is first type checked, we keep track of the dummy argument structure and the dummy result structure, as well as all the tycons that were created while elaborating the body. Then, if we later need to type check an application of the functor (as opposed to defunctorize an application), we pair up tycons in the dummy argument structure with the actual argument structure and then replace the dummy tycons with the actual tycons in the dummy result structure, yielding the actual result structure. We also generate new tycons for all the tycons that we created while originally elaborating the body.
- handles opaque signature constraints.

This is implemented by building a dummy structure realized from the signature, just as we would for a functor argument when type checking a functor. The dummy structure contains exactly the type information that is in the signature, which is what opacity requires. We then replace the variables (and constructors) in the dummy structure with the corresponding variables (and constructors) from the actual structure so that the translation to CoreML uses the right stuff. For each tycon in the dummy structure, we keep track of the corresponding type structure in the actual structure. This is used when producing the CoreML types (see `expandOpaque` in type-env.sig and type-env.fun).

Then, within each `structure` or `functor` body, for each declaration (`<dec>` in the Standard ML grammar), the Elaborate pass does three steps:

1. ScopeInference
2.
   - PrecedenceParse
   - `_{ex,im}port` expansion

- ♦ profiling insertion
- ♦ unification

3. Overloaded {constant, function, record pattern} resolution

Defunctorization

The Elaborate pass performs a number of duties historically assigned to the Defunctorize pass.

As part of the Elaborate pass, all module level constructs (`open`, `signature`, `structure`, `functor`, long identifiers) are removed. This works because the Elaborate pass assigns a unique name to every type and variable in the program. This also allows the Elaborate pass to eliminate `local` declarations, which are purely for namespace management.

Examples

Here are a number of examples of elaboration.

- All variables bound in `val` declarations are renamed.

```
val x = 13
val y = x
```

```
val x_0 = 13
val y_0 = x_0
```

- All variables in `fun` declarations are renamed.

```
fun f x = g x
and g y = f y
```

```
fun f_0 x_0 = g_0 x_0
and g_0 y_0 = f_0 y_0
```

- Type abbreviations are removed, and the abbreviation is expanded wherever it is used.

```
type 'a u = int * 'a
type 'b t = 'b u * real
fun f (x : bool t) = x
```

```
fun f_0 (x_0 : (int * bool) * real) = x_0
```

- Exception declarations create a new constructor and rename the type.

```
type t = int
exception E of t * real
```

```
exception E_0 of int * real
```

- The type and value constructors in datatype declarations are renamed.

```
datatype t = A of int | B of real * t
```

```
datatype t_0 = A_0 of int | B_0 of real * t_0
```

- Local declarations are moved to the top-level. The environment keeps track of the variables in scope.

```
val x = 13
local val x = 14
in val y = x
```

```
end
val z = x

val x_0 = 13
val x_1 = 14
val y_0 = x_1
val z_0 = x_0
```

- Structure declarations are eliminated, with all declarations moved to the top level. Long identifiers are renamed.

```
structure S =
   struct
      type t = int
      val x : t = 13
   end
val y : S.t = S.x

val x_0 : int = 13
val y_0 : int = x_0
```

- Open declarations are eliminated.

```
val x = 13
val y = 14
structure S =
   struct
     val x = 15
   end
open S
val z = x + y

val x_0 = 13
val y_0 = 14
val x_1 = 15
val z_0 = x_1 + y_0
```

- Functor declarations are eliminated, and the body of a functor is duplicated wherever the functor is applied.

```
functor F(val x : int) =
   struct
     val y = x
   end
structure F1 = F(val x = 13)
structure F2 = F(val x = 14)
val z = F1.y + F2.y

val x_0 = 13
val y_0 = x_0
val x_1 = 14
val y_1 = x_1
val z_0 = y_0 + y_1
```

- Signature constraints are eliminated. Note that signatures do affect how subsequent variables are renamed.

```
val y = 13
structure S : sig
                 val x : int
              end =
   struct
```

```
      val x = 14
      val y = x
   end
open S
val z = x + y
```

```
val y_0 = 13
val x_0 = 14
val y_1 = x_0
val z_0 = x_0 + y_0
```

Last edited on 2005-12-01 03:54:13 by StephenWeeks.

Emacs

SML Modes

There are a few Emacs modes for SML.

- `sml-mode`
    - http://www.xemacs.org/Documentation/packages/html/sml-mode_3.html
    - http://www.smlnj.org/doc/Emacs/sml-mode.html
- mlton.el contains the Emacs lisp that StephenWeeks uses to interact with MLton (in addition to using `sml-mode`).
- http://primate.net/~itz/mindent.tar, developed by Ian Zimmerman, who writes:

Unlike the widespread `sml-mode.el` it doesn't try to indent code based on ML syntax. I gradually got sceptical about this approach after writing the initial indentation support for caml mode and watching it bloat insanely as the language added new features. Also, any such attempts that I know of impose a particular coding style, or at best a choice among a limited set of styles, which I now oppose. Instead my mode is based on a generic package which provides manual bindable commands for common indentation operations (example: indent the current line under the n-th occurrence of a particular character in the previous non-blank line).

MLB modes

There is a mode for editing ML Basis files.

- esml-mlb-mode.el

Error messages

MLton's error messages are not in the format that the Emacs `next-error` parser natively understands. There are a couple of ways to fix this. The easiest way is to add the following to your `.emacs` to cause Emacs to recognize MLton's error messages.

```
(require 'compile)
(add-to-list 'compilation-error-regexp-alist
             '("^Error: \\([^\t\n]*\\) \\([0-9]+\\)\\.\\([0-9]+\\)\\.$"
               1 2 3))
```

Alternatively, you could use a sed script to rewrite MLton's errors. Here is one such script:

```
sed -e 's/^\([W|E].*\): \([^ ]*\) \([0-9][0-9]*\)\.\([0-9][0-9]*\)\./\2:\3:\1:\4/'
```

Last edited on 2005-12-01 03:57:27 by StephenWeeks.

Enscript

GNU Enscript converts ASCII files to PostScript, HTML, and other output languages, applying language sensitive highlighting (similar to Emacs's font lock mode). Here are a few *states* files for highlighting Standard ML.

- sml_simple.st -- Provides highlighting of keywords, string and character constants, and (nested) comments.

```
(* Comments (* can be nested *) *)
structure S = struct
  val x = (1, 2, "three")
end
```

- sml_verbose.st -- Supersedes the above, adding highlighting of numeric constants. Due to the limited parsing available, numeric record labels are highlighted as numeric constants, in all contexts. Likewise, a binding precedence separated from `infix` or `infixr` by a newline is highlighted as a numeric constant and a numeric record label selector separated from `#` by a newline is highlighted as a numeric constant.

```
structure S = struct
  (* These look good *)
  val x = (1, 2, "three")
  val z = #2 x

  (* Although these look bad (not all the numbers are constants),       *
   * they never occur in practice, as they are equivalent to the above. *)
  val x = {1 = 1, 3 = "three", 2 = 2}
  val z = #
            2 x
end
```

- sml_fancy.st -- Supersedes the above, adding highlighting of type and constructor bindings, highlighting of explicit binding of type variables at `val` and `fun` declarations, and separate highlighting of core and modules level keywords. Due to the limited parsing available, it is assumed that the input is a syntactically correct, top-level declaration.

```
structure S = struct
  val x = (1, 2, "three")
  datatype 'a t = T of 'a
       and u = U of v * v
  withtype v = {left: int t, right: int t}
  exception E1 of int and E2
  fun 'a id (x: 'a) : 'a = x

  (* Although this looks bad (the explicitly bound type variable 'a is *
   * not highlighted), it is unlikely to occur in practice.            *)
  val
      'a id = fn (x : 'a) => x
end
```

- sml_gaudy.st -- Supersedes the above, adding highlighting of type annotations, in both expressions and signatures. Due to the limited parsing available, it is assumed that the input is a syntactically correct, top-level declaration.

```
signature S = sig
  type t
  val x : t
```

```
  val f : t * int -> int
end
structure S : S = struct
  datatype t = T of int
  val x : t = T 0
  fun f (T x, i : int) : int = x + y
  fun 'a id (x: 'a) : 'a = x
end
```

Install and use

- Version 1.6.3 of GNU Enscript
  - Copy all files to `/usr/share/enscript/hl/` or `.enscript/` in your home directory.
  - Invoke `enscript` with `--highlight=sml_simple` (or `--highlight=sml_verbose` or `--highlight=sml_fancy` or `--highlight=sml_gaudy`).
- Version 1.6.1 of GNU Enscript
  - Append sml_all.st to `/usr/share/enscript/enscript.st`
  - Invoke `enscript` with `--pretty-print=sml_simple` (or `--pretty-print=sml_verbose` or `--pretty-print=sml_fancy` or `--pretty-print=sml_gaudy`).

This WebSite uses `sml_fancy` to pretty-print Standard ML source code. Comments and suggestions should be directed to MatthewFluet.

Last edited on 2005-12-02 03:28:59 by StephenWeeks.

EqualityType

An equality type is a type to which PolymorphicEquality can be applied. The Definition and the Basis Library precisely spell out which types are equality types.

- bool, char, IntInf.int, Int<N>.int, string, and Word<N>.word are equality types.
- for any t, both t array and t ref are equality types.
- if t is an equality type, then t list, and t vector are equality types.
- if t1, ..., tn are equality types, then t1 * ... * tn and {l1: t1, ..., ln: tn} are equality types.
- if t1, ..., tn are equality types and t AdmitsEquality, then (t1, ..., tn) t is an equality type.

To check that a type t is an equality type, use the following idiom.

```
structure S: sig eqtype t end =
   struct
      type t = ...
   end
```

Notably, exn and real are not equality types. Neither is t1 -> t2, for any t1 and t2.

Equality on arrays and ref cells is by identity, not structure. For example, ref 13 = ref 13 is false. On the other hand, equality for lists, strings, and vectors is by structure, not identity. For example, the following equalities hold.

```
[1, 2, 3] = 1 :: [2, 3]
"foo" = concat ["f", "o", "o"]
Vector.fromList [1, 2, 3] = Vector.tabulate (3, fn i => i + 1)
```

Last edited on 2005-12-02 01:19:02 by StephenWeeks.

EqualityTypeVariable

An equality type variable is a type variable that starts with two or more primes, as in `''a` or `''b`. The canonical use of equality type variables is in specifying the type of the PolymorphicEquality function, which is `''a * ''a -> bool`. Equality type variables ensure that polymorphic equality is only used on equality types, by requiring that at every use of a polymorphic value, equality type variables are instantiated by equality types.

For example, the following program is type correct because polymorphic equality is applied to variables of type `''a`.

```
fun f (x: ''a, y: ''a): bool = x = y
```

On the other hand, the following program is not type correct, because polymorphic equality is applied to variables of type `'a`, which is not an equality type.

```
fun f (x: 'a, y: 'a): bool = x = y
```

MLton reports the following error, indicating that polymorphic equality expects equality types, but didn't get them.

```
Error: z.sml 1.32.
  Function applied to incorrect argument.
    expects: [<equality>] * [<equality>]
    but got: [<non-equality>] * [<non-equality>]
    in: = (x, y)
```

As an example of using such a function that requires equality types, suppose that `f` has polymorphic type `''a -> unit`. Then, `f 13` is type correct because `int` is an equality type. On the other hand, `f 13.0` and `f (fn x => x)` are not type correct, because `real` and arrow types are not equality types. We can test these facts with the following short programs. First, we verify that such an `f` can be applied to integers.

```
functor Ok (val f: ''a -> unit): sig end =
   struct
      val () = f 13
      val () = f 14
   end
```

We can do better, and verify that such an `f` can be applied to any integer.

```
functor Ok (val f: ''a -> unit): sig end =
   struct
      fun g (x: int) = f x
   end
```

Even better, we don't need to introduce a dummy function name; we can use a type constraint.

```
functor Ok (val f: ''a -> unit): sig end =
   struct
      val _ = f: int -> unit
   end
```

Even better, we can use a signature constraint.

```
functor Ok (S: sig val f: ''a -> unit end):
   sig val f: int -> unit end = S
```

This functor concisely verifies that a function of polymorphic type `''a -> unit` can be safely used as a function of type `int -> unit`.

As above, we can verify that such an `f` can not be used at non equality types.

```
functor Bad (S: sig val f: ''a -> unit end):
   sig val f: real -> unit end = S
```

```
functor Bad (S: sig val f: ''a -> unit end):
   sig val f: ('a -> 'a) -> unit end = S
```

For each of these programs, MLton reports the following error.

```
Error: z.sml 2.4.
  Variable type in structure disagrees with signature.
    variable: f
    structure: [<equality>] -> _
    signature: [<non-equality>] -> _
```

Equality type variables in type and datatype declarations

Equality type variables can be used in type and datatype declarations; however they play no special role. For example,

```
type 'a t = 'a * int
```

is completely identical to

```
type ''a t = ''a * int
```

In particular, such a definition does *not* require that `t` only be applied to equality types.

Similarly,

```
datatype 'a t = A | B of 'a
```

is completely identical to

```
datatype ''a t = A | B of ''a
```

Last edited on 2005-12-01 04:00:38 by StephenWeeks.

EtaExpansion

Eta expansion is a simple syntactic change used to work around the ValueRestriction in Standard ML.

The eta expansion of an expression `e` is the expression `fn z => e z`, where `z` does not occur in `e`. This only makes sense if `e` denotes a function, i.e. is of arrow type. Eta expansion delays the evaluation of `e` until the function is applied, and will re-evaluate `e` each time the function is applied.

The name "eta expansion" comes from the eta-conversion rule of the lambda calculus. Expansion refers to the directionality of the equivalence being used, namely taking `e` to `fn z => e z` rather than `fn z => e z` to `e` (eta contraction).

Last edited on 2006-03-28 00:57:50 by StephenWeeks.

Experimental

This page is for experimental releases of MLton. These versions are not as well tested as our public releases, and may not be available for our all our usual platforms.

20060513 binary packages

This was built on AIX 5.1, should work on 5.2 and 5.3 as well.

- PowerPC
    - AIX 5.x

20051202 binary packages

This was built on HP-UX 11.00, should work on 11.11 and 11.23 as well.

- HPPA
    - HP-UX 11.x

Last edited on 2006-05-18 05:36:13 by VilleLaurikari.

FAQ

Feel free to ask questions and to update answers by editing this page. Since we try to make as much information as possible available on the web site and we like to avoid duplication, many of the answers are simply links to a web page that answers the question.

How do you pronounce MLton?

Pronounce

What SML software has been ported to MLton?

Libraries

What graphical libraries are available for MLton?

Libraries

How does MLton's performance compare to other SML compilers and to other languages?

MLton has excellent performance.

Does MLton treat monomorphic arrays and vectors specially?

MLton implements monomorphic arrays and vectors (e.g. `BoolArray`, `Word8Vector`) exactly as instantiations of their polymorphic counterpart (e.g. `bool array`, `Word8.word vector`). Thus, there is no need to use the monomorphic versions except when required to interface with the Basis Library or for portability with other SML implementations.

Why do I get a Segfault/Bus error in a program that uses IntInf/LargeInt to calculate numbers with several hundred thousand digits?

GnuMP

How can I decrease compile-time memory usage?

- Compile with `-verbose 3` to find out if the problem is due to an SSA optimization pass. If so, compile with `-drop-pass` *pass* to skip that pass.
- Compile with `@MLton hash-cons 0.5 --`, which will instruct the runtime to hash cons the heap every other GC.
- Compile with `-polyvariance false`, which is an undocumented option that causes less code duplication.

Also, please Contact us to let us know the problem to help us better understand MLton's limitations.

How do I see what has changed recently in the wiki?

RecentChanges

How portable is SML code across SML compilers?

StandardMLPortability

Last edited on 2005-12-02 01:19:12 by StephenWeeks.

Features

MLton has the following features.

Portability

- Runs on a variety of platforms.
    - hppa
        - Debian Linux
    - PowerPC
        - Darwin (Mac OS X)
        - Debian Linux
    - X86:
        - Cygwin/Windows
        - FreeBSD
        - Linux (Debian, Red Hat, ...)
        - MinGW/Windows
        - NetBSD
        - OpenBSD
    - Sparc
        - Debian Linux
        - Solaris

Robustness

- Supports the full SML 97 language as given in The Definition of Standard ML (Revised).
  If there is a program that is valid according to The Definition that is rejected by MLton, or a program that is invalid according to the Definition that is accepted by MLton, it is a bug. For a list of known bugs, see UnresolvedBugs.
- A complete implementation of the Basis Library.
  MLton's implementation matches latest Basis Library specification, and includes a complete implementation of all the required modules, as well as many of the optional modules.
- Generates standalone executables.
  No additional code or libraries are necessary in order to run an executable, except for the standard shared libraries. MLton can also generate statically linked executables.
- Compiles large programs.
  MLton is sufficiently efficient and robust that it can compile large programs, including itself (over 140K lines). The distributed version of MLton was compiled by MLton.
- Support for large amounts of memory (up to 4G).
- Array lengths up to 2^{31} - 1, the largest possible twos-complement 32 bit integer.
- Support for large files, using 64-bit file positions.

Performance

- Executables have excellent running times.
- Generates small executables.
  MLton takes advantage of whole-program compilation to perform very aggressive dead-code elimination, which often leads to smaller executables than with other SML compilers.
- Native integers, reals, and words.

In MLton, integers and words are 32 bits and arithmetic does not have any overhead due to tagging or boxing. Also, reals are stored unboxed, avoiding any overhead due to boxing.
- Unboxed native arrays.
  In MLton, an array (or vector) of integers, reals, or words uses the natural C-like representation. This is fast and supports easy exchange of data with C. Monomorphic arrays (and vectors) use the same C-like representations as their polymorphic counterparts.
- Multiple garbage collection strategies.
- Fast arbitrary precision arithmetic (`IntInf`) based on the GnuMP. For `IntInf` intensive programs, MLton can be an order of magnitude or more faster than Poly/ML or SML/NJ.

Tools

- Source-level Profiling of both time and allocation.
- MLLex lexer generator
- MLYacc parser generator
- ML-NLFFIGEN

Extensions

- A simple and fast C ForeignFunctionInterface that supports calling from SML to C and from C to SML.
- The ML Basis system for programming in the very large, separate delivery of library sources, and more.
- A number of extension libraries that provide useful functionality that cannot be implemented with the Basis Library. See below for an overview and MLtonStructure for details.
  - continuations
    MLton supports continuations via `callcc` and `throw`.
  - finalization
    MLton supports finalizable values of arbitrary type.
  - interval timers
    MLton supports the functionality of the C `setitimer` function.
  - random numbers
    MLton has functions similar to the C `rand` and `srand` functions, as well as support for access to `/dev/random` and `/dev/urandom`.
  - resource limits
    MLton has functions similar to the C `getrlimit` and `setrlimit` functions.
  - resource usage
    MLton supports a subset of the functionality of the C `getrusage` function.
  - signal handlers
    MLton supports signal handlers written in SML. Signal handlers run in a separate MLton thread, and have access to the thread that was interrupted by the signal. Signal handlers can be used in conjunction with threads to implement preemptive multitasking.
  - size primitive
    MLton includes a primitive that returns the size (in bytes) of any object. This can be useful in understanding the space behavior of a program.
  - system logging
    MLton has a complete interface to the C `syslog` function.
  - threads
    MLton has support for its own threads, upon which either preemptive or non-preemptive multitasking can be implemented. MLton also has support for Concurrent ML (CML).

- weak pointers
  MLton supports weak pointers, which allow the garbage collector to reclaim objects that it would otherwise be forced to keep. Weak pointers are also used to provide finalization.
- world save and restore

MLton has a facility for saving the entire state of a computation to a file and restarting it later. This facility can be used for staging and for checkpointing computations. It can even be used from within signal handlers, allowing interrupt driven checkpointing.

Last edited on 2006-07-20 19:37:02 by StephenWeeks.

FirstClassPolymorphism

First-class polymorphism is the ability to treat polymorphic functions just like other values: pass them as arguments, store them in data structures, etc. Although Standard ML does have polymorphic functions, it does not support first-class polymorphism.

For example, the following declares and uses the polymorphic function `id`.

```
val id = fn x => x
val _ = id 13
val _ = id "foo"
```

If SML supported first-class polymorphism, we could write the following.

```
fun useId id = (id 13; id "foo")
```

However, this does not type check. MLton reports the following error.

```
Error: z.sml 1.24.
  Function applied to incorrect argument.
    expects: [int]
    but got: [string]
    in: id "foo"
```

The error message arises because MLton infers from `id 13` that `id` accepts an integer argument, but that `id "foo"` is passing a string. Using explicit types sheds some light on the problem.

```
fun useId (id: 'a -> 'a) = (id 13; id "foo")
```

On this, MLton reports the following errors.

```
Error: z.sml 1.29.
  Function applied to incorrect argument.
    expects: ['a]
    but got: [int]
    in: id 13
Error: z.sml 1.36.
  Function applied to incorrect argument.
    expects: ['a]
    but got: [string]
    in: id "foo"
```

The errors arise because the argument `id` is *not* polymorphic; rather, it is monomorphic, with type `'a -> 'a`. It is perfectly valid to apply `id` to a value of type `'a`, as in the following

```
fun useId (id: 'a -> 'a, x: 'a) = id x  (* type correct *)
```

So, what is the difference between the type specification on `id` in the following two declarations?

```
val id: 'a -> 'a = fn x => x
fun useId (id: 'a -> 'a) = (id 13; id "foo")
```

While the type specifications on `id` look identical, they mean different things. The difference can be made clearer by explicitly scoping the type variables.

```
val 'a id: 'a -> 'a = fn x => x
fun 'a useId (id: 'a -> 'a) = (id 13; id "foo")  (* type error *)
```

In `val 'a id`, the type variable scoping means that for any `'a`, `id` has type `'a -> 'a`. Hence, `id` can be applied to arguments of type `int`, `real`, etc. Similarly, in `fun 'a useId`, the scoping means that `useId` is a polymorphic function that for any `'a` takes a function of type `'a -> 'a` and does something. Thus, `useId` could be applied to a function of type `int -> int`, `real -> real`, etc.

One could imagine an extension of SML that allowed scoping of type variables at places other than `fun` or `val` declarations, as in the following.

```
fun useId (id: ('a).'a -> 'a) = (id 13; id "foo")  (* not SML *)
```

Such an extension would need to be thought through very carefully, as it could cause significant complications with TypeInference, possible even undecidability.

Last edited on 2005-12-01 04:14:09 by StephenWeeks.

Fixpoints

In a strict language, such as Standard ML, you sometimes want to provide a fixpoint combinator for an abstract type `t` to make it possible to write recursive definitions. While it is easy to write an ad hoc fixpoint combinator for a single abstract type, it is more challenging to provide a general purpose framework for computing fixpoints. First of all, a single combinator `fix` with a type of the form `(t -> t) -> t` does not support mutual recursion over multiple values of type `t`. To support mutual recursion, you might provide a family of fixpoint combinators having types of the form `(u -> u) -> u` where `u` is a type of the form `t * ... * t`. Unfortunately, even such a family of fixpoint combinators does not support mutual recursion over multiple different abstract types. The gist of the problem is that we need a type-indexed family of fixpoint combinators. Below is a solution that allows for computing fixpoints over arbitrary products. The code on this page makes use of some Utilities.

First the signature of the fixpoint framework:

```
signature FIX =
  sig
    type 'a t1
    type 'a t2
    type 'a t = 'a t1 -> 'a t2

    exception Fix

    val fix : 'a t -> 'a fix
    val pure : ('a * 'a uop) thunk -> 'a t
    val tier : ('a * 'a effect) thunk -> 'a t
    val iso : ('a, 'b) emb -> 'b t -> 'a t
    val ^ : 'a t * 'b t -> ('a, 'b) product t
  end
```

`fix` is a type-indexed function. The type parameter to `fix` is called a "tier". To compute fixpoints over products, one uses the `^` operator to combine tiers. To provide a fixpoint combinator for an abstract type, one implements a tier providing a thunk whose instantation allocates a fresh "knot" and a procedure for "tying" it. Naturally this means that not all possible ways of computing a fixpoint of a particular type are possible under this framework. The `pure` combinator is a generalization of `tier`. The `iso` combinator is provided for reusing existing tiers.

Note that instead of using an infix operator, we could alternatively employ an interface using Fold. Also, the tiers are eta-expanded to work around the value restriction, while maintaining abstraction (the signature keeps tiers abstract).

Here is an implementation:

```
structure Fix :> FIX =
   struct
      type 'a t1 = unit
      type 'a t2 = 'a * 'a uop
      type 'a t = 'a t1 -> 'a t2

      exception Fix

      fun fix a f =
         let
            val (a, ta) = a ()
```

```
         in
            ta (f a)
         end

      val pure = id

      fun tier th () =
          let
             val (a, ta) = th ()
          in
             (a, const a o ta)
          end

      fun iso (a2b, b2a) b () =
         let
            val (b, tb) = b ()
         in
            (b2a b, b2a o tb o a2b)
         end

      fun (a ^ b) () =
         let
            val (a, ta) = a ()
            val (b, tb) = b ()
         in
            (a & b, fn a & b => ta a & tb b)
         end
   end
```

Let's then take a look at some examples. First a simple tier for functions:

```
structure Fn :>
   sig
      val Y : ('a -> 'b) Fix.t
   end = struct
      fun Y ? = Fix.tier (fn () =>
                             let
                                val r = ref (fail Fix.Fix)
                                fun f x = !r x
                             in
                                (f, r <\ op :=)
                             end) ?
   end
```

Here is an example of a mutually recursive definition of functions:

```
val isEven & isOdd =
    let open Fix Fn in fix (Y^Y) end
       (fn isEven & isOdd =>
           (fn 0w0 => true
             | 0w1 => false
             | n => isOdd (n-0w1)) &
           (fn 0w0 => false
             | 0w1 => true
             | n => isEven (n-0w1)))
```

Our second example is a naive implementation of lazy promises:

```
structure Promise :>
```

```
   sig
      type 'a t
      val lazy : 'a thunk -> 'a t
      val force : 'a t -> 'a
      val Y : 'a t Fix.t
   end = struct
      datatype 'a t' =
         Exn of exn
       | Thunk of 'a thunk
       | Value of 'a
      type 'a t = 'a t' ref
      fun lazy f = ref (Thunk f)
      fun force t =
         case !t of
            Exn e => raise e
          | Thunk f =>
            (t := Value (f ())
             handle e => t := Exn e
           ; force t)
          | Value v => v
      fun Y ? = Fix.tier (fn () =>
                             let
                                val r = lazy (fail Fix.Fix)
                             in
                                (r, r <\ op := o !)
                             end) ?
   end
```

An example use of our naive lazy promises is to implement equally naive lazy streams:

```
structure Stream :>
   sig
      type 'a t
      val cons : 'a * 'a t -> 'a t
      val get : 'a t -> ('a * 'a t) option
      val Y : 'a t Fix.t
   end = struct
      datatype 'a t = IN of ('a * 'a t) option Promise.t
      fun cons (x, xs) = IN (Promise.lazy (fn () => SOME (x, xs)))
      fun get (IN p) = Promise.force p
      fun Y ? = Fix.iso (fn IN p => p, IN) Promise.Y ?
   end
```

Note that above we make use of the `Fix.iso` combinator. Here is a finite representation of an infinite stream of ones:

```
val ones =
   let
      open Fix Stream
   in
      fix Y (fn ones => cons (1, ones))
   end
```

Last edited on 2006-08-30 09:45:25 by VesaKarvonen.

Flatten

Flatten is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This pass flattens arguments to SSA constructors, blocks, and functions.

If a tuple is explicitly available at all uses of a function (resp. block), then:

- The formals and call sites are changed so that the components of the tuple are passed.
- The tuple is reconstructed at the beginning of the body of the function (resp. block).

Similarly, if a tuple is explicitly available at all uses of a constructor, then:

- The constructor argument datatype is changed to flatten the tuple type.
- The tuple is passed flat at each `ConApp`.
- The tuple is reconstructed at each `Case` transfer target.

Implementation

flatten.sig flatten.fun

Details and Notes

Last edited on 2005-12-01 04:41:06 by MatthewFluet.

Fold

This page describes a technique that enables convenient syntax for a number of language features that are not explicitly supported by Standard ML, including: variable number of arguments, optional arguments and labeled arguments, array and vector literals, functional record update, and (seemingly) dependently typed functions like printf and scanf.

The key idea to *fold* is to define functions `fold`, `step0`, and `$` such that the following equation holds.

```
fold (a, f) (step0 h1) (step0 h2) ... (step0 hn) $
= f (hn (... (h2 (h1 a))))
```

The name `fold` comes because this is like a traditional list fold, where `a` is the *base element*, and each *step function*, `step0 hi`, corresponds to one element of the list and does one step of the fold. The name `$` is chosen to mean *end of arguments* from its common use in regular-expression syntax.

Unlike the usual list fold in which the same function is used to step over each element in the list, this fold allows the step functions to be different from each other, and even to be of different types. Also unlike the usual list fold, this fold includes a *finishing function*, `f`, that is applied to the result of the fold. The presence of the finishing function may seem odd because there is no analogy in list fold. However, the finishing function is essential; without it, there would be no way for the folder to perform an arbitrary computation after processing all the arguments. The examples below will make this clear.

The functions `fold`, `step0`, and `$` are easy to define.

```
fun $ (a, f) = f a
fun id x = x
structure Fold =
   struct
      fun fold (a, f) g = g (a, f)
      fun step0 h (a, f) = fold (h a, f)
   end
```

We've placed `fold` and `step0` in the `Fold` structure but left `$` at the toplevel because it is convenient in code to always have `$` in scope. We've also defined the identity function, `id`, at the toplevel since we use it so frequently.

Plugging in the definitions, it is easy to verify the equation from above.

```
fold (a, f) (step0 h1) (step0 h2) ... (step0 hn) $
= step0 h1 (a, f) (step0 h2) ... (step0 hn) $
= fold (h1 a, f) (step0 h2) ... (step0 hn) $
= step0 h2 (h1 a, f) ... (step0 hn) $
= fold (h2 (h1 a), f) ... (step0 hn) $
...
= fold (hn (... (h2 (h1 a))), f) $
= $ (hn (... (h2 (h1 a))), f)
= f (hn (... (h2 (h1 a))))
```

Example: variable number of arguments

The simplest example of fold is accepting a variable number of (curried) arguments. We'll define a function `f` and argument `a` such that all of the following expressions are valid.

```
f $
f a $
f a a $
f a a a $
f a a a ... a a a $ (* as many a's as we want *)
```

Off-hand it may appear impossible that all of the above expressions are type correct SML -- how can a function `f` accept a variable number of curried arguments? What could the type of `f` be? We'll have more to say later on how type checking works. For now, once we have supplied the definitions below, you can check that the expressions are type correct by feeding them to your favorite SML implementation.

It is simple to define `f` and `a`. We define `f` as a folder whose base element is `()` and whose finish function does nothing. We define `a` as the step function that does nothing. The only trickiness is that we must eta expand the definition of `f` and `a` to work around the ValueRestriction; we frequently use eta expansion for this purpose without mention.

```
val base = ()
fun finish () = ()
fun step () = ()
val f = fn z => Fold.fold (base, finish) z
val a = fn z => Fold.step0 step z
```

One can easily apply the fold equation to verify by hand that `f` applied to any number of `a`'s evaluates to `()`.

```
f a ... a $
= finish (step (... (step base)))
= finish (step (... ()))
...
= finish ()
= ()
```

Example: variable-argument sum

Let's look at an example that computes something: a variable-argument function `sum` and a stepper `a` such that

```
sum (a i1) (a i2) ... (a im) $ = i1 + i2 + ... + im
```

The idea is simple -- the folder starts with a base accumulator of `0` and the stepper adds each element to the accumulator, `s`, which the folder simply returns at the end.

```
val sum = fn z => Fold.fold (0, fn s => s) z
fun a i = Fold.step0 (fn s => i + s)
```

Using the fold equation, one can verify the following.

```
sum (a 1) (a 2) (a 3) $ = 6
```

Step1

It is sometimes syntactically convenient to omit the parentheses around the steps in a fold. This is easily done by defining a new function, `step1`, as follows.

```
structure Fold =
   struct
      open Fold
      fun step1 h (a, f) b = fold (h (b, a), f)
   end
```

From the definition of `step1`, we have the following equivalence.

```
fold (a, f) (step1 h) b
= step1 h (a, f) b
= fold (h (b, a), f)
```

Using the above equivalence, we can compute the following equation for `step1`.

```
fold (a, f) (step1 h1) b1 (step1 h2) b2 ... (step1 hn) bn $
= fold (h1 (b1, a), f) (step1 h2) b2 ... (step1 hn) bn $
= fold (h2 (b2, h1 (b1, a)), f) ... (step1 hn) bn $
= fold (hn (bn, ... (h2 (b2, h1 (b1, a)))), f) $
= f (hn (bn, ... (h2 (b2, h1 (b1, a)))))
```

Here is an example using `step1` to define a variable-argument product function, `prod`, with a convenient syntax.

```
val prod = fn z => Fold.fold (1, fn p => p) z
val ` = fn z => Fold.step1 (fn (i, p) => i * p) z
```

The functions `prod` and ` satisfy the following equation.

```
prod `i1 `i2 ... `im $ = i1 * i2 * ... * im
```

Note that in SML, `` `i1 `` is two different tokens, ` and `i1`. We often use ` for an instance of a `step1` function because of its syntactic unobtrusiveness and because no space is required to separate it from an alphanumeric token.

Also note that there are no parenthesis around the steps. That is, the following expression is not the same as the above one (in fact, it is not type correct).

```
prod (`i1) (`i2) ... (`im) $
```

Example: list literals

SML already has a syntax for list literals, e.g. `[w, x, y, z]`. However, using fold, we can define our own syntax.

```
val list = fn z => Fold.fold ([], rev) z
val ` = fn z => Fold.step1 (op ::) z
```

The idea is that the folder starts out with the empty list, the steps accumulate the elements into a list, and then the finishing function reverses the list at the end.

With these definitions one can write a list like:

```
list `w `x `y `z $
```

While the example is not practically useful, it does demonstrate the need for the finishing function to be incorporated in `fold`. Without a finishing function, every use of `list` would need to be wrapped in `rev`, as follows.

```
rev (list `w `x `y `z $)
```

The finishing function allows us to incorporate the reversal into the definition of `list`, and to treat `list` as a truly variable argument function, performing an arbitrary computation after receiving all of its arguments.

See ArrayLiteral for a similar use of `fold` that provides a syntax for array and vector literals, which are not built in to SML.

Fold right

Just as `fold` is analogous to a fold left, in which the functions are applied to the accumulator left-to-right, we can define a variant of `fold` that is analogous to a fold right, in which the functions are applied to the accumulator right-to-left. That is, we can define functions `foldr` and `step0` such that the following equation holds.

```
foldr (a, f) (step0 h1) (step0 h2) ... (step0 hn) $
= f (h1 (h2 (... (hn a))))
```

The implementation of fold right is easy, using fold. The idea is for the fold to start with `f` and for each step to precompose the next `hi`. Then, the finisher applies the composed function to the base value, `a`. Here is the code.

```
structure Foldr =
   struct
      fun foldr (a, f) = Fold.fold (f, fn g => g a)
      fun step0 h = Fold.step0 (fn g => g o h)
   end
```

Verifying the fold-right equation is straightforward, using the fold-left equation.

```
foldr (a, f) (Foldr.step0 h1) (Foldr.step0 h2) ... (Foldr.step0 hn) $
= fold (f, fn g => g a)
    (Fold.step0 (fn g => g o h1))
    (Fold.step0 (fn g => g o h2))
    ...
    (Fold.step0 (fn g => g o hn)) $
= (fn g => g a)
  ((fn g => g o hn) (... ((fn g => g o h2) ((fn g => g o h1) f))))
= (fn g => g a)
  ((fn g => g o hn) (... ((fn g => g o h2) (f o h1))))
= (fn g => g a) ((fn g => g o hn) (... (f o h1 o h2)))
= (fn g => g a) (f o h1 o h2 o ... o hn)
= (f o h1 o h2 o ... o hn) a
= f (h1 (h2 (... (hn a))))
```

One can also define the fold-right analogue of `step1`.

```
structure Foldr =
   struct
      open Foldr
      fun step1 h = Fold.step1 (fn (b, g) => g o (fn a => h (b, a)))
```

```
end
```

Example: list literals via fold right

Revisiting the list literal example from earlier, we can use fold right to define a syntax for list literals that doesn't do a reversal.

```
val list = fn z => Foldr.foldr ([], fn l => l) z
val ` = fn z => Foldr.step1 (op ::) z
```

As before, with these definitions, one can write a list like:

```
list `w `x `y `z $
```

The difference between the fold-left and fold-right approaches is that the fold-right approach does not have to reverse the list at the end, since it accumulates the elements in the correct order. In practice, MLton will simplify away all of the intermediate function composition, so the the fold-right approach will be more efficient.

Mixing steppers

All of the examples so far have used the same step function throughout a fold. This need not be the case. For example, consider the following.

```
val n = fn z => Fold.fold (0, fn i => i) z
val I = fn z => Fold.step0 (fn i => i * 2) z
val O = fn z => Fold.step0 (fn i => i * 2 + 1) z
```

Here we have one folder, `n`, that can be used with two different steppers, `I` and `O`. By using the fold equation, one can verify the following equations.

```
n O $ = 0
n I $ = 1
n I O $ = 2
n I O I $ = 5
n I I I O $ = 14
```

That is, we've defined a syntax for writing binary integer constants.

Not only can one use different instances of `step0` in the same fold, one can also intermix uses of `step0` and `step1`. For example, consider the following.

```
val n = fn z => Fold.fold (0, fn i => i) z
val O = fn z => Fold.step0 (fn i => n * 8) z
val ` = fn z => Fold.step1 (fn (i, n) => n * 8 + i) z
```

Using the straightforward generalization of the fold equation to mixed steppers, one can verify the following equations.

```
n 0 $ = 0
n `3 O $ = 24
n `1 O `7 $ = 71
```

That is, we've defined a syntax for writing octal integer constants, with a special syntax, `O`, for the zero digit (admittedly contrived, since one could just write `` ` 0`` instead of `O`).

See NumericLiteral for a practical extension of this approach that supports numeric constants in any base and of any type.

(Seemingly) dependent types

A normal list fold always returns the same type no matter what elements are in the list or how long the list is. Variable-argument fold is more powerful, because the result type can vary based both on the arguments that are passed and on their number. This can provide the illusion of dependent types.

For example, consider the following.

```
val f = fn z => Fold.fold ((), id) z
val a = fn z => Fold.step0 (fn () => "hello") z
val b = fn z => Fold.step0 (fn () => 13) z
val c = fn z => Fold.step0 (fn () => (1, 2)) z
```

Using the fold equation, one can verify the following equations.

```
f a $ = "hello": string
f b $ = 13: int
f c $ = (1, 2): int * int
```

That is, `f` returns a value of a different type depending on whether it is applied to argument `a`, argument `b`, or argument `c`.

The following example shows how the type of a fold can depend on the number of arguments.

```
val grow = fn z => Fold.fold ([], fn l => l) z
val a = fn z => Fold.step0 (fn x => [x]) z
```

Using the fold equation, one can verify the following equations.

```
grow $ = []: 'a list
grow a $ = [[]]: 'a list list
grow a a $ = [[[]]]: 'a list list list
```

Clearly, the result type of a call to the variable argument `grow` function depends on the number of arguments that are passed.

As a reminder, this is well-typed SML. You can check it out in any implementation.

(Seemingly) dependently-typed functional results

Fold is especially useful when it returns a curried function whose arity depends on the number of arguments. For example, consider the following.

```
val makeSum = fn z => Fold.fold (id, fn f => f 0) z
val I = fn z => Fold.step0 (fn f => fn i => fn x => f (x + i)) z
```

The `makeSum` folder constructs a function whose arity depends on the number of `I` arguments and that adds together all of its arguments. For example, `makeSum I $` is of type `int -> int` and `makeSum I I $` is of type `int -> int -> int`.

One can use the fold equation to verify that the `makeSum` works correctly. For example, one can easily check by hand the following equations.

```
makeSum I $ 1 = 1
makeSum I I $ 1 2 = 3
makeSum I I I $ 1 2 3 = 6
```

Returning a function becomes especially interesting when there are steppers of different types. For example, the following `makeSum` folder constructs functions that sum integers and reals.

```
val makeSum = fn z => Foldr.foldr (id, fn f => f 0.0) z
val I = fn z => Foldr.step0 (fn f => fn x => fn i => f (x + real i)) z
val R = fn z => Foldr.step0 (fn f => fn x: real => fn r => f (x + r)) z
```

With these definitions, `makeSum I R $` is of type `int -> real -> real` and `makeSum R I I $` is of type `real -> int -> int -> real`. One can use the foldr equation to check the following equations.

```
makeSum I $ 1 = 1.0
makeSum I R $ 1 2.5 = 3.5
makeSum R I I $ 1.5 2 3 = 6.5
```

We used `foldr` instead of `fold` for this so that the order in which the specifiers `I` and `R` appear is the same as the order in which the arguments appear. Had we used `fold`, things would have been reversed.

An extension of this idea is sufficient to define Printf-like functions in SML.

An idiom for combining steps

It is sometimes useful to combine a number of steps together and name them as a single step. As a simple example, suppose that one often sees an integer follower by a real in the `makeSum` example above. One can define a new *compound step* `IR` as follows.

```
val IR = fn u => Fold.fold u I R
```

With this definition in place, one can verify the following.

```
makeSum IR IR $ 1 2.2 3 4.4 = 10.6
```

In general, one can combine steps `s1`, `s2`, ... `sn` as

```
fn u => Fold.fold u s1 s2 ... sn
```

The following calculation shows why a compound step behaves as the composition of its constituent steps.

```
fold u (fn u => fold u s1 s2 ... sn)
= (fn u => fold u s1 s2 ... sn) u
= fold u s1 s2 ... sn
```

Post composition

Suppose we already have a function defined via fold, `w = fold (a, f)`, and we would like to construct a new fold function that is like `w`, but applies `g` to the result produced by `w`. This is similar to function composition, but we can't just do `g o w`, because we don't want to use `g` until `w` has been applied to all of its arguments and received the end-of-arguments terminator `$`.

More precisely, we want to define a post-composition function `post` that satisfies the following equation.

```
post (w, g) s1 ... sn $ = g (w s1 ... sn $)
```

Here is the definition of `post`.

```
structure Fold =
   struct
      open Fold
      fun post (w, g) s = w (fn (a, h) => s (a, g o h))
   end
```

The following calculations show that `post` satisfies the desired equation, where `w = fold (a, f)`.

```
post (w, g) s
= w (fn (a, h) => s (a, g o h))
= fold (a, f) (fn (a, h) => s (a, g o h))
= (fn (a, h) => s (a, g o h)) (a, f)
= s (a, g o f)
= fold (a, g o f) s
```

Now, suppose `si = step0 hi` for `i` from `1` to `n`.

```
post (w, g) s1 s2 ... sn $
= fold (a, g o f) s1 s2 ... sn $
= (g o f) (hn (... (h1 a)))
= g (f (hn (... (h1 a))))
= g (fold (a, f) s1 ... sn $)
= g (w s1 ... sn $)
```

For a practical example of post composition, see ArrayLiteral.

Lift

We now define a peculiar-looking function, `lift0`, that is, equationally speaking, equivalent to the identity function on a step function.

```
fun lift0 s (a, f) = fold (fold (a, id) s $, f)
```

Using the definitions, we can prove the following equation.

```
fold (a, f) (lift0 (step0 h)) = fold (a, f) (step0 h)
```

Here is the proof.

```
fold (a, f) (lift0 (step0 h))
= lift0 (step0 h) (a, f)
```

```
= fold (fold (a, id) (step0 h) $, f)
= fold (step0 h (a, id) $, f)
= fold (fold (h a, id) $, f)
= fold ($ (h a, id), f)
= fold (id (h a), f)
= fold (h a, f)
= step0 h (a, f)
= fold (a, f) (step0 h)
```

If `lift0` is the identity, then why even define it? The answer lies in the typing of fold expressions, which we have, until now, left unexplained.

Typing

Perhaps the most surprising aspect of fold is that it can be checked by the SML type system. The types involved in fold expressions are complex; fortunately type inference is able to deduce them. Nevertheless, it is instructive to study the types of fold functions and steppers. More importantly, it is essential to understand the typing aspects of fold in order to write down signatures of functions defined using fold and step.

Here is the `FOLD` signature, and a recaptulation of the entire `Fold` structure, with additional type annotations.

```
signature FOLD =
   sig
      type ('a, 'b, 'c, 'd) step = 'a * ('b -> 'c) -> 'd
      type ('a, 'b, 'c, 'd) t = ('a, 'b, 'c, 'd) step -> 'd
      type ('a1, 'a2, 'b, 'c, 'd) step0 =
         ('a1, 'b, 'c, ('a2, 'b, 'c, 'd) t) step
      type ('a11, 'a12, 'a2, 'b, 'c, 'd) step1 =
         ('a12, 'b, 'c, 'a11 -> ('a2, 'b, 'c, 'd) t) step

      val fold: 'a * ('b -> 'c) -> ('a, 'b, 'c, 'd) t
      val lift0: ('a1, 'a2, 'a2, 'a2, 'a2) step0
                 -> ('a1, 'a2, 'b, 'c, 'd) step0
      val post: ('a, 'b, 'c1, 'd) t * ('c1 -> 'c2)
                -> ('a, 'b, 'c2, 'd) t
      val step0: ('a1 -> 'a2) -> ('a1, 'a2, 'b, 'c, 'd) step0
      val step1: ('a11 * 'a12 -> 'a2)
                 -> ('a11, 'a12, 'a2, 'b, 'c, 'd) step1
   end

structure Fold:> FOLD =
   struct
      type ('a, 'b, 'c, 'd) step = 'a * ('b -> 'c) -> 'd

      type ('a, 'b, 'c, 'd) t = ('a, 'b, 'c, 'd) step -> 'd

      type ('a1, 'a2, 'b, 'c, 'd) step0 =
         ('a1, 'b, 'c, ('a2, 'b, 'c, 'd) t) step

      type ('a11, 'a12, 'a2, 'b, 'c, 'd) step1 =
         ('a12, 'b, 'c, 'a11 -> ('a2, 'b, 'c, 'd) t) step

      fun fold (a: 'a, f: 'b -> 'c)
               (g: ('a, 'b, 'c, 'd) step): 'd =
         g (a, f)

      fun step0 (h: 'a1 -> 'a2)
                (a1: 'a1, f: 'b -> 'c): ('a2, 'b, 'c, 'd) t =
```

```
         fold (h a1, f)

      fun step1 (h: 'a11 * 'a12 -> 'a2)
                (a12: 'a12, f: 'b -> 'c)
                (a11: 'a11): ('a2, 'b, 'c, 'd) t =
         fold (h (a11, a12), f)

      fun lift0 (s: ('a1, 'a2, 'a2, 'a2, 'a2) step0)
                (a: 'a1, f: 'b -> 'c): ('a2, 'b, 'c, 'd) t =
         fold (fold (a, id) s $, f)

      fun post (w: ('a, 'b, 'c1, 'd) t,
                g: 'c1 -> 'c2)
               (s: ('a, 'b, 'c2, 'd) step): 'd =
         w (fn (a, h) => s (a, g o h))
   end
```

That's a lot to swallow, so let's walk through it one step at a time. First, we have the definition of type `Fold.step`.

```
type ('a, 'b, 'c, 'd) step = 'a * ('b -> 'c) -> 'd
```

As a fold proceeds over its arguments, it maintains two things: the accumulator, of type `'a`, and the finishing function, of type `'b -> 'c`. Each step in the fold is a function that takes those two pieces (i.e. `'a * ('b -> 'c)` and does something to them (i.e. produces `'d`). The result type of the step is completely left open to be filled in by type inference, as it is an arrow type that is capable of consuming the rest of the arguments to the fold.

A folder, of type `Fold.t`, is a function that consumes a single step.

```
type ('a, 'b, 'c, 'd) t = ('a, 'b, 'c, 'd) step -> 'd
```

Expanding out the type, we have:

```
type ('a, 'b, 'c, 'd) t = ('a * ('b -> 'c) -> 'd) -> 'd
```

This shows that the only thing a folder does is to hand its accumulator (`'a`) and finisher (`'b -> 'c`) to the next step (`'a * ('b -> 'c) -> 'd`). If SML had first-class polymorphism, we would write the fold type as follows.

```
type ('a, 'b, 'c) t = Forall 'd. ('a, 'b, 'c, 'd) step -> 'd
```

This type definition shows that a folder had nothing to do with the rest of the fold, it only deals with the next step.

We now can understand the type of `fold`, which takes the initial value of the accumulator and the finishing function, and constructs a folder, i.e. a function awaiting the next step.

```
val fold: 'a * ('b -> 'c) -> ('a, 'b, 'c, 'd) t
fun fold (a: 'a, f: 'b -> 'c)
         (g: ('a, 'b, 'c, 'd) step): 'd =
   g (a, f)
```

Continuing on, we have the type of step functions.

```
type ('a1, 'a2, 'b, 'c, 'd) step0 =
   ('a1, 'b, 'c, ('a2, 'b, 'c, 'd) t) step
```

Expanding out the type a bit gives:

```
type ('a1, 'a2, 'b, 'c, 'd) step0 =
   'a1 * ('b -> 'c) -> ('a2, 'b, 'c, 'd) t
```

So, a step function takes the accumulator (`'a1`) and finishing function (`'b -> 'c`), which will be passed to it by the previous folder, and transforms them to a new folder. This new folder has a new accumulator (`'a2`) and the same finishing function.

Again, imagining that SML had first-class polymorphism makes the type clearer.

```
type ('a1, 'a2) step0 =
   Forall ('b, 'c). ('a1, 'b, 'c, ('a2, 'b, 'c) t) step
```

Thus, in essence, a `step0` function is a wrapper around a function of type `'a1 -> 'a2`, which is exactly what the definition of `step0` does.

```
val step0: ('a1 -> 'a2) -> ('a1, 'a2, 'b, 'c, 'd) step0
fun step0 (h: 'a1 -> 'a2)
          (a1: 'a1, f: 'b -> 'c): ('a2, 'b, 'c, 'd) t =
   fold (h a1, f)
```

It is not much beyond `step0` to understand `step1`.

```
type ('a11, 'a12, 'a2, 'b, 'c, 'd) step1 =
   ('a12, 'b, 'c, 'a11 -> ('a2, 'b, 'c, 'd) t) step
```

A `step1` function takes the accumulator (`'a12`) and finisher (`'b -> 'c`) passed to it by the previous folder and transforms them into a function that consumes the next argument (`'a11`) and produces a folder that will continue the fold with a new accumulator (`'a2`) and the same finisher.

```
fun step1 (h: 'a11 * 'a12 -> 'a2)
          (a12: 'a12, f: 'b -> 'c)
          (a11: 'a11): ('a2, 'b, 'c, 'd) t =
   fold (h (a11, a12), f)
```

With first-class polymorphism, a `step1` function is more clearly seen as a wrapper around a binary function of type `'a11 * 'a12 -> 'a2`.

```
type ('a11, 'a12, 'a2) step1 =
   Forall ('b, 'c). ('a12, 'b, 'c, 'a11 -> ('a2, 'b, 'c) t) step
```

The type of `post` is clear: it takes a folder with a finishing function that produces type `'c1`, and a function of type `'c1 -> 'c2` to postcompose onto the folder. It returns a new folder with a finishing function that produces type `'c2`.

```
val post: ('a, 'b, 'c1, 'd) t * ('c1 -> 'c2)
          -> ('a, 'b, 'c2, 'd) t
fun post (w: ('a, 'b, 'c1, 'd) t,
          g: 'c1 -> 'c2)
         (s: ('a, 'b, 'c2, 'd) step): 'd =
   w (fn (a, h) => s (a, g o h))
```

We will return to `lift0` after an example.

An example typing

Let's type check our simplest example, a variable-argument fold. Recall that we have a folder `f` and a stepper `a` defined as follows.

```
val f = fn z => Fold.fold ((), fn () => ()) z
val a = fn z => Fold.step0 (fn () => ()) z
```

Since the accumulator and finisher are uninteresting, we'll use some abbreviations to simplify things.

```
type 'd step = (unit, unit, unit, 'd) Fold.step
type 'd fold = 'd step -> 'd
```

With these abbreviations, `f` and `a` have the following polymorphic types.

```
f: 'd fold
a: 'd step
```

Suppose we want to type check

```
f a a a $: unit
```

As a reminder, the fully parenthesized expression is

```
((((f a) a) a) a) $
```

The observation that we will use repeatedly is that for any type `z`, if `f: z fold` and `s: z step`, then `f s: z`. So, if we want

```
(f a a a) $: unit
```

then we must have

```
f a a a: unit fold
$: unit step
```

Applying the observation again, we must have

```
f a a: unit fold fold
a: unit fold step
```

Applying the observation two more times leads to the following type derivation.

```
f: unit fold fold fold fold  a: unit fold fold fold step
f a: unit fold fold fold     a: unit fold fold step
f a a: unit fold fold        a: unit fold step
f a a a: unit fold           $: unit step
f a a a $: unit
```

So, each application is a fold that consumes the next step, producing a fold of one smaller type.

One can expand some of the type definitions in f to see that it is indeed a function that takes four curried arguments, each one a step function.

```
f: unit fold fold fold step
   -> unit fold fold step
   -> unit fold step
   -> unit step
   -> unit
```

This example shows why we must eta expand uses of fold and step0 to work around the value restriction and make folders and steppers polymorphic. The type of a fold function like f depends on the number of arguments, and so will vary from use to use. Similarly, each occurrence of an argument like a has a different type, depending on the number of remaining arguments.

This example also shows that the type of a folder, when fully expanded, is exponential in the number of arguments: there are as many nested occurrences of the fold type constructor as there are arguments, and each occurence duplicates its type argument. One can observe this exponential behavior in a type checker that doesn't share enough of the representation of types (e.g. one that represents types as trees rather than directed acyclic graphs).

Generalizing this type derivation to uses of fold where the accumulator and finisher are more interesting is straightforward. One simply includes the type of the accumulator, which may change, for each step, and the type of the finisher, which doesn't change from step to step.

Typing lift

The lack of first-class polymorphism in SML causes problems if one wants to use a step in a first-class way. Consider the following double function, which takes a step, s, and produces a composite step that does s twice.

```
fun double s = fn u => Fold.fold u s s
```

The definition of double is not type correct. The problem is that the type of a step depends on the number of remaining arguments but that the parameter s is not polymorphic, and so can not be used in two different positions.

Fortunately, we can define a function, lift0, that takes a monotyped step function and *lifts* it into a polymorphic step function. This is apparent in the type of lift0.

```
val lift0: ('a1, 'a2, 'a2, 'a2, 'a2) step0
           -> ('a1, 'a2, 'b, 'c, 'd) step0
fun lift0 (s: ('a1, 'a2, 'a2, 'a2, 'a2) step0)
          (a: 'a1, f: 'b -> 'c): ('a2, 'b, 'c, 'd) t =
   fold (fold (a, id) s $, f)
```

The following definition of double uses lift0, appropriately eta wrapped, to fix the problem.

```
fun double s =
   let
      val s = fn z => Fold.lift0 s z
   in
      fn u => Fold.fold u s s
   end
```

With that definition of `double` in place, we can use it as in the following example.

```
val f = fn z => Fold.fold ((), fn () => ()) z
val a = fn z => Fold.step0 (fn () => ()) z
val a2 = fn z => double a z
val () = f a a2 a a2 $
```

Of course, we must eta wrap the call `double` in order to use its result, which is a step function, polymorphically.

Hiding the type of the accumulator

For clarity and to avoid mistakes, it can be useful to hide the type of the accumulator in a fold. Reworking the simple variable-argument example to do this leads to the following.

```
structure S:>
  sig
     type ac
     val f: (ac, ac, unit, 'd) Fold.t
     val s: (ac, ac, 'b, 'c, 'd) Fold.step0
  end =
  struct
     type ac = unit
     val f = fn z => Fold.fold ((), fn () => ()) z
     val s = fn z => Fold.step0 (fn () => ()) z
  end
```

The idea is to name the accumulator type and use opaque signature matching to make it abstract. This can prevent improper manipulation of the accumulator by client code and ensure invariants that the folder and stepper would like to maintain.

For a practical example of this technique, see ArrayLiteral.

Also see

Fold has a number of practical applications. Here are some of them.

- ArrayLiteral
- Fold01N
- FunctionalRecordUpdate
- NumericLiteral
- OptionalArguments
- Printf
- VariableArityPolymorphism

Last edited on 2006-03-28 00:58:28 by StephenWeeks.

Fold01N

A common use pattern of Fold is to define a variable-arity function that combines multiple arguments together using a binary function. It is slightly tricky to do this directly using fold, because of the special treatment required for the case of zero or one argument. Here is a structure, `Fold01N`, that solves the problem once and for all, and eases the definition of such functions.

```
structure Fold01N =
   struct
      fun fold {finish, start, zero} =
         Fold.fold ((id, finish, fn () => zero, start),
                    fn (finish, _, p, _) => finish (p ()))

      fun step0 {combine, input} =
         Fold.step0 (fn (_, finish, _, f) =>
                      (finish,
                       finish,
                       fn () => f input,
                       fn x' => combine (f input, x')))

      fun step1 {combine} z input =
         step0 {combine = combine, input = input} z
   end
```

If one has a value `zero`, and functions `start`, `c`, and `finish`, then one can define a variable-arity function `f` and stepper ` as follows.

```
val f = fn z => Fold01N.fold {finish = finish, start = start, zero = zero} z
val ` = fn z => Fold01N.step1 {combine = c} z
```

One can then use the fold equation to prove the following equations.

```
f $ = zero
f `a1 $ = finish (start a1)
f `a1 `a2 $ = finish (c (start a1, a2))
f `a1 `a2 `a3 $ = finish (c (c (start a1, a2), a3))
...
```

For an example of `Fold01N`, see VariableArityPolymorphism.

Typing Fold01N

Here is the signature for `Fold01N`. We use a trick to avoid having to duplicate the definition of some rather complex types in both the signature and the structure. We first define the types in a structure. Then, we define them via type re-definitions in the signature, and via `open` in the full structure.

```
structure Fold01N =
   struct
      type ('input, 'accum1, 'accum2, 'answer, 'zero,
            'a, 'b, 'c, 'd, 'e) t =
         (('zero -> 'zero)
          * ('accum2 -> 'answer)
          * (unit -> 'zero)
          * ('input -> 'accum1),
          ('a -> 'b) * 'c * (unit -> 'a) * 'd,
```

```
         'b,
         'e) Fold.t

      type ('input1, 'accum1, 'input2, 'accum2,
           'a, 'b, 'c, 'd, 'e, 'f) step0 =
         ('a * 'b * 'c * ('input1 -> 'accum1),
          'b * 'b * (unit -> 'accum1) * ('input2 -> 'accum2),
          'd, 'e, 'f) Fold.step0

      type ('accum1, 'input, 'accum2,
           'a, 'b, 'c, 'd, 'e, 'f, 'g) step1 =
         ('a,
          'b * 'c * 'd * ('a -> 'accum1),
          'c * 'c * (unit -> 'accum1) * ('input -> 'accum2),
          'e, 'f, 'g) Fold.step1
   end

signature FOLD_01N =
   sig
      type ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) t =
         ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) Fold01N.t
      type ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) step0 =
         ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) Fold01N.step0
      type ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) step1 =
         ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) Fold01N.step1

      val fold:
         {finish: 'accum2 -> 'answer,
          start: 'input -> 'accum1,
          zero: 'zero}
         -> ('input, 'accum1, 'accum2, 'answer, 'zero,
             'a, 'b, 'c, 'd, 'e) t

      val step0:
         {combine: 'accum1 * 'input2 -> 'accum2,
          input: 'input1}
         -> ('input1, 'accum1, 'input2, 'accum2,
             'a, 'b, 'c, 'd, 'e, 'f) step0

      val step1:
         {combine: 'accum1 * 'input -> 'accum2}
         -> ('accum1, 'input, 'accum2,
             'a, 'b, 'c, 'd, 'e, 'f, 'g) step1
   end

structure Fold01N: FOLD_01N =
   struct
      open Fold01N

      fun fold {finish, start, zero} =
         Fold.fold ((id, finish, fn () => zero, start),
                    fn (finish, _, p, _) => finish (p ()))

      fun step0 {combine, input} =
         Fold.step0 (fn (_, finish, _, f) =>
                     (finish,
                      finish,
                      fn () => f input,
                      fn x' => combine (f input, x')))

      fun step1 {combine} z input =
```

```
        step0 {combine = combine, input = input} z
    end
```

Last edited on 2006-03-21 23:10:39 by VesaKarvonen.

ForLoops

A `for`-loop is typically used to iterate over a range of consecutive integers that denote indices of some sort. For example, in OCaml a `for`-loop takes either the form

```
for <name> = <lower> to <upper> do <body> done
```

or the form

```
for <name> = <upper> downto <lower> do <body> done
```

Some languages provide considerably more flexible `for`-loop or `foreach`-constructs.

A bit surprisingly, Standard ML provides special syntax for `while`-loops, but not for `for`-loops. Indeed, in SML, many uses of `for`-loops are better expressed using `app`, `foldl/foldr`, `map` and many other higher-order functions provided by the Basis Library for manipulating lists, vectors and arrays. However, the Basis Library does not provide a function for iterating over a range of integer values. Fortunately, it is very easy to write one.

A fairly simple design

The following implementation imitates both the syntax and semantics of the OCaml `for`-loop.

```
datatype for = to of int * int
             | downto of int * int

infix to downto

val for =
    fn lo to up =>
       (fn f => let fun loop lo = if lo > up then ()
                                  else (f lo; loop (lo+1))
                in loop lo end)
     | up downto lo =>
       (fn f => let fun loop up = if up < lo then ()
                                  else (f up; loop (up-1))
                in loop up end)
```

For example,

```
for (1 to 9)
    (fn i => print (Int.toString i))
```

would print `123456789` and

```
for (9 downto 1)
    (fn i => print (Int.toString i))
```

would print `987654321`.

Straightforward formatting of nested loops

```
for (a to b)
```

```
    (fn i =>
        for (c to d)
            (fn j =>
                ...))
```

is fairly readable, but tends to cause the body of the loop to be indented quite deeply.

Off-by-one

The above design has an annoying feature. In practice, the upper bound of the iterated range is almost always excluded and most loops would subtract one from the upper bound:

```
for (0 to n-1) ...
for (n-1 downto 0) ...
```

It is probably better to break convention and exclude the upper bound by default, because it leads to more concise code and becomes idiomatic with very little practise. The iterator combinators described below exclude the upper bound by default.

Iterator combinators

While the simple `for`-function described in the previous section is probably good enough for many uses, it is a bit cumbersome when one needs to iterate over a cartesian product. One might also want to iterate over more than just consecutive integers. It turns out that one can provide a library of iterator combinators that allow one to implement iterators more flexibly.

Since the types of the combinators may be a bit difficult to infer from their implementations, let's first take a look at a signature of the iterator combinator library:

```
signature ITER =
  sig
    type 'a t = ('a -> unit) -> unit

    val return : 'a -> 'a t
    val >>= : 'a t * ('a -> 'b t) -> 'b t

    val none : 'a t

    val to : int * int -> int t
    val downto : int * int -> int t

    val inList : 'a list -> 'a t
    val inVector : 'a vector -> 'a t
    val inArray : 'a array -> 'a t

    val using : ('a, 'b) StringCvt.reader -> 'b -> 'a t

    val when : 'a t * ('a -> bool) -> 'a t
    val by : 'a t * ('a -> 'b) -> 'b t
    val @@ : 'a t * 'a t -> 'a t
    val ** : 'a t * 'b t -> ('a, 'b) product t

    val for : 'a -> 'a
  end
```

Several of the above combinators are meant to be used as infix operators. Here is a set of suitable infix declarations:

```
infix 2 to downto
infix 1 @@ when by
infix 0 >>= **
```

A few notes are in order:

- The `'a t` type constructor with the `return` and `>>=` operators forms a monad.
- The `to` and `downto` combinators will omit the upper bound of the range.
- `for` is the identity function. It is purely for syntactic sugar and is not strictly required.
- The `@@` combinator produces an iterator for the concatenation of the given iterators.
- The `**` combinator produces an iterator for the cartesian product of the given iterators.
    - See ProductType for the type constructor `('a, 'b) product` used in the type of the iterator produced by `**`.
- The `using` combinator allows one to iterate over slices, streams and many other kinds of sequences.
- `when` is the filtering combinator. The name `when` is inspired by OCaml's guard clauses.
- `by` is the mapping combinator.

The below implementation of the `ITER`-signature makes use of the following basic combinators:

```
fun const x _ = x
fun flip f x y = f y x
fun id x = x
fun opt fno fso = fn NONE => fno () | SOME ? => fso ?
fun pass x f = f x
```

Here is an implementation the `ITER`-signature:

```
structure Iter :> ITER =
  struct
    type 'a t = ('a -> unit) -> unit

    val return = pass
    fun (iA >>= a2iB) f = iA (flip a2iB f)

    val none = ignore

    fun (l to u) f = let fun `l = if l<u then (f l; `(l+1)) else () in `l end
    fun (u downto l) f = let fun `u = if u>l then (f (u-1); `(u-1)) else () in `u end

    fun inList ? = flip List.app ?
    fun inVector ? = flip Vector.app ?
    fun inArray ? = flip Array.app ?

    fun using get s f = let fun `s = opt (const ()) (fn (x, s) => (f x; `s)) (get s) in `s end

    fun (iA when p) f = iA (fn a => if p a then f a else ())
    fun (iA by g) f = iA (f o g)
    fun (iA @@ iB) f = (iA f : unit; iB f)
    fun (iA ** iB) f = iA (fn a => iB (fn b => f (a & b)))

    val for = id
  end
```

Note that some of the above combinators (e.g. `**`) could be expressed in terms of the other combinators, most notably `return` and `>>=`. Another implementation issue worth mentioning is that `downto` is written specifically to avoid computing `l-1`, which could cause an `Overflow`.

To use the above combinators the `Iter`-structure needs to be opened

```
open Iter
```

and one usually also wants to declare the infix status of the operators as shown earlier.

Here is an example that illustrates some of the features:

```
for (0 to 10 when (fn x => x mod 3 <> 0) ** inList ["a", "b"] ** 2 downto 1 by real)
    (fn x & y & z =>
       print ("("^Int.toString x^", \""^y^"\", "^Real.toString z^")\n"))
```

Using the `Iter` combinators one can easily produce more complicated iterators. For example, here is an iterator over a "triangle":

```
fun triangle (l, u) = l to u >>= (fn i => i to u >>= (fn j => return (i, j)))
```

Last edited on 2006-03-26 14:16:53 by VesaKarvonen.

ForeignFunctionInterface

MLton's foreign function interface (FFI) extends Standard ML and makes it easy to take the address of C global objects, access C global variables, call from SML to C, and call from C to SML.

Overview

- Foreign Function Interface Types
- Foreign Function Interface Syntax

Importing Code into SML

- Calling From SML To C
- Calling From SML To C Function Pointer

Exporting Code from SML

- Calling From C To SML

Last edited on 2005-12-02 04:19:50 by StephenWeeks.

ForeignFunctionInterfaceSyntax

MLton extends the syntax of SML with expressions that enable a ForeignFunctionInterface to C. The following description of the syntax uses some abbreviations.

C base type	*cBaseTy*	Foreign Function Interface types
C argument type	*cArgTy*	$cBaseTy_1$ `*` ... `*` $cBaseTy_n$ or `unit`
C return type	*cRetTy*	*cBaseTy* or `unit`
C function type	*cFuncTy*	*cArgTy* `->` *cRetTy*
C pointer type	*cPtrTy*	`MLton.Pointer.t`

The type annotation and the semicolon are not optional in the syntax of ForeignFunctionInterface expressions. However, the type is lexed, parsed, and elaborated as an SML type, so any type (including type abbreviations) may be used, so long as it elaborates to a type of the correct form.

Address

```
_address "C function or variable name" : cPtrTy;
```

Denotes the address of the C function or variable.

Symbol

```
_symbol "C variable name" attr... : (unit -> cBaseTy) * (cBaseTy -> unit);
```

Denotes the *getter* and *setter* for a C variable. The *cBaseTy*s must be identical.

`attr...` denotes a (possibly empty) sequence of attributes.

- `alloc` : allocate storage (and export a symbol) for the C variable

```
_symbol * : cPtrTy -> (unit -> cBaseTy) * (cBaseTy -> unit);
```

Denotes the *getter* and *setter* for a C pointer to a variable. The *cBaseTy*s must be identical.

Import

```
_import "CFunctionName" attr... : cFuncTy;
```

Denotes an SML function whose behavior is implemented by calling the C function. See Calling from SML to C for more details.

```
_import * attr... : cPtrTy -> cFuncTy;
```

Denotes a SML function whose behavior is implemented by calling a C function through a C function pointer.

`attr...` denotes a (possibly empty) sequence of attributes.

- `cdecl` : call with the `cdecl` calling convention.
- `stdcall` : call with the `stdcall` calling convention.

See Calling from SML to C function pointer for more details.

Export

```
_export "CFunctionName" attr... : cFuncTy -> unit;
```

Exports a C function with the name `CFunctionName` that can be used to call an SML function of the type *cFuncTy*. When the function denoted by the export expression is applied to an SML function `f`, subsequent C calls to `CFunctionName` will call `f`. It is an error to call `CFunctionName` before the export has been applied. The export may be applied more than once, with each application replacing any previous definition of `CFunctionName`.

`attr...` denotes a (possibly empty) sequence of attributes.

- `cdecl` : call with the `cdecl` calling convention.
- `stdcall` : call with the `stdcall` calling convention.

See Calling from C to SML for more details.

Last edited on 2005-12-01 04:25:23 by StephenWeeks.

ForeignFunctionInterfaceTypes

MLton's ForeignFunctionInterface only allows values of certain SML types to be passed between SML and C. The following types are allowed: `bool`, `char`, `int`, `real`, `word`. All of the different sizes of (fixed-sized) integers, reals, and words are supported as well: `Int8.int`, `Int16.int`, `Int32.int`, `Int64.int`, `Real32.real`, `Real64.real`, `Word8.word`, `Word16.word`, `Word32.word`, `Word64.word`. There is a special type, `MLton.Pointer.t`, for passing C pointers -- see MLtonPointer for details.

Arrays, refs, and vectors of the above types are also allowed. Because in MLton monomorphic arrays and vectors are exactly the same as their polymorphic counterpart, these are also allowed. Hence, `string`, `char vector`, and `CharVector.vector` are also allowed. Strings are not null terminated, unless you manually do so from the SML side.

Unfortunately, passing tuples or datatypes is not allowed because that would interfere with representation optimizations.

The C header file that `-export-header` generates includes `typedefs` for the C types corresponding to the SML types. Here is the mapping between SML types and C types.

SML type	C typedef	C type
`array`	`Pointer`	`char *`
`bool`	`Int32`	`long`
`char`	`Int8`	`char`
`Int8.int`	`Int8`	`char`
`Int16.int`	`Int16`	`short`
`Int32.int`	`Int32`	`long`
`Int64.int`	`Int64`	`long long`
`int`	`Int32`	`long`
`MLton.Pointer.t`	`Pointer`	`char *`
`Real32.real`	`Real32`	`float`
`Real64.real`	`Real64`	`double`
`real`	`Real64`	`double`
`ref`	`Pointer`	`char *`
`string`	`Pointer`	`char *` **(read-only)**
`vector`	`Pointer`	`char *` **(read-only)**
`Word8.word`	`Word8`	`unsigned char`
`Word16.word`	`Word16`	`unsigned short`
`Word32.word`	`Word32`	`unsigned long`
`Word64.word`	`Word64`	`unsigned long long`
`word`	`Word32`	`unsigned int`

Because MLton assumes that vectors and strings are read-only (and will perform optimizations that, for instance, cause them to share space), you must not modify the data pointed to by the `char *` in C code.

Although the C type of an array, ref, or vector is always `Pointer`, in reality, the object has the natural C representation. Your C code should cast to the appropriate C type if you want to keep the C compiler from complaining.

When calling an imported C function from SML that returns an array, ref, or vector result or when calling an exported SML function from C that takes an array, ref, or string argument, then the object must be an ML object allocated on the ML heap. (Although an array, ref, or vector object has the natural C representation, the object also has an additional header used by the SML runtime system.)

Last edited on 2005-12-01 04:27:38 by StephenWeeks.

FrontEnd

FrontEnd is a translation pass from source to the AST IntermediateLanguage.

Description

This pass performs lexing and parsing to produce an abstract syntax tree.

Implementation

front-end.sig front-end.fun

Details and Notes

The lexer is produced by MLLex from ml.lex .

The parser is produced by MLYacc from ml.grm .

The specifications for the lexer and parser were originally taken from SML/NJ (version 109.32), but have been heavily modified since then.

Last edited on 2005-12-02 04:20:24 by StephenWeeks.

FunctionalRecordUpdate

Functional record update is the copying of a record while replacing the values of some of the fields. Standard ML does not have explicit syntax for functional record update. We will show below how to implement functional record update in SML, with a little boilerplate code.

As an example, the functional update of the record

```
{a = 13, b = 14, c = 15}
```

with `c = 16` yields a new record

```
{a = 13, b = 14, c = 16}
```

Functional record update also makes sense with multiple simultaneous updates. For example, the functional update of the record above with `a = 18, c = 19` yields a new record

```
{a = 18, b = 14, c = 19}
```

One could easily imagine an extension of the SML that supports functional record update. For example

```
e with {a = 16, b = 17}
```

would create a copy of the record denoted by `e` with field `a` replaced with `16` and `b` replaced with `17`.

Since there is no such syntax in SML, we now show how to implement functional record update directly. We first give a simple implementation that has a number of problems. We then give an advanced implementation, that, while complex underneath, is a resuable library that admits simple use.

Simple implementation

To support functional record update on the record type

```
{a: 'a, b: 'b, c: 'c}
```

first, define an update function for each component.

```
fun withA ({a = _, b, c}, a) = {a = a, b = b, c = c}
fun withB ({a, b = _, c}, b) = {a = a, b = b, c = c}
fun withC ({a, b, c = _}, c) = {a = a, b = b, c = c}
```

Then, one can express `e with {a = 16, b = 17}` as

```
withB (withA (e, 16), 17)
```

With infix notation

```
infix withA withB withC
```

the syntax is almost as concise as a language extension.

```
e withA 16 withB 17
```

This approach suffers from the fact that the amount of boilerplate code is quadratic in the number of record fields. Furthermore, changing, adding, or deleting a field requires time proportional to the number of fields (because each `with` function must be changed). It is also annoying to have to define a `with` function, possibly with a fixity declaration, for each field.

Fortunately, there is a solution to these problems.

Advanced implementation

Using Fold one can define a family of `makeUpdate<N>` functions and single *update* operator `U` so that one can define a functional record update function for any record type simply by specifying a (trivial) isomorphism between that type and a product type. For example, suppose that we would like to do functional record update on records with fields `a` and `b`. Then one defines a function `updateAB` as follows.

```
val updateAB =
   fn z =>
   let
      fun p2r (v1 & v2) = {a = v1, b = v2}
      fun r2p {a = v1, b = v2} = (v1 & v2)
   in
      makeUpdate2 (p2r, p2r, r2p)
   end
   z
```

The functions `p2r` (think *product to record*) and `r2p` (think *record to product*) specify an isomorphism between `a,b` records and binary products. There is a second use of `p2r` to work around the lack of first-class polymorphism in SML.

With the definition of `updateAB` in place, the following expressions are valid.

```
updateAB {a = 13, b = "hello"} (U#b "goodbye") $
updateAB {a = 13.5, b = true} (U#b false) (U#a 12.5) $
```

As another example, suppose that we would like to do functional record update on records with fields `b`, `c`, and `d`. Then one defines a function `updateBCD` as follows.

```
val updateBCD =
   fn z =>
   let
      fun p2r (v1 & v2 & v3) = {b = v1, c = v2, d = v3}
      fun r2p {b = v1, c = v2, d = v3} = (v1 & v2 & v3)
   in
      makeUpdate3 (p2r, p2r, r2p)
   end
   z
```

With the definition of `updateBCD` in place, the following expression is valid.

```
updateBCD {b = 1, c = 2, d = 3} (U#c 4) (U#c 5) $
```

Note that not all fields need be updated and that the same field may be updated multiple times. Further note that the same `U` operator is used for all update functions (in the above, for both `updateAB` and `updateBCD`).

In general, to define a functional-record-update function on records with fields `f1`, `f2`, ..., `fN`, use the following template.

```
val update =
   fn z =>
   let
      fun p2r (v1 & v2 & ... & vn) = {f1 = v1, f2 = v2, ..., fn = vn}
      fun r2p {f1 = v1, f2 = v2, ..., fn = vn} = (v1 & v2 & ... & vn)
   in
      makeUpdateN (p2r, p2r, r2p)
   end
   z
```

With this, one can update a record as follows.

```
update {f1 = v1, ..., fn = vn} (U#fi1 vi1) ... (U#fim vim) $
```

If `makeUpdateN` is not already defined for the desired `N`, a generic `makeUpdate` function and special value, `A`, is defined so that one can use the following for `makeUpdateN`, where `A` is repeated `N` times.

```
makeUpdate A ... A $
```

The FunctionalRecordUpdate structure

Here is the implementation of functional record update.

```
structure FunctionalRecordUpdate =
   struct
      datatype ('x, 'y) u = X of 'x | Y of 'y

      val makeUpdate =
         fn z =>
         Fold.fold
         (((), (),
           fn f => f o X,
           fn (a, u) => case u of X x => x | _ => a),
          fn (p, up, _, _) => fn (p2r, p2r', r2p) => fn r =>
          Fold.fold ((p2r' (p id), up, r2p r),
                     fn (_, _, p) => p2r p))
         z

      val A =
         fn z =>
         Fold.step0
         (fn (_, _, p, up) =>
          (p, up, fn f => p (f o X) & (f o Y),
           fn (a & b, u) =>
           (case u of X x => up (a, x) | _ => a)
           & (case u of Y y => y | _ => b)))
         z

      fun makeUpdate2 z = makeUpdate A A $ z
      fun makeUpdate3 z = makeUpdate A A A $ z
      fun makeUpdate4 z = makeUpdate A A A A $ z

      fun U s v = Fold.step0 (fn (r, up, p) => (r, up, up (p, s r v)))
   end
```

The idea of `makeUpdate` is to inductively build the update function for n-ary product types. Each `A` supplied to `makeUpdate` adds one more level to the product. When finished with its arguments, `makeUpdate` begins a second fold, this time to process a variable number of `U` steps. The second fold begins by converting the supplied record to a product, using the supplied isomorphism (`p2r'`). Each step works by selecting a "path", `s r v`), from the inductively constructed product, reformatted by the supplied isomorphism to look like a record. Then, the inductively constructed update function is applied to the record-as-product and the path `up (p, s r v)` to yield a new record-as-product. Finally, at the end of the fold, the product is converted back to a record using the supplied isomorphism (`p2r`).

Efficiency

With MLton, the efficiency of this approach is as good as one would expect with the special syntax. Namely a sequence of updates will be optimized into a single record construction that copies the unchanged fields and fills in the changed fields with their new values.

Last edited on 2006-03-22 03:29:37 by MatthewFluet.

GarbageCollection

For a good introduction and overview to garbage collection, see Jones99.

MLton's garbage collector uses copying, mark-compact, and generational collection, automatically switching between them at run time based on the amount of live data relative to the amount of RAM. The runtime system tries to keep the heap within RAM if at all possible.

MLton's copying collector is a simple, two-space, breadth-first, Cheney-style collector. The design for the generational and mark-compact GC is based on Sansom91.

Design notes

- http://mlton.org/pipermail/mlton/2002-May/012420.html
  object layout and header word design

Also see

- Regions

Last edited on 2005-09-06 23:28:47 by MatthewFluet.

GenerativeDatatype

In Standard ML, datatype declarations are said to be *generative*, because each time a datatype declaration is evaluated, it yields a new type. Thus, any attempt to mix the types will lead to a type error at compile-time. The following program, which does not type check, demonstrates this.

```
functor F () =
   struct
      datatype t = T
   end
structure S1 = F ()
structure S2 = F ()
val _: S1.t -> S2.t = fn x => x
```

Generativity also means that two different datatype declarations define different types, even if they define identical constructors. The following program does not type check due to this.

```
datatype t = A | B
val a1 = A
datatype t = A | B
val a2 = A
val _ = if true then a1 else a2
```

Last edited on 2005-01-26 20:34:48 by MatthewFluet.

GenerativeException

In Standard ML, exception declarations are said to be *generative*, because each time an exception declaration is evaluated, it yields a new exception.

The following program demonstrates the generativity of exceptions.

```
exception E
val e1 = E
fun isE1 (e: exn): bool =
   case e of
      E => true
    | _ => false
exception E
val e2 = E
fun isE2 (e: exn): bool =
   case e of
      E => true
    | _ => false
fun pb (b: bool): unit =
   print (concat [Bool.toString b, "\n"])
val () = (pb (isE1 e1)
          ;pb (isE1 e2)
          ; pb (isE2 e1)
          ; pb (isE2 e2))
```

In the above program, two different exception declarations declare an exception `E` and a corresponding function that returns `true` only on that exception. Although declared by syntactically identical exception declarations, `e1` and `e2` are different exceptions. The program, when run, prints `true`, `false`, `false`, `true`.

A slight modification of the above program shows that even a single exception declaration yields a new exception each time it is evaluated.

```
fun f (): exn * (exn -> bool) =
   let
      exception E
   in
      (E, fn E => true | _ => false)
   end
val (e1, isE1) = f ()
val (e2, isE2) = f ()
fun pb (b: bool): unit =
   print (concat [Bool.toString b, "\n"])
val () = (pb (isE1 e1)
          ; pb (isE1 e2)
          ; pb (isE2 e1)
          ; pb (isE2 e2))
```

Each call to `f` yields a new exception and a function that returns `true` only on that exception. The program, when run, prints `true`, `false`, `false`, `true`.

Type Safety

Exception generativity is required for type safety. Consider the following valid SML program.

```
fun f (): ('a -> exn) * (exn -> 'a) =
   let
      exception E of 'a
   in
      (E, fn E x => x | _ => raise Fail "f")
   end
fun cast (a: 'a): 'b =
   let
      val (make: 'a -> exn, _) = f ()
      val (_, get: exn -> 'b) = f ()
   in
      get (make a)
   end
val _ = ((cast 13): int -> int) 14
```

If exceptions weren't generative, then each call `f ()` would yield the same exception constructor `E`. Then, our `cast` function could use `make: 'a -> exn` to convert any value into an exception and then `get: exn -> 'b` to convert that exception to a value of arbitrary type. If `cast` worked, then we could cast an integer as a function and apply. Of course, because of generative exceptions, this program raises `Fail "f"`.

Last edited on 2005-01-26 20:34:34 by MatthewFluet.

Glade

Glade is a tool for generating Gtk user interfaces.

WesleyTerpstra is working on a Glade->mGTK converter.

- http://mlton.org/pipermail/mlton/2004-December/016865.html

Last edited on 2005-12-02 07:11:13 by StephenWeeks.

Globalize

Globalize is an analysis pass for the SXML IntermediateLanguage, invoked from ClosureConvert.

Description

This pass marks values that are constant, allowing ClosureConvert to move them out to the top level so they are only evaluated once and do not appear in closures.

Implementation

globalize.sig globalize.fun

Details and Notes

Last edited on 2005-12-01 04:31:24 by StephenWeeks.

GnuMP

The GnuMP (GNU multiprecision library) is a library for arbitrary precision integer arithmetic. MLton uses the GnuMP to implement the SML Basis `IntInf` module.

There is a known problem with the GnuMP, where it requires a lot of stack space for some computations, e.g. `IntInf.toString` of a million digit number. If you run with stack size limited, you may see a segfault in such programs. This problem is mentioned in the GnuMP FAQ, where they describe two solutions.

- Increase (or unlimit) your stack space. From your program, use `setrlimit`, or from the shell, use `ulimit`.
- Configure and rebuild `libgmp` with `--disable-alloca`, which will cause it to allocate temporaries using `malloc` instead of on the stack.

Last edited on 2005-12-02 04:20:35 by StephenWeeks.

HaMLet

Hamlet is a Standard ML Implementation. It is intended as reference implementation of the Definition of Standard ML and not for serious practical work.

Last edited on 2005-12-01 04:32:39 by StephenWeeks.

HenryCejtin

I was one of the original developers of Mathematica (actually employee #1). My background is a combination of mathematics and computer science. Currently I am doing various things in Chicago.

Last edited on 2005-12-01 03:27:33 by HenryCejtin.

History

In April 1997, Stephen Weeks wrote a defunctorizer for Standard ML and integrated it with SML/NJ. The defunctorizer used SML/NJ's visible compiler and operated on the `Ast` intermediate representation produced by the SML/NJ front end. Experiments showed that defunctorization gave a speedup of up to six times over separate compilation and up to two times over batch compilation without functor expansion.

In August 1997, we began development of an independent compiler for SML. At the time the compiler was called `smlc`. By October, we had a working monomorphiser. By November, we added a polyvariant higher-order control-flow analysis. At that point, MLton was about 10,000 lines of code.

Over the next year and half, `smlc` morphed into a full-fledged compiler for SML. It was renamed MLton, and first released in March 1999.

From the start, MLton has been driven by whole-program optimization and an emphasis on performance. Also from the start, MLton has had a fast C FFI and `IntInf` based on the GNU multiprecision library. At its first release, MLton was 48,006 lines.

Between the March 1999 and January 2002, MLton grew to 102,541 lines, as we added a native code generator, mllex, mlyacc, a profiler, many optimizations, and many libraries including threads and signal handling.

During 2002, MLton grew to 112,204 lines and we had releases in April and September. We added support for cross compilation and used this to enable MLton to run on Cygwin/Windows and FreeBSD. We also made improvements to the garbage collector, so that it now works with large arrays and up to 4G of memory and so that it automatically uses copying, mark-compact, or generational collection depending on heap usage and RAM size. We also continued improvements to the optimizer and libraries.

During 2003, MLton grew to 122,299 lines and we had releases in March and July. We extended the profiler to support source-level profiling of time and allocation and to display call graphs. We completed the Basis Library implementation, and added new MLton-specific libraries for weak pointers and finalization. We extended the FFI to allow callbacks from C to SML. We added support for the Sparc/Solaris platform, and made many improvements to the C code generator.

Last edited on 2005-12-02 04:23:16 by MatthewFluet.

HowProfilingWorks

Here's how Profiling works. If profiling is on, the front end (elaborator) inserts `Enter` and `Leave` statements into the source program for function entry and exit. For example,

```
fun f n = if n = 0 then 0 else 1 + f (n - 1)
```

becomes

```
fun f n =
   let
      val () = Enter "f"
      val res = (if n = 0 then 0 else 1 + f (n - 1))
                handle e => (Leave "f"; raise e)
      val () = Leave "f"
   in
      res
   end
```

Actually there is a bit more information than just the source function name; there is also lexical nesting and file position.

Most of the middle of the compiler ignores, but preserves, `Enter` and `Leave`. However, so that profiling preserves tail calls, the Ssa shrinker has an optimization that notices when the only operations that cause a call to be a nontail call are profiling operations, and if so, moves them before the call, turning it into a tail call. If you observe a program that has a tail call that appears to be turned into a nontail when compiled with profiling, please report a bug.

There is the `checkProf` function in type-check.fun , which checks that the `Enter`/`Leave` statements match up.

In the backend, just before translating to the Machine IL, the profiler uses the `Enter`/`Leave` statements to infer the "local" portion of the control stack at each program point. The profiler then removes the `Enter`s/`Leave`s and inserts different information depending on which kind of profiling is happening. For time profiling, the profiler inserts labels that cover the code (i.e. each statement has a unique label in its basic block that prefixes it) and associates each label with the local control stack. For allocation profiling, the profiler inserts calls to a C function that will maintain byte counts. With stack profiling, the profiler also inserts a call to a C function at each nontail call in order to maintain information at runtime about what SML functions are on the stack.

At run time, the profiler associates counters (either clock ticks or byte counts) with source functions. When the program finishes, the profiler writes the counts out to the `mlmon.out` file. Then, `mlprof` uses source information stored in the executable to associate the counts in the `mlmon.out` file with source functions.

For time profiling, the profiler catches the `SIGPROF` signal 100 times per second and increments the appropriate counter, determined by looking at the label prefixing the current program counter and mapping that to the current source function.

Caveats

There may be a few missed clock ticks or bytes allocated at the very end of the program after the data is written.

Profiling has not been tested with signals or threads. In particular, stack profiling may behave strangely.

Last edited on 2005-12-01 04:35:20 by StephenWeeks.

Identifier

In Standard ML, there are syntactically two kinds of identifiers.

- Alphanumeric: starts with a letter or prime (`'`) and is followed by letters, digits, primes and underbars (_).

  Examples: `abc`, `ABC123`, `Abc_123`, `'a`.
- Symbolic: a sequence of the following

  ```
  ! % & $ # + - / : < = > ? @ | ~ ` ^ | *
  ```

  Examples: `+=`, `<=`, `>>`, `$`.

With the exception of =, reserved words can not be identifiers.

There are a number of different classes of identifiers, some of which have additional syntactic rules.

- Identifiers not starting with a prime.
  - value identifier (includes variables and constructors)
  - type constructor
  - structure identifier
  - signature identifier
  - functor identifier
- Identifiers starting with a prime.
  - type variable (must start with prime)
- Identifiers + numeric labels (`1`, `2`, ...).
  - record label

Last edited on 2005-01-18 15:02:21 by MatthewFluet.

Immutable

Immutable means not mutable, and is an adjective meaning "can not be modified". Most values in Standard ML are immutable. For example, constants, tuples, records, lists, and vectors are all immutable.

Last edited on 2004-12-08 18:51:10 by StephenWeeks.

ImperativeTypeVariable

In Standard ML, an imperative type variable is a type variable whose second character is a digit, as in `'1a` or `'2b`. Imperative type variables were used as an alternative to the ValueRestriction in an earlier version of SML, but no longer play a role. They are treated exactly as other type variables.

Last edited on 2004-11-29 22:58:32 by StephenWeeks.

ImplementExceptions

ImplementExceptions is a pass for the SXML IntermediateLanguage, invoked from SXMLSimplify.

Description

This pass implements exceptions.

Implementation

implement-exceptions.sig implement-exceptions.fun

Details and Notes

Last edited on 2005-12-01 04:37:39 by StephenWeeks.

ImplementHandlers

ImplementHandlers is a pass for the RSSA IntermediateLanguage, invoked from RSSASimplify.

Description

This pass implements the (threaded) exception handler stack.

Implementation

implement-handlers.sig implement-handlers.fun

Details and Notes

Last edited on 2005-12-01 04:38:13 by StephenWeeks.

ImplementProfiling

ImplementProfiling is a pass for the RSSA IntermediateLanguage, invoked from RSSASimplify.

Description

This pass implements profiling.

Implementation

profile.sig profile.fun

Details and Notes

See HowProfilingWorks.

Last edited on 2005-12-01 04:38:54 by StephenWeeks.

ImplementSuffix

ImplementSuffix is a pass for the SXML IntermediateLanguage, invoked from SXMLSimplify.

Description

This pass implements the `TopLevel_setSuffix` primitive, which installs a function to exit the program.

Implementation

implement-suffix.sig implement-suffix.fun

Details and Notes

ImplementSuffix works by introducing a new `ref` cell to contain the function of type `unit -> unit` that should be called on program exit.

* The following code (appropriately alpha-converted) is appended to the beginning of the SXML program:

```
 val z_0 =
   fn a_0 =>
   let
      val x_0 =
         "toplevel suffix not installed"
      val x_1 =
         MLton_bug (x_0)
   in
      x_1
   end
val topLevelSuffixCell =
   Ref_ref (z_0)
```

* Any occurrence of

```
val x_0 =
   TopLevel_setSuffix (f_0)
```

  is rewritten to

```
val x_0 =
   Ref_assign (topLevelSuffixCell, f_0)
```

* The following code (appropriately alpha-converted) is appended to the end of the SXML program:

```
val f_0 =
   Ref_deref (topLevelSuffixCell)
val z_0 =
   ()
val x_0 =
   f_0 z_0
```

Last edited on 2005-12-02 04:20:49 by StephenWeeks.

InfixingOperators

Fixity specifications are not part of signatures in Standard ML. When one wants to use a module that provides functions designed to be used as infix operators there are several obvious alternatives:

- Use only prefix applications. Unfortunately there are situations where infix applications lead to considerably more readable code.
- Make the fixity declarations at the top-level. This may lead to collisions and may be unsustainable in a large project. Pollution of the top-level should be avoided.
- Make the fixity declarations at each scope where you want to use infix applications. The duplication becomes inconvenient if the operators are widely used. Duplication of code should be avoided.
- Use non-standard extensions, such as the ML Basis system to control the scope of fixity declarations. This has the obvious drawback of reduced portability.
- Reuse existing infix operator symbols (`^`, `+`, `-`, ...). This can be convenient when the standard operators aren't needed in the same scope with the new operators. On the other hand, one is limited to the standard operator symbols and the code may appear confusing.

None of the obvious alternatives is best in every case. The following describes a slightly less obvious alternative that can sometimes be useful. The idea is to approximate Haskell's special syntax for treating any identifier enclosed in grave accents (backquotes) as an infix operator. In Haskell, instead of writing the prefix application `f x y` one can write the infix application `` x `f` y ``.

Infixing operators

Let's first take a look at the definitions of the operators:

```
infix  3 <\     fun x <\ f = fn y => f (x, y)     (* Left section      *)
infix  3 \>     fun f \> y = f y                  (* Left application  *)
infixr 3 />     fun f /> y = fn x => f (x, y)     (* Right section     *)
infixr 3 </     fun x </ f = f x                  (* Right application *)

infix  2 o  (* See motivation below *)
infix  0 :=
```

The left and right sectioning operators, `<\` and `/>`, are useful in SML for partial application of infix operators. ML For the Working Programmer describes curried functions `secl` and `secr` for the same purpose on pages 179-181. For example,

```
List.map (op- /> y)
```

is a function for subtracting `y` from a list of integers and

```
List.exists (x <\ op=)
```

is a function for testing whether a list contains an `x`.

Together with the left and right application operators, `\>` and `</`, the sectioning operators provide a way to treat any binary function (i.e. a function whose domain is a pair) as an infix operator. In general,

```
x0 <\f1\> x1 <\f2\> x2 ... <\fN\> xN  =  fN (... f2 (f1 (x0, x1), x2) ..., xN)
```

and

```
xN </fN/> ... x2 </f2/> x1 </f1/> x0  =  fN (xN, ... f2 (x2, f1 (x1, x0)) ...)
```

Examples

As a fairly realistic example, consider providing a function for sequencing comparisons:

```
structure Order (* ... *) =
   struct
      (* ... *)
      val orWhenEq = fn (EQUAL, th) => th ()
                      | (other,  _) => other
      (* ... *)
   end
```

Using `orWhenEq` and the infixing operators, one can write a `compare` function for triples as

```
fun compare (fad, fbe, fcf) ((a, b, c), (d, e, f)) =
    fad (a, d) <\Order.orWhenEq\> `fbe (b, e) <\Order.orWhenEq\> `fcf (c, f)
```

where ` is defined as

```
fun `f x = fn () => f x
```

Although `orWhenEq` can be convenient (try rewriting the above without it), it is probably not useful enough to be defined at the top level as an infix operator. Fortunately we can use the infixing operators and don't have to.

Another fairly realistic example would be to use the infixing operators with the technique described on the Printf page. Assuming that you would have a `Printf` module binding `printf`, `` ` ``, and formatting combinators named `int` and `string`, you could write

```
let open Printf in
  printf (`"Here's an int "<\int\>" and a string "<\string\>".") 13 "foo" end
```

without having to duplicate the fixity declarations. Alternatively, you could write

```
P.printf (P.`"Here's an int "<\P.int\>" and a string "<\P.string\>".") 13 "foo"
```

assuming you have the made the binding

```
structure P = Printf
```

Application and piping operators

The left and right application operators may also provide some notational convenience on their own. In general,

```
f \> x1 \> ... \> xN = f x1 ... xN
```

and

```
xN </ ... </ x1 </ f = f x1 ... xN
```

If nothing else, both of them can eliminate parentheses. For example,

```
foo (1 + 2) = foo \> 1 + 2
```

The left and right application operators are related to operators that could be described as the right and left piping operators:

```
infix  1 >|     val op>| = op</      (* Left pipe *)
infixr 1 |<     val op|< = op\>      (* Right pipe *)
```

As you can see, the left and right piping operators, >| and |<, are the same as the right and left application operators, respectively, except the associativities are reversed and the binding strength is lower. They are useful for piping data trough a sequence of operations. In general,

```
  x >| f1 >| ... >| fN
= fN (... (f1 x) ...)
= (fN o ... o f1) x
```

and

```
  fN |< ... |< f1 |< x
= fN (... (f1 x) ...)
= (fN o ... o f1) x
```

The right piping operator, `|<`, is provided by the Haskell prelude as `$`. It can be convenient in CPS or continuation passing style.

A use for the left piping operator is with parsing combinators. In a strict language, like SML, eta-reduction is generally unsafe. Using the left piping operator, parsing functions can be formatted conveniently as

```
fun parsingFunc input =
   input >| (* ... *)
         || (* ... *)
         || (* ... *)
```

where `||` is supposed to be a combinator provided by the parsing combinator library.

About precedences

You probably noticed that we redefined the precedences of the function composition operator `o` and the assignment operator `:=`. Doing so is not strictly necessary, but can be convenient and should be relatively safe. Consider the following motivating examples from Wesley W. Terpstra relying on the redefined precedences:

```
Word8.fromInt o Char.ord o s <\String.sub
(* Combining sectioning and composition *)

x := s <\String.sub\> i
(* Assigning the result of an infixed application *)
```

In imperative languages, assignment usually has the lowest precedence (ignoring statement separators). The precedence of `:=` in the Basis library is perhaps unnecessarily high, because an expression of the form `r := x` always returns a unit, which makes little sense to combine with anything. Dropping `:=` to the lowest precedence level makes it behave more like in other imperative languages.

The case for `o` is different. With the exception of `before` and `:=`, it doesn't seem to make much sense to use `o` with any of the operators defined by the Basis library in an unparenthesized expression. This is simply because none of the other operators deal with functions. It would seem that the precedence of `o` could be chosen completely arbitrarily from the set `{1, ..., 9}` without having any adverse effects with respect to other infix operators defined by the Basis library.

Design of the symbols

The closest approximation of Haskell's ``x `f` y`` syntax achievable in Standard ML would probably be something like ``x `f^ y``, but `^` is already used for string concatenation by the Basis library. Other combinations of the characters `` ` `` and `^` would be possible, but none seems clearly the best visually. The symbols `<\`, `\>`, `</` and `/>` are reasonably concise and have a certain self-documenting appearance and symmetry, which can help to remember them. As the names suggest, the symbols of the piping operators `>|` and `|<` are inspired by Unix shell pipelines.

Last edited on 2006-08-01 10:32:53 by VesaKarvonen.

Inline

Inline is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This pass inlines SSA functions using a size-based metric.

Implementation

inline.sig inline.fun

Details and Notes

The Inline pass can be invoked to use one of three metrics:

- `NonRecursive(product, small)` -- inline any function satisfying `(numCalls - 1) * (size - small) <= product`, where `numCalls` is the static number of calls to the function and `size` is the size of the function.
- `Leaf(size)` -- inline any leaf function smaller than `size`
- `LeafNoLoop(size)` -- inline any leaf function without loops smaller than `size`

Last edited on 2005-12-01 04:42:09 by StephenWeeks.

InsertLimitChecks

InsertLimitChecks is a pass for the RSSA IntermediateLanguage, invoked from RSSASimplify.

Description

This pass inserts limit checks.

Implementation

limit-check.sig limit-check.fun

Details and Notes

Last edited on 2005-12-01 04:42:38 by StephenWeeks.

InsertSignalChecks

InsertSignalChecks is a pass for the RSSA IntermediateLanguage, invoked from RSSASimplify.

Description

This pass inserts signal checks.

Implementation

limit-check.sig limit-check.fun

Details and Notes

Last edited on 2005-12-02 04:21:03 by StephenWeeks.

Installation

MLton runs on a variety of platforms and is distributed in both source and binary form. The format for the binary package depends on the platform. The binary package will install under `/usr` or `/usr/local`, depending on the platform. A `.tgz` or `.tbz` binary package should be extracted in the root directory. If you install MLton somewhere else, you must set the `lib` variable in the `/usr/bin/mlton` script to the directory that contains the libraries (`/usr/lib/mlton` by default).

MLton requires that you have the GNU multiprecision library installed on your machine. MLton must be able to find both the `gmp.h` include file and the `libgmp.a` or `libgmp.so` library. If you see the error message `gmp.h: No such file or directory`, you should copy `gmp.h` to `/usr/lib/mlton/self/include`. If you see the error message `/usr/bin/ld: cannot find -lgmp`, you should add a `-link-opt -L` argument in the `/usr/bin/mlton` script so that the linker can find `libgmp`. If, for example, `libgmp.a` is in `/tmp`, then add `-link-opt -L/tmp`.

Installation of MLton creates the following files and directories.

- `/usr/bin/mllex`
  The MLLex lexer generator.
- `/usr/bin/mlnlffigen`
  The ML-NLFFI tool.
- `/usr/bin/mlprof`
  A Profiling tool.
- `/usr/bin/mlton`
  A script to call the compiler. This script may be moved anywhere, however, it makes use of files in `/usr/lib/mlton`.
- `/usr/bin/mlyacc`
  The MLYacc parser generator.
- `/usr/lib/mlton`
  Directory containing libraries and include files needed during compilation.
- `/usr/share/man/man1/mllex.1, mlnlffigen.1, mlprof.1, mlton.1, mlyacc.1`
  Man pages.
- `/usr/share/doc/mlton`

Directory containing the user guide for MLton, mllex, and mlyacc, as well as example SML programs (in the `examples` dir), and license information.

Hello, World!

Once you have installed MLton, create a file called `hello-world.sml` with the following contents.

```
print "Hello, world!\n";
```

Now create an executable, `hello-world`, with the following command.

```
mlton hello-world.sml
```

You can now run `hello-world` to verify that it works. There are more small examples in `/usr/share/doc/mlton/examples`.

Installation on Cygwin

When installing the Cygwin `tgz`, you should use Cygwin's `bash` and `tar`. The use of an archiving tool that is not aware of Cygwin's mounts will put the files in the wrong place.

Last edited on 2005-12-08 16:14:00 by StephenWeeks.

IntermediateLanguage

MLton uses a number of intermediate languages in translating from the input source program to low-level code. Here is a list in the order which they are translated to.

- AST. Pretty close to the source.
- CoreML. Explicitly typed, no module constructs.
- XML. Polymorphic, HigherOrder.
- SXML. SimplyTyped, HigherOrder.
- SSA. SimplyTyped, FirstOrder.
- SSA2. SimplyTyped, FirstOrder.
- RSSA. Explicit data representations.
- Machine. Untyped register transfer language.

Last edited on 2004-11-29 02:16:14 by MatthewFluet.

IntroduceLoops

IntroduceLoops is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This pass rewrites any SSA function that calls itself in tail position into one with a local loop and no self tail calls.

A SSA function like

```
fun F (arg_0, arg_1) = L_0 ()
  ...
  L_16 (x_0)
    ...
    F (z_0, z_1) Tail
  ...
```

becomes

```
fun F (arg_0', arg_1') = loopS_0 ()
  loopS_0 ()
    loop_0 (arg_0', arg_1')
  loop_0 (arg_0, arg_1)
    L_0 ()
  ...
  L_16 (x_0)
    ...
    loop_0 (z_0, z_1)
  ...
```

Implementation

introduce-loops.sig introduce-loops.fun

Details and Notes

Last edited on 2005-12-01 04:46:37 by StephenWeeks.

JesperLouisAndersen

Jesper Louis Andersen

Jesper Louis Andersen is an undergraduate student at DIKU, the department of computer science, Copenhagen university. His contributions to MLton are few, though he has made the port of MLton to the NetBSD and OpenBSD platforms.

His general interests in computer science are compiler theory, language theory, algorithms and datastructures and programming. His assets are his general knowledge of UNIX systems, knowledge of system administration, knowledge of operating system kernels; NetBSD in particular.

He was employed by the university as a system administrator for 2 years, which has set him back somewhat in his studies. Currently he is trying to learn mathematics (real analysis, general topology, complex functional analysis and algebra).

Projects using MLton

A register allocator

For internal use at a compiler course at DIKU. It is written in the literate programming style and implements the *Iterated Register Coalescing* algorithm by Lal George and Andrew Appel http://citeseer.ist.psu.edu/george96iterated.html. The status of the project is that it is unfinished. Most of the basic parts of the algorithm is done, but the interface to the students (simple) datatype takes some conversion.

A configuration management system in SML

At this time, only loose plans exists for this. The plan is to build a Configuration Management system on the principles of the OpenCM system, see http://www.opencm.org/docs.html. The basic idea is to unify "naming" and "identity" into one by uniquely identifying all objects managed in the repository by the use of cryptographic checksums. This mantra guides the rest of the system, providing integrity, accessibility and confidentiality.

Last edited on 2004-12-06 13:45:22 by JesperLouisAndersen.

JohnnyAndersen

Johnny Andersen (aka Anoq of the Sun)

Here is a picture in front of the academy building at the University of Athens, Greece, taken in September 2003.

image

Last edited on 2004-10-27 18:12:11 by eponym.

KnownCase

KnownCase is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This pass duplicates and simplifies `Case` transfers when the constructor of the scrutinee is known.

Uses Restore.

For example, the program

```
val rec last =
   fn [] => 0
    | [x] => x
    | _ :: l => last l

val _ = 1 + last [2, 3, 4, 5, 6, 7]
```

gives rise to the SSA function

```
fun last_0 (x_142) = loopS_1 ()
  loopS_1 ()
    loop_11 (x_142)
  loop_11 (x_143)
    case x_143 of
      nil_1 => L_73 | ::_0 => L_74
  L_73 ()
    return global_5
  L_74 (x_145, x_144)
    case x_145 of
      nil_1 => L_75 | _ => L_76
  L_75 ()
    return x_144
  L_76 ()
    loop_11 (x_145)
```

which is simplified to

```
fun last_0 (x_142) = loopS_1 ()
  loopS_1 ()
    case x_142 of
      nil_1 => L_73 | ::_0 => L_118
  L_73 ()
    return global_5
  L_118 (x_230, x_229)
    L_74 (x_230, x_229, x_142)
  L_74 (x_145, x_144, x_232)
    case x_145 of
      nil_1 => L_75 | ::_0 => L_114
  L_75 ()
    return x_144
  L_114 (x_227, x_226)
    L_74 (x_227, x_226, x_145)
```

Implementation

known-case.sig known-case.fun

Details and Notes

One interesting aspect of KnownCase, is that it often has the effect of unrolling list traversals by one iteration, moving the `nil`/`::` check to the end of the loop, rather than the beginning.

Last edited on 2005-12-02 04:21:19 by StephenWeeks.

LLVM

LLVM (Low Level Virtual Machine) is an abstract machine, optimizer, and code generator. It might make a nice backend for MLton, and there has been some discussion about this on the MLton list.

http://mlton.org/pipermail/mlton/2005-November/028263.html

The latest is that LLVM's gcc variant has been used in place of gcc, and so there has been no work toward changing MLton to target LLVM's IL directly.

http://mlton.org/pipermail/mlton/2006-August/029021.html

Also see

- CMinusMinus

Last edited on 2006-09-04 20:25:17 by StephenWeeks.

LambdaCalculus

The lambda calculus is the formal system underlying Standard ML.

Last edited on 2006-03-28 00:58:46 by StephenWeeks.

LambdaFree

LambdaFree is an analysis pass for the SXML IntermediateLanguage, invoked from ClosureConvert.

Description

This pass descends the entire SXML program and attaches a property to each `Lambda` `PrimExp.t` in the program. Then, you can use `lambdaFree` and `lambdaRec` to get free variables of that `Lambda`.

Implementation

lambda-free.sig lambda-free.fun

Details and Notes

For `Lambda`s bound in a `Fun` dec, `lambdaFree` gives the union of the frees of the entire group of mutually recursive functions. Hence, `lambdaFree` for every `Lambda` in a single `Fun` dec is the same. Furthermore, for a `Lambda` bound in a `Fun` dec, `lambdaRec` gives the list of other functions bound in the same dec defining that `Lambda`. For example:

```
val rec f = fn x => ... y ... g ... f ...
and g = fn z => ... f ... w ...

 * lambdaFree(fn x =>) = [y, w]
 * lambdaFree(fn z =>) = [y, w]
 * lambdaRec(fn x =>) = [g, f]
 * lambdaRec(fn z =>) = [f]
```

Last edited on 2005-12-02 04:21:28 by StephenWeeks.

LanguageChanges

We are sometimes asked to modify MLton to change the language it compiles. In short, we are very conservative about making such changes. There are a number of reasons for this.

- The Definition of Standard ML is an extremely high standard of specification. The value of the Definition would be significantly diluted by changes that are not specified at an equally high level, and the dilution increases with the complexity of the language change and its interaction with other language features.
- The SML community is small and there are a number of SML implementations. Without an agreed-upon standard, it becomes very difficult to port programs between compilers, and the community would be balkanized.
- Our main goal is to enable programmers to be as effective as possible with MLton/SML. There are a number of improvements other than language changes that we could spend our time on that would provide more benefit to programmers.
- The more the language that MLton compiles changes over time, the more difficult it is to use MLton as a stable platform for serious program development.

Despite these drawbacks, we have extended SML in a couple of cases.

- Foreign function interface
- ML Basis system

We allow these language extensions because they provide functionality that is impossible to achieve without them. The Definition does not define a foreign function interface. So, we must either extend the language or greatly restrict the class of programs that can be written. Similarly, the Definition does not provide a mechanism for namespace control at the module level, making it impossible to deliver packaged libraries and have a hope of users using them without name clashes. The ML Basis system addresses this problem. We have also provided a formal specification of the ML Basis system at the level of the Definition.

Also see

- http://mlton.org/pipermail/mlton/2004-August/016165.html
- http://mlton.org/pipermail/mlton-user/2004-December/000320.html

---

Last edited on 2005-09-06 23:28:57 by MatthewFluet.

Lazy

In a lazy (or non-strict) language, the arguments to a function are not evaluated before calling the function. Instead, the arguments are suspended and only evaluated by the function if needed.

Standard ML is an eager (or strict) language, not a lazy language. However, it is easy to delay evaluation of an expression in SML by creating a *thunk*, which is a nullary function. In SML, a thunk is written `fn () => e`. Another essential feature of laziness is *memoization*, meaning that once a suspended argument is evaluated, subsequent references look up the value. We can express this in SML with a function that maps a thunk to a memoized thunk.

```
signature LAZY =
   sig
      val lazy: (unit -> 'a) -> unit -> 'a
   end
```

This is easy to implement in SML.

```
structure Lazy: LAZY =
   struct
      fun lazy (th: unit -> 'a): unit -> 'a =
         let
            val r: 'a option ref = ref NONE
         in
            fn () =>
            case !r of
               NONE =>
                  let
                     val a = th ()
                     val () = r := SOME a
                  in
                     a
                  end
             | SOME a => a
         end
   end
```

Last edited on 2005-01-26 20:33:55 by MatthewFluet.

Libraries

In theory every strictly conforming Standard ML program should run on MLton. However, often large SML projects use implementation specific features so some "porting" is required. Here is a partial list of software that is known to run on MLton.

- Concurrency: ConcurrentML - distributed with MLton
- Graphics
    - GTK: mGTK.
    - OpenGL
- Lex-like lexer generator: MLLex - distributed with MLton.
- Regular expressions
    - The SMLNJLibrary has a regexp module.
    - The internal MLton library has a regexp module which we hope to cleanup and make more accessible someday. See regexp.sig regexp.sml
- SMLNJLibrary - distributed with MLton
- CKitLibrary - distributed with MLton
- ML-NLFFI - distributed with MLton
- MLRISCLibrary - distributed with MLton
- sml-lib, a grab bag of libraries for MLton and other SML implementations.
- Swerve, an HTTP server.
- Twelf. The version in CVS should compile out of the box.
- XML: fxp
- Yacc-like parser generator: MLYacc - distributed with MLton.

Ports in progress

Contact us for details on any of these.

- MLDoc http://people.cs.uchicago.edu/~jhr/tools/ml-doc.html
- Unicode

More

More projects using MLton can be seen on the Users page.

Software for SML implementations other than MLton

- PostgreSQL
    - Moscow ML: http://www.dina.kvl.dk/~sestoft/mosmllib/Postgres.html
    - SML/NJ NLFFI: http://smlweb.sourceforge.net/smlsql/
- Web:
    - ML Kit: SMLserver (a plugin for AOLserver)
    - Moscow ML: ML Server Pages (support for PHP-style CGI scripting)
    - SML/NJ: smlweb

Last edited on 2006-03-04 17:04:10 by MatthewFluet.

License

Web Site

In order to allow the maximum freedom for the future use of the content in this web site, we require that contributions to the web site be dedicated to the public domain. That means that you can only add works that are already in the public domain, or that you must hold the copyright on the work that you agree to dedicate the work to the public domain.

By contributing to this web site, you agree to dedicate your contribution to the public domain.

Software

As of 20050812, MLton software is licensed under the BSD-style license below. By contributing code to the project, you agree to release the code under this license. Contributors can retain copyright to their contributions by asserting copyright in their code. Contributors may also add to the list of copyright holders in `doc/license/MLton-LICENSE`, which appears below.

```
This is the license for MLton, a whole-program optimizing compiler for
the Standard ML programming language.  Send comments and questions to
MLton@mlton.org.

MLton COPYRIGHT NOTICE, LICENSE AND DISCLAIMER.

Copyright (C) 1999-2006 Henry Cejtin, Matthew Fluet, Suresh
   Jagannathan, and Stephen Weeks.
Copyright (C) 1997-2000 by the NEC Research Institute

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both the copyright notice and this permission notice and warranty
disclaimer appear in supporting documentation, and that the name of
the above copyright holders, or their entities, not be used in
advertising or publicity pertaining to distribution of the software
without specific, written prior permission.

The above copyright holders disclaim all warranties with regard to
this software, including all implied warranties of merchantability and
fitness. In no event shall the above copyright holders be liable for
any special, indirect or consequential damages or any damages
whatsoever resulting from loss of use, data or profits, whether in an
action of contract, negligence or other tortious action, arising out
of or in connection with the use or performance of this software.
```

Last edited on 2006-10-06 17:07:40 by StephenWeeks.

LineDirective

To aid in the debugging of code produced by program generators such as Noweb, MLton supports comments with line directives of the form `(*#line` *line*`.`*col* `"`*file*`"*)`. Here, *line* and *col* are sequences of decimal digits and *file* is the source file. A line directive causes the front end to believe that the character following the right parenthesis is at the line and column of the specified file. A line directive only affects the reporting of error messages and does not affect program semantics (except for functions like `MLton.Exn.history` that report source file positions). Syntactically invalid line directives are ignored. To prevent incompatibilities with SML, the file name may not contain the character sequence `*)`.

Last edited on 2005-12-02 04:21:37 by StephenWeeks.

LocalFlatten

LocalFlatten is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This pass flattens arguments to SSA blocks.

A block argument is flattened as long as it only flows to selects and there is some tuple constructed in this function that flows to it.

Implementation

local-flatten.sig local-flatten.fun

Details and Notes

Last edited on 2005-12-01 04:52:47 by StephenWeeks.

LocalRef

LocalRef is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This pas optimizes `ref` cells local to a SSA function:

- global `refs` only used in one function are moved to the function
- `refs` only created, read from, and written to (i.e., don't escape) are converted into function local variables

Uses Multi and Restore.

Implementation

local-ref.sig local-ref.fun

Details and Notes

Moving a global `ref` requires the Multi analysis, because a global `ref` can only be moved into a function that is executed at most once.

Conversion of non-escaping `refs` is structured in three phases:

- analysis -- a variable `r = Ref_ref x` escapes if
    - `r` is used in any context besides `Ref_assign (r, _)` or `Ref_deref r`
    - all uses `r` reachable from a (direct or indirect) call to `Thread_copyCurrent` are of the same flavor (either `Ref_assign` or `Ref_deref`); this also requires the Multi analysis.
- transformation
    - rewrites `r = Ref_ref x` to `r = x`
    - rewrites `_ = Ref_assign (r, y)` to `r = y`
    - rewrites `z = Ref_deref r` to `z = r` Note that the resulting program violates the SSA condition.
- Restore -- restore the SSA condition.

Last edited on 2005-12-02 03:27:53 by StephenWeeks.

LoopInvariant

LoopInvariant is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This pass removes loop invariant arguments to local loops.

```
loop (x, y)
  ...
...
  loop (x, z)
...
```

becomes

```
loop' (x, y)
  loop (y)
loop (y)
  ...
...
  loop (z)
...
```

Implementation

loop-invariant.sig loop-invariant.fun

Details and Notes

Last edited on 2005-12-01 04:53:57 by StephenWeeks.

ML

ML stands for *meta language*. ML was originally designed in the 1970s as a programming language to assist theorem proving in the logic LCF. In the 1980s, ML split into two variants, Standard ML and OCaml, both of which are still used today.

Last edited on 2004-12-06 06:00:35 by StephenWeeks.

MLBasis

The ML Basis system extends Standard ML to support programming-in-the-very-large, namespace management at the module level, separate delivery of library sources, and more. While Standard ML modules are a sophisticated language for programming-in-the-large, it is difficult, if not impossible, to accomplish a number of routine namespace management operations when a program draws upon multiple libraries provided by different vendors.

The ML Basis system is a simple, yet powerful, approach that builds upon the programmer's intuitive notion (and the Definition of Standard ML's formal notion) of the top-level environment (a *basis*). The system is designed as a natural extension of Standard ML; the formal specification of the ML Basis system (pdf) is given in the style of the Definition.

Here are some of the key features of the ML Basis system:

1. Explicit file order: The order of files (and, hence, the order of evaluation) in the program is explicit. The ML Basis system's semantics are structured in such a way that for any well-formed project, there will be exactly one possible interpretation of the project's syntax, static semantics, and dynamic semantics.
2. Implicit dependencies: A source file (corresponding to a SML top-level declaration) is elaborated in the environment described by preceding declarations. It is not necessary to explicitly list the dependencies of a file.
3. Scoping and renaming: The ML Basis system provides mechanisms for limiting the scope of (i.e, hiding) and renaming identifiers.
4. No naming convention for finding the file that defines a module. To import a module, its defining file must appear in some ML Basis file.

Next steps

- MLBasisSyntaxAndSemantics
- MLBasisExamples
- MLBasisPathMap
- MLBasisAnnotations
- MLBasisAvailableLibraries

Last edited on 2005-12-01 20:09:32 by StephenWeeks.

MLBasisAnnotationExamples

Here are some example uses of MLBasisAnnotations.

Eliminate spurious warnings in automatically generated code

Programs that automatically generate source code can often produce nonexhaustive matches, relying on invariants of the generated code to ensure that the matches never fail. A programmer may wish to elide the nonexhaustive match warnings from this code, in order that legitimate warnings are not missed in a flurry of false positives. To do so, the programmer simply annotates the generated code with the `nonexhaustiveMatch ignore` annotation:

```
local
  $(GEN_ROOT)/gen-lib.mlb

  ann "nonexhaustiveMatch ignore" in
    foo.gen.sml
  end
in
  signature FOO
  structure Foo
end
```

Deliver a library

Standard ML libraries can be delivered via `.mlb` files. Authors of such libraries should strive to be mindful of the ways in which programmers may choose to compile their programs. For example, although the defaults for `sequenceNonUnit` and `warnUnused` are `ignore` and `false`, periodically compiling with these annotations defaulted to `warn` and `true` can help uncover likely bugs. However, a programmer is unlikely to be interested in unused modules from an imported library, and the behavior of `sequenceNonUnit error` may be incompatible with some libraries. Hence, a library author may choose to deliver a library as follows:

```
ann
  "nonexhaustiveMatch warn" "redundantMatch warn"
  "sequenceNonUnit warn"
  "warnUnused true" "forceUsed"
in
  local
    file1.sml
    ...
    filen.sml
  in
    functor F1
    ...
    signature S1
    ...
    structure SN
    ...
  end
end
```

The annotations `nonexhaustiveMatch warn`, `redundantMatch warn`, and `sequenceNonUnit warn` have the obvious effect on elaboration. The annotations `warnUnused true` and `forceUsed` work in conjunction --- warning on any identifiers that do not contribute to the exported modules, and preventing warnings on exported modules that are not used in the remainder of the program. Many of the available libraries are delivered with these annotations.

Last edited on 2005-12-01 19:45:40 by StephenWeeks.

MLBasisAnnotations

ML Basis annotations control options that affect the elaboration of SML source files. Conceptually, a basis file is elaborated in a default annotation environment (just as it is elaborated in an empty basis). The declaration `ann "ann" in basdec end` merges the annotation *ann* with the "current" annotation environment for the elaboration of *basdec*. To allow for future expansion, `"ann"` is lexed as a single SML string constant. To conveniently specify multiple annotations, the following derived form is provided:

```
ann "ann" ("ann")+ in basdec end ==>
ann "ann" in ann ("ann")+ in basdec end end
```

Here are the available annotations. In the explanation below, for annotations that take an argument, the first value listed is the default.

`allowFFI {false|true}`

If `true`, allow `_address`, `_export`, `_import`, and `_symbol` expressions to appear in source files. See ForeignFunctionInterface.

`forceUsed`

Force all identifiers in the basis denoted by the body of the `ann` to be considered used; use in conjunction with `warnUnused true`.

`nonexhaustiveExnMatch {default|ignore}`

If `ignore`, suppress errors and warnings about nonexhaustive matches that arise solely from unmatched exceptions. If `default`, follow the behavior of `nonexhaustiveMatch`.

`nonexhaustiveMatch {warn|error|ignore}`

If `error` or `warn`, report nonexhaustive matches. An error will abort a compile, while a warning will not.

`redundantMatch {warn|error|ignore}`

If `error` or `warn`, report redundant matches. An error will abort a compile, while a warning will not.

`sequenceNonUnit {ignore|error|warn}`

If `error` or `warn`, report when `e1` is not of type `unit` in the sequence expression `(e1; e2)`. This can be helpful in detecting curried applications that are mistakenly not fully applied. To silence spurious messages, you can use `ignore e1`.

`warnUnused {false|true}`

Report unused identifiers.

Next Steps

- MLBasisAnnotationExamples

Last edited on 2005-12-01 19:50:46 by StephenWeeks.

MLBasisAvailableLibraries

MLton comes with the following ML Basis files available.

`$(SML_LIB)/basis/basis.mlb`

The Basis Library.

`$(SML_LIB)/basis/basis-1997.mlb`

The (deprecated) 1997 version of the Basis Library.

`$(SML_LIB)/basis/mlton.mlb`

The MLton structure and signatures.

`$(SML_LIB)/basis/sml-nj.mlb`

The SMLofNJ structure and signature.

`$(SML_LIB)/basis/unsafe.mlb`

The Unsafe structure and signature.

`$(SML_LIB)/mlyacc-lib/mlyacc-lib.mlb`

Modules used by parsers built with MLYacc.

`$(SML_LIB)/cml/cml.mlb`

ConcurrentML, a library for message-passing concurrency.

`$(SML_LIB)/mlnlffi-lib/mlnlffi-lib.mlb`

ML-NLFFI, a library for foreign function interfaces.

`$(SML_LIB)/smlnj-lib/...`

SMLNJLibrary, a collection of libraries distributed with SML/NJ.

`$(SML_LIB)/ckit-lib/ckit-lib.mlb`

CKitLibrary, a library for C source code.

Basis fragments

There are a number of specialized ML Basis files for importing fragments of the Basis Library that can not be expressed within SML.

`$(SML_LIB)/basis/pervasive-types.mlb`

The top-level types and constructors of the Basis Library.

`$(SML_LIB)/basis/pervasive-exns.mlb`

The top-level exception constructors of the Basis Library.

`$(SML_LIB)/basis/pervasive-vals.mlb`

The top-level values of the Basis Library, without infix status.

`$(SML_LIB)/basis/overloads.mlb`

The top-level overloaded values of the Basis Library, without infix status.

`$(SML_LIB)/basis/equal.mlb`

The polymorphic equality = and inequality <> values, without infix status.

`$(SML_LIB)/basis/infixes.mlb`

The infix declarations of the Basis Library.

`$(SML_LIB)/basis/pervasive.mlb`

The entire top-level value and type environment of the Basis Library, with infix status. This is the same as importing the above six MLB files.

Last edited on 2006-06-02 15:32:30 by TomMurphy.

MLBasisExamples

Here are some example uses of ML Basis files.

Complete program

Suppose your complete program consists of the files `file1.sml`, ..., `filen.sml`, which depend upon libraries `lib1.mlb`, ..., `libm.mlb`.

```
(* import libraries *)
lib1.mlb
...
libm.mlb

(* program files *)
file1.sml
...
filen.sml
```

The bases denoted by `lib1.mlb`, ..., `libm.mlb` are merged (bindings of names in later bases take precedence over bindings of the same name in earlier bases), producing a basis in which `file1.sml`, ..., `filen.sml` are elaborated, adding additional bindings to the basis.

Export filter

Suppose you only want to export certain structures, signatures, and functors from a collection of files.

```
local
  file1.sml
  ...
  filen.sml
in
  (* export filter here *)
  functor F
  structure S
end
```

While `file1.sml`, ..., `filen.sml` may declare top-level identifiers in addition to `F` and `S`, such names are not accessible to programs and libraries that import this `.mlb`.

Export filter with renaming

Suppose you want an export filter, but want to rename one of the modules.

```
local
  file1.sml
  ...
  filen.sml
in
  (* export filter, with renaming, here *)
  functor F
  structure S' = S
end
```

Note that `functor F` is an abbreviation for `functor F = F`, which simply exports an identifier under the same name.

Import filter

Suppose you only want to import a functor `F` from one library and a structure `S` from another library.

```
local
  lib1.mlb
in
  (* import filter here *)
  functor F
end
local
  lib2.mlb
in
  (* import filter here *)
  structure S
end
file1.sml
...
filen.sml
```

Import filter with renaming

Suppose you want to import a structure `S` from one library and another structure `S` from another library.

```
local
  lib1.mlb
in
  (* import filter, with renaming, here *)
  structure S1 = S
end
local
  lib2.mlb
in
  (* import filter, with renaming, here *)
  structure S2 = S
end
file1.sml
...
filen.sml
```

Full Basis

Since the Modules level of SML is the natural means for organizing program and library components, MLB files provide convenient syntax for renaming Modules level identifiers (in fact, renaming of functor identifiers provides a mechanism that is not available in SML). However, please note that `.mlb` files elaborate to full bases including top-level types and values (including infix status), in addition to structures, signatures, and functors. For example, suppose you wished to extend the Basis Library with an `('a, 'b) either` datatype corresponding to a disjoint sum; the type and some operations should be available at the top-level; additionally, a signature and structure provide the complete interface.

We could use the following files.

either-sigs.sml

```
signature EITHER_GLOBAL =
  sig
    datatype ('a, 'b) either = Left of 'a | Right of 'b
    val &  : ('a -> 'c) * ('b -> 'c) -> ('a, 'b) either -> 'c
    val && : ('a -> 'c) * ('b -> 'd) -> ('a, 'b) either -> ('c, 'd) either
  end

signature EITHER =
  sig
    include EITHER_GLOBAL
    val isLeft  : ('a, 'b) either -> bool
    val isRight : ('a, 'b) either -> bool
    ...
  end
```

either-strs.sml

```
structure Either : EITHER =
  struct
    datatype ('a, 'b) either = Left of 'a | Right of 'b
    fun f & g = fn x =>
      case x of Left z => f z | Right z => g z
    fun f && g = (Left o f) & (Right o g)
    fun isLeft x = ((fn _ => true) & (fn _ => false)) x
    fun isRight x = (not o isLeft) x
    ...
  end
structure EitherGlobal : EITHER_GLOBAL = Either
```

either-infixes.sml

```
infixr 3 & &&
```

either-open.sml

```
open EitherGlobal
```

either.mlb

```
either-infixes.sml
local
  (* import Basis Library *)
  $(SML_LIB)/basis/basis.mlb
  either-sigs.sml
  either-strs.sml
in
  signature EITHER
  structure Either
  either-open.sml
end
```

A client that imports `either.mlb` will have access to neither `EITHER_GLOBAL` nor `EitherGlobal`, but will have access to the type `either` and the values `&` and `&&` (with infix status) in the top-level environment. Note that `either-infixes.sml` is outside the scope of the local, because we want the infixes available in the implementation of the library and to clients of the library.

Last edited on 2005-12-02 04:21:48 by StephenWeeks.

MLBasisPathMap

An ML Basis *path map* describes a map from ML Basis path variables (of the form `$(VAR)`) to file system paths. ML Basis path variables provide a flexible way to refer to libraries while allowing them to be moved without changing their clients.

The format of an `mlb-path-map` file is a sequence of lines; each line consists of two, white-space delimited tokens. The first token is a path variable `VAR` and the second token is the path to which the variable is mapped. The path may include path variables, which are recursively expanded.

The mapping from path variables to paths is initialized by reading a system-wide configuration file: `/usr/lib/mlton/mlb-path-map`. Additional path maps can be specified with `-mlb-path-map` (see CompileTimeOptions). Configuration files are processed from first to last and from top to bottom, later mappings take precedence over earlier mappings.

The compiler and system-wide configuration file makes the following path variables available.

MLB path variable	Description
`SML_LIB`	path to system-wide libaries, usually `/usr/lib/mlton/sml`
`TARGET_ARCH`	string representation of target architecture
`TARGET_OS`	string representation of target operating system

Last edited on 2006-04-23 21:06:47 by MatthewFluet.

MLBasisSyntaxAndSemantics

An ML Basis (MLB) file should have the `.mlb` suffix and should contain a basis declaration.

Syntax

A basis declaration must be one of the following forms.

- `basis` *basid* `=` *basexp* (`and` *basid* `=` *basexp*)*
- `open` $basid_1$ ... $basid_n$
- `local` *basdec* `in` *basdec* `end`
- *basdec* [`;`] *basdec*
- `structure` *strid* [`=` *strid*] (`and` *strid*[`=` *strid*])*
- `signature` *sigid* [`=` *sigid*] (`and` *sigid* [`=` *sigid*])*
- `functor` *funid* [`=` *funid*] (`and` *funid* [`=` *funid*])*
- *path*`.sml`, *path*`.sig`, or *path*`.fun`
- *path*`.mlb`
- `ann "`*ann*`" in` *basdec* `end`

A basis expression *basexp* must be of one the following forms.

- `bas` *basdec* `end`
- *basid*
- `let` *basdec* `in` *basexp* `end`

Nested SML-style comments (enclosed with `(*` and `*)`) are ignored (but LineDirectives are recognized).

Paths can be relative or absolute. Relative paths are relative to the directory containing the MLB file. Paths may include path variables and are expanded according to a path map. Unquoted paths may include alpha-numeric characters and the symbols "`-`" and "`_`", along with the arc separator "`/`" and extension separator "`.`". More complicated paths, including paths with spaces, may be included by quoting the path with `"`. A quoted path is lexed as a SML string constant.

Annotations allow a library author to control options that affect the elaboration of SML source files.

Semantics

There is a formal semantics for ML Basis files in the style of the Definition. Here, we give an informal explanation.

An SML structure is a collection of types, values, and other structures. Similarly, a basis is a collection, but of more kinds of objects: types, values, structures, fixities, signatures, functors, and other bases.

A basis declaration denotes a basis. A structure, signature, or functor declaration denotes a basis containing the corresponding module. Sequencing of basis declarations merges bases, with later definitions taking precedence over earlier ones, just like sequencing of SML declarations. Local declarations provide name hiding, just like SML local declarations. A reference to an SML source file causes the file to be elaborated in the basis extant at the point of reference. A reference to an MLB file causes the basis denoted by that MLB file to be imported -- the basis at the point of reference does *not* affect the imported basis.

Basis expressions and basis identifiers allow binding a basis to a name.

An MLB file is elaborated starting in an empty basis. Each MLB file is elaborated and evaluated only once, with the result being cached. Subsequent references use the cached value. Thus, any observable effects due to evaluation are not duplicated if the MLB file is referred to multiple times.

Last edited on 2005-12-01 20:54:18 by StephenWeeks.

MLKit

The ML Kit is a Standard ML Compiler.

Some Properties of the MLKit

- SML'97
    - including most of the latest Standard ML Basis Library
- Supports ML Basis Files
    - and ***separate compilation***
- Region-Based Memory Management
    - and garbage collection
- Two Backends
    - native x86 and bytecode

At the time of writing, MLKit does not support:

- concurrent programming / threads,
- calling from C to SML.

Last edited on 2006-02-04 11:17:46 by VesaKarvonen.

MLNLFFI

ML-NLFFI is the no-longer-foreign-function interface library for SML.

As of 20050212, MLton has an initial port of ML-NLFFI from SML/NJ to MLton. All of the ML-NLFFI functionality is present.

Additionally, MLton has an initial port of the `mlnlffigen` tool from SML/NJ to MLton. Due to low-level details, the code generated by SML/NJ's `ml-nlffigen` is not compatible with MLton, and vice-versa. However, the generated code has the same interface, so portable client code can be written. MLton's `mlnlffigen` does not currently support C functions with `struct` or `union` arguments.

Usage

- You can import the ML-NLFFI Library into an MLB file with `$(SML_LIB)/mlnlffi-lib/mlnlffi-lib.mlb`
- If you are porting a project from SML/NJ's CompilationManager to MLton's ML Basis system using `cm2mlb`, note that the following maps are included by default:

  ```
  $c/c.mlb  $(SML_LIB)/mlnlffi-lib/mlnlffi-lib.mlb
  ```

  This will automatically convert a `$/c.cm` import in an input `.cm` file into a `$(SML_LIB)/mlnlffi-lib/mlnlffi-lib.mlb` import in the output `.mlb` file.

Also see

- MLNLFFIImplementation

Last edited on 2005-12-01 20:57:29 by StephenWeeks.

MLNLFFIImplementation

MLton's implementation(s) of the MLNLFFI library differs from the SML/NJ implementation in two important ways:

- MLton cannot utilize the `Unsafe.cast` "cheat" described in Section 3.7 of Blume01. (MLton's representation of closures and aggressive representation optimizations make an `Unsafe.cast` even more "unsafe" than in other implementations.) We have considered two solutions:
  - One solution is to utilize an additional type parameter (as described in Section 3.7 of Blume01):

    ```
    signature C = sig
        type ('t, 'f, 'c) obj
        eqtype ('t, 'f, 'c) obj'
        ...
        type ('o, 'f) ptr
        eqtype ('o, 'f) ptr'
        ...
        type 'f fptr
        type 'f ptr'
        ...
        structure T : sig
            type ('t, 'f) typ
            ...
        end
    end
    ```

    The rule for `('t, 'f, 'c) obj`, `('t, 'f, 'c) ptr`, and also `('t, 'f) T.typ` is that whenever `F fptr` occurs within the instantiation of `'t`, then `'f` must be instantiated to `F`. In all other cases, `'f` will be instantiated to `unit`. (In the actual MLton implementation, an abstract type `naf` (not-a-function) is used instead of `unit`.)

    While this means that type-annotated programs may not type-check under both the SML/NJ implementation and the MLton implementation, this should not be a problem in practice. Tools, like `ml-nlffigen`, which are necessarily implementation dependent (in order to make calls through a C function pointer), may be easily extended to emit the additional type parameter. Client code which uses such generated glue-code (e.g., Section 1 of Blume01) need rarely write type-annotations, thanks to the magic of type inference.
  - The above implementation suffers from two disadvantages. First, it changes the MLNLFFI Library interface, meaning that the same program may not type-check under both the SML/NJ implementation and the MLton implementation (though, in light of type inference and the richer `MLRep` structure provided by MLton, this point is mostly moot).

    Second, it appears to unecessarily duplicate type information. For example, an external C variable of type `int (* f[3])(int)` (that is, an array of three function pointers), would be represented by the SML type `(((sint -> sint) fptr, dec dg3) arr, sint -> sint, rw) obj`. One might well ask why the `'f` instantiation (`sint -> sint` in this case) cannot be *extracted* from the `'t` instantiation (`((sint -> sint) fptr, dec dg3) arr` in this case), obviating the need for a separate *function-type* type argument. There are a number of components to an complete answer to this question. Foremost is the fact that Standard ML supports neither (general) type-level functions nor intensional polymorphism.

A more direct answer for MLNLFFI is that in the SML/NJ implemention, the definition of the types `('t, 'c) obj` and `('t, 'c) ptr` are made in such a way that the type variables `'t` and `'c` are phantom (not contributing to the run-time representation of an `('t, 'c) obj` or `('t, 'c) ptr` value), despite the fact that the types `((sint -> sint) fptr, rw) ptr` and `((double -> double) fptr, rw) ptr` necessarily carry distinct (and type incompatible) run-time (C-)type information (RTTI), corresponding to the different calling conventions of the two C functions. The `Unsafe.cast` "cheat" overcomes the type incompatibility without introducing a new type variable (as in the first solution above).

Hence, the reason that *function-type* type cannot be extracted from the `'t` type variable instantiation is that the type of the representation of RTTI doesn't even *see* the (phantom) `'t` type variable. The solution which presents itself is to give up on the phantomness of the `'t` type variable, making it available to the representation of RTTI.

This is not without some small drawbacks. Because many of the types used to instatiate `'t` carry more structure than is strictly necessary for `'t`'s RTTI, it is sometimes necessary to wrap and unwrap RTTI to accomodate the additional structure. (In the other implementations, the corresponding operations can pass along the RTTI unchanged.) However, these coercions contribute miniscule overhead; in fact, in a majority of cases, MLton's optimizations will completely eliminate the RTTI from the final program.

The implementation distributed with MLton uses the second solution.

Bonus question: Why can't one use a universal type to eliminate the use of `Unsafe.cast`?

- Answer: ???

- MLton (in both of the above implementations) provides a richer `MLRep` structure, utilizing `Int<N>` and `Word<N>` structures.

```
structure MLRep = struct
    structure Char =
       struct
          structure Signed = Int8
          structure Unsigned = Word8
          (* word-style bit-operations on integers... *)
          structure SignedBitops = IntBitOps(structure I = Signed
                                             structure W = Unsigned)
       end
    structure Short =
       struct
          structure Signed = Int16
          structure Unsigned = Word16
          (* word-style bit-operations on integers... *)
          structure SignedBitops = IntBitOps(structure I = Signed
                                             structure W = Unsigned)
       end
    structure Int =
       struct
          structure Signed = Int32
          structure Unsigned = Word32
          (* word-style bit-operations on integers... *)
          structure SignedBitops = IntBitOps(structure I = Signed
                                             structure W = Unsigned)
       end
    structure Long =
```

```
      struct
         structure Signed = Int32
         structure Unsigned = Word32
         (* word-style bit-operations on integers... *)
         structure SignedBitops = IntBitOps(structure I = Signed
                                            structure W = Unsigned)
      end
   structure LongLong =
      struct
         structure Signed = Int64
         structure Unsigned = Word64
         (* word-style bit-operations on integers... *)
         structure SignedBitops = IntBitOps(structure I = Signed
                                            structure W = Unsigned)
      end
   structure Float = Real32
   structure Double = Real64
end
```

This would appear to be a better interface, even when an implementation must choose `Int32` and `Word32` as the representation for smaller C-types.

Last edited on 2005-12-02 13:11:01 by MatthewFluet.

MLRISCLibrary

The MLRISC Library is a framework for retargetable and optimizing compiler back ends. The MLRISC Library is distributed with SML/NJ. Due to differences between SML/NJ and MLton, this library will not work out-of-the box with MLton.

As of 20060304, MLton includes a port of the MLRISC Library synchronized with SML/NJ version 110.58.

Usage

- You can import a sub-library of the MLRISC Library into an MLB file with:

MLB file	Description
`$(SML_LIB)/mlrisc-lib/mlb/ALPHA.mlb`	The ALPHA backend
`$(SML_LIB)/mlrisc-lib/mlb/Control.mlb`	
`$(SML_LIB)/mlrisc-lib/mlb/Graphs.mlb`	
`$(SML_LIB)/mlrisc-lib/mlb/HPPA.mlb`	The HPPA backend
`$(SML_LIB)/mlrisc-lib/mlb/IA32.mlb`	The IA32 backend
`$(SML_LIB)/mlrisc-lib/mlb/Lib.mlb`	
`$(SML_LIB)/mlrisc-lib/mlb/MLRISC.mlb`	
`$(SML_LIB)/mlrisc-lib/mlb/MLTREE.mlb`	
`$(SML_LIB)/mlrisc-lib/mlb/PPC.mlb`	The PPC backend
`$(SML_LIB)/mlrisc-lib/mlb/RA.mlb`	
`$(SML_LIB)/mlrisc-lib/mlb/SPARC.mlb`	The Sparc backend
`$(SML_LIB)/mlrisc-lib/mlb/Visual.mlb`	

- If you are porting a project from SML/NJ's CompilationManager to MLton's ML Basis system using `cm2mlb`, note that the following map is included by default:

```
$SMLNJ-MLRISC   $(SML_LIB)/mlrisc-lib/mlb
```

  This will automatically convert a `$SMLNJ-MLRISC/MLRISC.cm` import in an input `.cm` file into a `$(SML_LIB)/mlrisc-lib/mlb/MLRISC.mlb` import in the output `.mlb` file.

Details

The following changes were made to the MLRISC Library, in addition to deriving the `.mlb` file from the `.cm` files:

- eliminate or-patterns: Duplicate the whole match (`p => e`) at each of the patterns.
- eliminate vector constants: Change `#[` to `Vector.fromList [`.
- eliminate `withtype` in signatures
- eliminate sequential `withtype` expansions: Most could be rewritten as a sequence of type definitions and datatype definitions.
- eliminate higher-order functors: Every higher-order functor definition and application could be uncurried in the obvious way.
- eliminate where <str> = <str>: Quite painful to expand out all the flexible types in the respective structures. Furthermore, many of the implied type equalities aren't needed, but it's too hard

to pick out the right ones.

Patch

* MLRISC.patch

Last edited on 2006-03-04 17:07:19 by MatthewFluet.

MLj

MLj is a Standard ML Compiler that targets Java bytecode. It is no longer maintained. It has morphed into SML.NET.

BentonEtAl98 and BentonKennedy99 describe MLj.

Last edited on 2004-12-30 20:11:59 by StephenWeeks.

MLmon

An `mlmon.out` file records dynamic profiling counts.

File format

An `mlmon.out` file is a text file with a sequence of lines.

- The string "`MLton prof`".
- The string "`alloc`", "`count`", or "`time`", depending on the kind of profiling information, corresponding to the command-line argument supplied to `mlton -profile`.
- The string "`current`" or "`stack`" depending on whether profiling data was gathered for only the current function (the top of the stack) or for all functions on the stack. This corresponds to whether the executable was compiled with `-profile-stack false` or `-profile-stack true`.
- The magic number of the executable.
- The number of non-gc ticks, followed by a space, then the number of GC ticks.
- The number of (split) functions for which data is recorded.
- A line for each (split) function with counts. Each line contains an integer count of the number of ticks while the function was current. In addition, if stack data was gathered (`-profile-stack true`), then the line contains two additional tick counts:
    - the number of ticks while the function was on the stack.
    - the number of ticks while the function was on the stack and a GC was performed.
- The number of (master) functions for which data is recorded.
- A line for each (master) function with counts. The lines have the same format and meaning as with split-function counts.

Last edited on 2006-10-23 22:02:16 by StephenWeeks.

MLtonArray

```
signature MLTON_ARRAY =
   sig
      val unfoldi: int * 'b * (int * 'b -> 'a * 'b) -> 'a array
   end
```

- `unfoldi (n, b, f)`

constructs an array *a* of length `n`, whose elements a_i are determined by the equations $b_0 = b$ and $(a_i, b_{i+1}) = f(i, b_i)$.

Last edited on 2005-12-01 22:27:14 by StephenWeeks.

MLtonBinIO

```
signature MLTON_BIN_IO = MLTON_IO
```

See MLtonIO.

Last edited on 2005-12-01 21:00:20 by StephenWeeks.

MLtonCont

```
signature MLTON_CONT =
   sig
      type 'a t

      val callcc: ('a t -> 'a) -> 'a
      val prepend: 'a t * ('b -> 'a) -> 'b t
      val throw: 'a t * 'a -> 'b
      val throw': 'a t * (unit -> 'a) -> 'b
   end
```

- `type 'a t`
  the type of continuations that expect a value of type `'a`.
- `callcc f`
  applies `f` to the current continuation. This copies the entire stack; hence, `callcc` takes time proportional to the current stack size.
- `prepend (k, f)`
  composes a function `f` with a continuation `k` to create a continuation that first does `f` and then does `k`. This is a constant time operation.
- `throw (k, v)`
  throws value `v` to continuation `k`. This copies the entire stack of `k`; hence, `throw` takes time proportional to the size of this stack.
- `throw' (k, th)`

a generalization of throw that evaluates `th ()` in the context of `k`. Thus, for example, if `th ()` raises an exception or grabs another continuation, it will see `k`, not the current continuation.

Last edited on 2005-12-01 22:27:22 by StephenWeeks.

MLtonExn

```
signature MLTON_EXN =
   sig
      val addExnMessager: (exn -> string option) -> unit
      val history: exn -> string list
      val topLevelHandler: exn -> 'a
   end
```

- `addExnMessager f`
  adds `f` as a pretty-printer to be used by `General.exnMessage` for converting exceptions to strings. Messagers are tried in order from most recently added to least recently added.
- `history e`
  returns call stack at the point that `e` was first raised. Each element of the list is a file position. The elements are in reverse chronological order, i.e. the function called last is at the front of the list.

  `history e` will return `[]` unless the program is compiled with `-const 'Exn.keepHistory true'`.

- `topLevelHandler e`

behaves as if the top level handler received the exception `e`, that is, print out the unhandled exception message for `e` and exit.

Last edited on 2005-12-02 04:22:01 by StephenWeeks.

MLtonFinalizable

```
signature MLTON_FINALIZABLE =
   sig
      type 'a t

      val addFinalizer: 'a t * ('a -> unit) -> unit
      val finalizeBefore: 'a t * 'b t -> unit
      val new: 'a -> 'a t
      val touch: 'a t -> unit
      val withValue: 'a t * ('a -> 'b) -> 'b
   end
```

A *finalizable* value is a container to which finalizers can be attached. A container holds a value, which is reachable as long as the container itself is reachable. A *finalizer* is a function that runs at some point after garbage collection determines that the container to which it is attached has become unreachable. A finalizer is treated like a signal handler, in that it runs asynchronously in a separate thread, with signals blocked, and will not interrupt a critical section (see MLtonThread).

- `addFinalizer (v, f)`
  adds `f` as a finalizer to `v`. This means that sometime after the last call to `withValue` on `v` completes and `v` becomes unreachable, `f` will be called with the value of `v`.
- `finalizeBefore (v1, v2)`
  ensures that `v1` will be finalized before `v2`. A cycle of values `v` = `v1`, ..., `vn` = `v` with `finalizeBefore (vi, vi+1)` will result in none of the `vi` being finalized.
- `new x`
  creates a new finalizable value, `v`, with value `x`. The finalizers of `v` will run sometime after the last call to `withValue` on `v` when the garbage collector determines that `v` is unreachable.
- `touch v`
  ensures that `v`'s finalizers will not run before the call to `touch`.
- `withValue (v, f)`

returns the result of applying `f` to the value of `v` and ensures that `v`'s finalizers will not run before `f` completes. The call to `f` is a nontail call.

Example

Suppose that `finalizable.sml` contains the following.

```
signature CLIST =
   sig
      type t

      val cons: int * t -> t
      val sing: int -> t
      val sum: t -> int
   end

functor CList (structure F: MLTON_FINALIZABLE
               structure Prim:
                  sig
                     val cons: int * Word32.word -> Word32.word
                     val free: Word32.word -> unit
                     val sing: int -> Word32.word
```

```
                  val sum: Word32.word -> int
               end): CLIST =
   struct
      type t = Word32.word F.t

      fun cons (n: int, l: t) =
         F.withValue
         (l, fn w' =>
          let
             val c = F.new (Prim.cons (n, w'))
             val _ = F.addFinalizer (c, Prim.free)
             val _ = F.finalizeBefore (c, l)
          in
             c
          end)

      fun sing n =
         let
            val c = F.new (Prim.sing n)
            val _ = F.addFinalizer (c, Prim.free)
         in
            c
         end

      fun sum c = F.withValue (c, Prim.sum)
   end

functor Test (structure CList: CLIST
              structure MLton: sig
                                  structure GC:
                                     sig
                                        val collect: unit -> unit
                                     end
                               end) =
   struct
      fun f n =
         if n = 1
            then ()
         else
            let
               val a = Array.tabulate (n, fn i => i)
               val _ = Array.sub (a, 0) + Array.sub (a, 1)
            in
               f (n - 1)
            end

      val l = CList.sing 2
      val l = CList.cons (2,l)
      val l = CList.cons (2,l)
      val l = CList.cons (2,l)
      val l = CList.cons (2,l)
      val l = CList.cons (2,l)
      val l = CList.cons (2,l)
      val _ = MLton.GC.collect ()
      val _ = f 100
      val _ = print (concat ["listSum(l) = ",
                             Int.toString (CList.sum l),
                             "\n"])
      val _ = MLton.GC.collect ()
      val _ = f 100
   end
```

```
structure CList =
   CList (structure F = MLton.Finalizable
          structure Prim =
             struct
                val cons = _import "listCons": int * Word32.word -> Word32.word;
                val free = _import "listFree": Word32.word -> unit;
                val sing = _import "listSing": int -> Word32.word;
                val sum = _import "listSum": Word32.word -> int;
             end)

structure S = Test (structure CList = CList
                    structure MLton = MLton)
```

Suppose that `cons.c` contains the following.

```
#include <stdio.h>

typedef unsigned int uint;

typedef struct Cons {
        struct Cons *next;
        int value;
} *Cons;

Cons listCons (int n, Cons c) {
        Cons res;

        res = (Cons) malloc (sizeof(*res));
        fprintf (stderr, "0x%08x = listCons (%d)\n", (uint)res, n);
        res->next = c;
        res->value = n;
        return res;
}

Cons listSing (int n) {
        Cons res;

        res = (Cons) malloc (sizeof(*res));
        fprintf (stderr, "0x%08x = listSing (%d)\n", (uint)res, n);
        res->next = NULL;
        res->value = n;
        return res;
}

void listFree (Cons p) {
        fprintf (stderr, "listFree (0x%08x)\n", (uint)p);
        free (p);
}

int listSum (Cons c) {
        int res;

        fprintf (stderr, "listSum\n");
        res = 0;
        for (; c != NULL; c = c->next)
                res += c->value;
        return res;
}
```

We can compile these to create an executable with

```
% mlton -default-ann 'allowFFI true' finalizable.sml cons.c
```

Running this executable will create output like the following.

```
% finalizable
0x08072890 = listSing (2)
0x080728a0 = listCons (2)
0x080728b0 = listCons (2)
0x080728c0 = listCons (2)
0x080728d0 = listCons (2)
0x080728e0 = listCons (2)
0x080728f0 = listCons (2)
listSum
listSum(l) = 14
listFree (0x080728f0)
listFree (0x080728e0)
listFree (0x080728d0)
listFree (0x080728c0)
listFree (0x080728b0)
listFree (0x080728a0)
listFree (0x08072890)
```

Synchronous Finalizers

Finalizers in MLton are asynchronous. That is, they run at an unspecified time, interrupting the user program. It is also possible, and sometimes useful, to have synchronous finalizers, where the user program explicitly decides when to run enabled finalizers. We have considered this in MLton, and it seems possible, but there are some unresolved design issues. See the thread at

- http://mlton.org/pipermail/mlton/2004-September/016570.html

Also see

- Boehm03

Last edited on 2006-08-21 19:41:33 by StephenWeeks.

MLtonGC

```
signature MLTON_GC =
   sig
      val collect: unit -> unit
      val pack: unit -> unit
      val setMessages: bool -> unit
      val setSummary: bool -> unit
      val unpack: unit -> unit
   end
```

- `collect ()`
  causes a garbage collection to occur.
- `pack ()`
  shrinks the heap as much as possible so that other processes can use available RAM.
- `setMessages b`
  controls whether diagnostic messages are printed at the beginning and end of each garbage collection. It is the same as the `gc-messages` runtime system option.
- `setSummary b`
  controls whether a summary of garbage collection statistics is printed upon termination of the program. It is the same as the `gc-summary` runtime system option.
- `unpack ()`

resizes a packed heap to the size desired by the runtime.

Last edited on 2004-11-02 04:24:34 by StephenWeeks.

MLtonIO

```
signature MLTON_IO =
   sig
      type instream
      type outstream

      val inFd: instream -> Posix.IO.file_desc
      val mkstemp: string -> string * outstream
      val mkstemps: {prefix: string, suffix: string} -> string * outstream
      val newIn: Posix.IO.file_desc * string -> instream
      val newOut: Posix.IO.file_desc * string -> outstream
      val outFd: outstream -> Posix.IO.file_desc
   end
```

- `inFd ins`
  returns the file descriptor corresponding to `ins`.
- `mkstemp s`
  like the C `mkstemp` function, generates and open a temporary file with prefix `s`.
- `mkstemps {prefix, suffix}`
  like `mkstemp`, except it has both a prefix and suffix.
- `newIn (fd, name)`
  creates a new instream from file descriptor `fd`, with `name` used in any `Io` exceptions later raised.
- `newOut (fd, name)`
  creates a new outstream from file descriptor `fd`, with `name` used in any `Io` exceptions later raised.
- `outFd out`

returns the file descriptor corresponding to `out`.

Last edited on 2005-12-01 22:27:48 by StephenWeeks.

MLtonIntInf

```
signature MLTON_INT_INF =
   sig
      type t

      val areSmall: t * t -> bool
      val gcd: t * t -> t
      val isSmall: t -> bool
      datatype rep =
         Big of word vector
       | Small of int
      val rep: t -> rep
   end
```

MLton represents an arbitrary precision integer either as an unboxed 32 bit word with the bottom bit set to 1 and the top 31 bits representing a small integer in $[-2^{30}, 2^{30})$, or as a pointer to a vector of words where the first word indicates the sign and the rest are the limbs of GnuMP big integer.

- `type t`
  the same as type `IntInf.int`.
- `areSmall (a, b)`
  returns true iff both `a` and `b` are small.
- `gcd (a, b)`
  uses the GnuMP's fast gcd implementation.
- `isSmall a`
  returns true iff `a` is small.
- `datatype rep`
  the underlying representation of an `IntInf.int`.
- `rep i`

returns the underlying representation of `i`.

Last edited on 2005-12-02 03:46:17 by MatthewFluet.

MLtonItimer

```
signature MLTON_ITIMER =
   sig
      datatype t =
         Prof
       | Real
       | Virtual

      val set: t * {interval: Time.time, value: Time.time} -> unit
      val signal: t -> Posix.Signal.signal
   end
```

- `set (t, {interval, value})`
  sets the interval timer (using `setitimer`) specified by `t` to the given `interval` and `value`.
- `signal t`

returns the signal corresponding to `t`.

Last edited on 2005-12-01 22:27:07 by StephenWeeks.

MLtonPlatform

```
signature MLTON_PLATFORM =
   sig
      structure Arch:
         sig
            datatype t = Alpha | AMD64 | ARM | HPPA | IA64 | m68k
                       | MIPS | PowerPC | S390 | Sparc | X86

            val fromString: string -> t option
            val host: t
            val toString: t -> string
         end

      structure OS:
         sig
            datatype t = Cygwin | Darwin | FreeBSD | Linux
                       | MinGW | NetBSD | OpenBSD | Solaris

            val fromString: string -> t option
            val host: t
            val toString: t -> string
         end
   end
```

- `datatype Arch.t`
  processor architectures
- `Arch.fromString a`
  converts from string to architecture. Case insensitive.
- `Arch.host`
  the architecture for which the program is compiled.
- `Arch.toString`
  string for architecture.
- `datatype OS.t`
  operating systems
- `OS.fromString`
  converts from string to operating system. Case insensitive.
- `OS.host`
  the operating system for which the program is compiled.
- `OS.toString`

string for operating system.

Last edited on 2005-12-01 22:27:55 by StephenWeeks.

MLtonPointer

```
signature MLTON_POINTER =
   sig
      eqtype t

      val add: t * word -> t
      val compare: t * t -> order
      val diff: t * t -> word
      val getInt8: t * int -> Int8.int
      val getInt16: t * int -> Int16.int
      val getInt32: t * int -> Int32.int
      val getInt64: t * int -> Int64.int
      val getPointer: t * int -> t
      val getReal32: t * int -> Real32.real
      val getReal64: t * int -> Real64.real
      val getWord8: t * int -> Word8.word
      val getWord16: t * int -> Word16.word
      val getWord32: t * int -> Word32.word
      val getWord64: t * int -> Word64.word
      val null: t
      val setInt8: t * int * Int8.int -> unit
      val setInt16: t * int * Int16.int -> unit
      val setInt32: t * int * Int32.int -> unit
      val setInt64: t * int * Int64.int -> unit
      val setPointer: t * int * t -> unit
      val setReal32: t * int * Real32.real -> unit
      val setReal64: t * int * Real64.real -> unit
      val setWord8: t * int * Word8.word -> unit
      val setWord16: t * int * Word16.word -> unit
      val setWord32: t * int * Word32.word -> unit
      val setWord64: t * int * Word64.word -> unit
      val sub: t * word -> t
   end
```

- `eqtype t`
  the type of pointers, i.e. machine addresses.
- `add (p, w)`
  returns the pointer `w` bytes after than `p`. Does not check for overflow.
- `compare (p1, p2)`
  compares the pointer `p1` to the pointer `p2` (as addresses).
- `diff (p1, p2)`
  returns the number of bytes `w` such that `add (p2, w) = p1`. Does not check for overflow.
- `get`*X* `(p, i)`
  returns the object stored at index i of the array of *X* objects pointed to by `p`. For example, `getWord32 (p, 7)` returns the 32-bit word stored 28 bytes beyond `p`.
- `null`
  the null pointer, i.e. 0.
- `set`*X* `(p, i, v)`
  assigns `v` to the object stored at index i of the array of *X* objects pointed to by `p`. For example, `setWord32 (p, 7, w)` stores the 32-bit word `w` at the address 28 bytes beyond `p`.
- `sub (p, w)`

returns the pointer `w` bytes before `p`. Does not check for overflow.

Last edited on 2006-02-03 15:59:41 by MatthewFluet.

MLtonProcEnv

```
signature MLTON_PROC_ENV =
   sig
      type gid

      val setenv: {name: string, value: string} -> unit
      val setgroups: gid list -> unit
  end
```

- `setenv {name, value}`
  like the C `setenv` function. Does not require `name` or `value` to be null terminated.
- `setgroups grps`

like the C `setgroups` function.

Last edited on 2005-12-01 22:28:03 by StephenWeeks.

MLtonProcess

```
signature MLTON_PROCESS =
   sig
      type pid

      val spawn: {args: string list, path: string} -> pid
      val spawne: {args: string list, env: string list, path: string} -> pid
      val spawnp: {args: string list, file: string} -> pid

      structure Child:
        sig
          type ('use, 'dir) t

          val binIn: (BinIO.instream, input) t -> BinIO.instream
          val binOut: (BinIO.outstream, output) t -> BinIO.outstream
          val fd: (Posix.FileSys.file_desc, 'dir) t -> Posix.FileSys.file_desc
          val remember: (any, 'dir) t -> ('use, 'dir) t
          val textIn: (TextIO.instream, input) t -> TextIO.instream
          val textOut: (TextIO.outstream, output) t -> TextIO.outstream
        end

      structure Param:
        sig
          type ('use, 'dir) t

          val child: (chain, 'dir) Child.t -> (none, 'dir) t
          val fd: Posix.FileSys.file_desc -> (none, 'dir) t
          val file: string -> (none, 'dir) t
          val forget: ('use, 'dir) t -> (any, 'dir) t
          val null: (none, 'dir) t
          val pipe: ('use, 'dir) t
          val self: (none, 'dir) t
        end

      type ('stdin, 'stdout, 'stderr) t
      type any
      type chain
      type input
      type none
      type output

      exception MisuseOfForget
      exception DoublyRedirected

      val create:
         {args: string list,
          env: string list option,
          path: string,
          stderr: ('stderr, output) Param.t,
          stdin: ('stdin, input) Param.t,
          stdout: ('stdout, output) Param.t}
         -> ('stdin, 'stdout, 'stderr) t
      val getStderr: ('stdin, 'stdout, 'stderr) t -> ('stderr, input) Child.t
      val getStdin:  ('stdin, 'stdout, 'stderr) t -> ('stdin, output) Child.t
      val getStdout: ('stdin, 'stdout, 'stderr) t -> ('stdout, input) Child.t
      val kill: ('stdin, 'stdout, 'stderr) t * Posix.Signal.signal -> unit
      val reap: ('stdin, 'stdout, 'stderr) t -> Posix.Process.exit_status
   end
```

Spawn

The `spawn` functions provide an alternative to the `fork`/`exec` idiom that is typically used to create a new process. On most platforms, the `spawn` functions are simple wrappers around `fork`/`exec`. However, under Windows, the `spawn` functions are primitive. All `spawn` functions return the process id of the spawned process. They differ in how the executable is found and the environment that it uses.

- `spawn {args, path}`
  starts a new process running the executable specified by `path` with the arguments `args`. Like `Posix.Process.exec`.
- `spawne {args, env, path}`
  starts a new process running the executable specified by `path` with the arguments `args` and environment `env`. Like `Posix.Process.exece`.
- `spawnp {args, file}`

search the `PATH` environment variable for an executable named `file`, and start a new process running that executable with the arguments `args`. Like `Posix.Process.execp`.

Create

`MLton.Process.create` provides functionality similar to `Unix.executeInEnv`, but provides more control control over the input, output, and error streams. In addition, `create` works on all platforms, including Cygwin and MinGW (Windows) where `Posix.fork` is unavailable. For greatest portability programs should still use the standard `Unix.execute`, `Unix.executeInEnv`, and `OS.Process.system`.

The following types and sub-structures are used by the `create` function. They provide static type checking of correct stream usage.

Child

- `('use, 'dir) Child.t`
  This represents a handle to one of a child's standard streams. The `'dir` is viewed with respect to the parent. Thus a `('a, input) Child.t` handle means that the parent may input the output from the child.
- `Child.{bin,text}{In,Out} h`
  These functions take a handle and bind it to a stream of the named type. The type system will detect attempts to reverse the direction of a stream or to use the same stream in multiple, incompatible ways.
- `Child.fd h`
  This function behaves like the other `Child.*` functions; it opens a stream. However, it does not enforce that you read or write from the handle. If you use the descriptor in an inappropriate direction, the behavior is undefined. Furthermore, this function may potentially be unavailable on future MLton host platforms.
- `Child.remember h`

This function takes a stream of use `any` and resets the use of the stream so that the stream may be used by `Child.*`. An `any` stream may have had use `none` or `'use` prior to calling `Param.forget`. If the stream was `none` and is used, `MisuseOfForget` is raised.

Param

- `('use, 'dir) Param.t`
  This is a handle to an input/output source and will be passed to the created child process. The `'dir` is relative to the child process. Input means that the child process will read from this stream.
- `Param.child h`
  Connect the stream of the new child process to the stream of a previously created child process. A single child stream should be connected to only one child process or else `DoublyRedirected` will be raised.
- `Param.fd fd`
  This creates a stream from the provided file descriptor which will be closed when `create` is called. This function may not be available on future MLton host platforms.
- `Param.forget h`
  This hides the type of the actual parameter as `any`. This is useful if you are implementing an application which conditionally attaches the child process to files or pipes. However, you must ensure that your use after `Child.remember` matches the original type.
- `Param.file s`
  Open the given file and connect it to the child process. Note that the file will be opened only when `create` is called. So any exceptions will be raised there and not by this function. If used for `input`, the file is opened read-only. If used for `output`, the file is opened read-write.
- `Param.null`
  In some situations, the child process should have its output discarded. The `null` param when passed as `stdout` or `stderr` does this. When used for `stdin`, the child process will either receive `EOF` or a failure condition if it attempts to read from `stdin`.
- `Param.pipe`

This will connect the input/output of the child process to a pipe which the parent process holds. This may later form the input to one of the `Child.*` functions and/or the `Param.child` function.

Process

- `type ('stdin, 'stdout, 'stderr) t`
  represents a handle to a child process. The type arguments capture how the named stream of the child process may be used.
- `type any`
  bypasses the type system in situations where an application does not want the it to enforce correct usage. See `Child.remember` and `Param.forget`.
- `type chain`
  means that the child process's stream was connected via a pipe to the parent process. The parent process may pass this pipe in turn to another child, thus chaining them together.
- `type input, output`
  record the direction that a stream flows. They are used as a part of `Param.t` and `Child.t` and is detailed there.
- `type none`

means that the child process's stream my not be used by the parent process. This happens when the child process is connected directly to some source.

The types `BinIO.instream`, `BinIO.outstream`, `TextIO.instream`, `TextIO.outstream`, and `Posix.FileSys.file_desc` are also valid types with which to instantiate child streams.

- `exception MisuseOfForget`
  may be raised if `Child.remember` and `Param.forget` are used to bypass the normal type checking. This exception will only be raised in cases where the `forget` mechanism allows a misuse that would be impossible with the type-safe versions.
- `exception DoublyRedirected`
  raised if a stream connected to a child process is redirected to two separate child processes. It is safe, though bad style, to use the a `Child.t` with the same `Child.*` function repeatedly.
- `create {args, path, env, stderr, stdin, stdout}`
  starts a child process with the given command-line `args` (excluding the program name). `path` should be an absolute path to the executable run in the new child process; relative paths work, but are less robust. Optionally, the environment may be overridden with `env` where each string element has the form `"key=value"`. The `std*` options must be provided by the `Param.*` functions documented above.

  Processes which are `create`d must be either `reap`ed or `kill`ed.

- `getStd{in,out,err} proc`
  gets a handle to the specified stream. These should be used by the `Child.*` functions. Failure to use a stream connected via pipe to a child process may result in runtime dead-lock and elicits a compiler warning.
- `kill (proc, sig)`
  terminates the child process immediately. The signal may or may not mean anything depending on the host platform. A good value is `Posix.Signal.term`.
- `reap proc`

waits for the child process to terminate and return its exit status.

Important usage notes

When building an application with many pipes between child processes, it is important to ensure that there are no cycles in the undirected pipe graph. If this property is not maintained, deadlocks are a very serious potential bug which may only appear under difficult to reproduce conditions.

The danger lies in that most operating systems implement pipes with a fixed buffer size. If process A has two output pipes which process B reads, it can happen that process A blocks writing to pipe 2 because it is full while process B blocks reading from pipe 1 because it is empty. This same situation can happen with any undirected cycle formed between processes (vertexes) and pipes (undirected edges) in the graph.

It is possible to make this safe using low-level I/O primitives for polling. However, these primitives are not very portable and difficult to use properly. A far better approach is to make sure you never create a cycle in the first place.

For these reasons, the `Unix.executeInEnv` is a very dangerous function. Be careful when using it to ensure that the child process only operates on either `stdin` or `stdout`, but not both.

Example use of MLton.Process.create

The following example program launches the `ipconfig` utility, pipes its output through `grep`, and then reads the result back into the program.

```
open MLton.Process
val p =
        create {args = [ "/all" ],
                env = NONE,
                path = "C:\\WINDOWS\\system32\\ipconfig.exe",
                stderr = Param.self,
                stdin = Param.null,
                stdout = Param.pipe}
val q =
        create {args = [ "IP-Ad" ],
                env = NONE,
                path = "C:\\msys\\bin\\grep.exe",
                stderr = Param.self,
                stdin = Param.child (getStdout p),
                stdout = Param.pipe}
fun suck h =
        case TextIO.inputLine h of
                NONE => ()
                | SOME s => (print ("'" ^ s ^ "'\n"); suck h)

val () = suck (Child.textIn (getStdout q))
```

Last edited on 2005-12-02 04:22:19 by StephenWeeks.

MLtonProfile

```
signature MLTON_PROFILE =
   sig
      structure Data:
         sig
            type t

            val equals: t * t -> bool
            val free: t -> unit
            val malloc: unit -> t
            val write: t * string -> unit
         end

      val isOn: bool
      val withData: Data.t * (unit -> 'a) -> 'a
   end
```

`MLton.Profile` provides Profiling control from within the program, allowing you to profile individual portions of your program. With `MLton.Profile`, you can create many units of profiling data (essentially, mappings from functions to counts) during a run of a program, switch between them while the program is running, and output multiple `mlmon.out` files.

- `isOn`
  a compile-time constant that is false only when compiling `-profile no`.
- `type Data.t`
  the type of a unit of profiling data. In order to most efficiently execute non-profiled programs, when compiling `-profile no` (the default), `Data.t` is equivalent to `unit ref`.
- `Data.equals (x, y)`
  returns true if the `x` and `y` are the same unit of profiling data.
- `Data.free x`
  frees the memory associated with the unit of profiling data `x`. It is an error to free the current unit of profiling data or to free a previously freed unit of profiling data. When compiling `-profile no`, `Data.free x` is a no-op.
- `Data.malloc ()`
  returns a new unit of profiling data. Each unit of profiling data is allocated from the process address space (but is *not* in the MLton heap) and consumes memory proportional to the number of source functions. When compiling `-profile no`, `Data.malloc ()` is equivalent to allocating a new `unit ref`.
- `write (x, f)`
  writes the accumulated ticks in the unit of profiling data `x` to file `f`. It is an error to write a previously freed unit of profiling data. When compiling `-profile no`, `write (x, f)` is a no-op. A profiled program will always write the current unit of profiling data at program exit to a file named `mlmon.out`.
- `withData (d, f)`

runs `f` with `d` as the unit of profiling data, and returns the result of `f` after restoring the current unit of profiling data. When compiling `-profile no`, `withData (d, f)` is equivalent to `f ()`.

Example

Here is an example, taken from the `examples/profiling` directory, showing how to profile the executions of the `fib` and `tak` functions separately. Suppose that `fib-tak.sml` contains the following.

```
structure Profile = MLton.Profile

val fibData = Profile.Data.malloc ()
val takData = Profile.Data.malloc ()

fun wrap (f, d) x =
   Profile.withData (d, fn () => f x)

val rec fib =
   fn 0 => 0
    | 1 => 1
    | n => fib (n - 1) + fib (n - 2)
val fib = wrap (fib, fibData)

fun tak (x,y,z) =
   if not (y < x)
      then z
   else tak (tak (x - 1, y, z),
             tak (y - 1, z, x),
             tak (z - 1, x, y))
val tak = wrap (tak, takData)

val rec f =
   fn 0 => ()
    | n => (fib 38; f (n-1))
val _ = f 2

val rec g =
   fn 0 => ()
    | n => (tak (18,12,6); g (n-1))
val _ = g 500

fun done (data, file) =
   (Profile.Data.write (data, file)
    ; Profile.Data.free data)

val _ = done (fibData, "mlmon.fib.out")
val _ = done (takData, "mlmon.tak.out")
```

Compile and run the program.

```
% mlton -profile time fib-tak.sml
% ./fib-tak
```

Separately display the profiling data for `fib`

```
% mlprof fib-tak mlmon.fib.out
5.77 seconds of CPU time (0.00 seconds GC)
function   cur
--------- -----
fib       96.9%
<unknown>  3.1%
```

and for `tak`

```
% mlprof fib-tak mlmon.tak.out
0.68 seconds of CPU time (0.00 seconds GC)
function  cur
-------- ------
tak      100.0%
```

Combine the data for `fib` and `tak` by calling `mlprof` with multiple `mlmon.out` files.

```
% mlprof fib-tak mlmon.fib.out mlmon.tak.out mlmon.out
6.45 seconds of CPU time (0.00 seconds GC)
function   cur
--------- -----
fib       86.7%
tak       10.5%
<unknown>  2.8%
```

Last edited on 2005-12-01 22:21:31 by StephenWeeks.

MLtonRandom

```
signature MLTON_RANDOM =
   sig
      val alphaNumChar: unit -> char
      val alphaNumString: int -> string
      val rand: unit -> word
      val seed: unit -> word option
      val srand: word -> unit
      val useed: unit -> word option
   end
```

- `alphaNumChar ()`
  returns a random alphanumeric character.
- `alphaNumString n`
  returns a string of length `n` of random alphanumeric characters.
- `rand ()`
  returns the next pseudo-random number.
- `seed ()`
  returns a random word from `/dev/random`. Useful as an arg to `srand`. If `/dev/random` can not be read from, `seed ()` returns `NONE`. A call to `seed` may block until enough random bits are available.
- `srand w`
  sets the seed used by `rand` to `w`.
- `useed ()`

returns a random word from `/dev/urandom`. Useful as an arg to `srand`. If `/dev/urandom` can not be read from, `useed ()` returns `NONE`. A call to `useed` will never block -- it will instead return lower quality random bits.

Last edited on 2005-12-02 04:22:31 by StephenWeeks.

MLtonRlimit

```
signature MLTON_RLIMIT =
   sig
      type rlim = word
      type t

      val coreFileSize: t         (* CORE    max core file size *)
      val cpuTime: t              (* CPU     CPU time in seconds *)
      val dataSize: t             (* DATA    max data size *)
      val fileSize: t             (* FSIZE   Maximum filesize *)
      val get: t -> {hard: rlim, soft: rlim}
      val infinity: rlim
      val lockedInMemorySize: t   (* MEMLOCK max locked address space *)
      val numFiles: t             (* NOFILE  max number of open files *)
      val numProcesses: t         (* NPROC   max number of processes *)
      val residentSetSize: t      (* RSS     max resident set size *)
      val set: t * {hard: rlim, soft: rlim} -> unit
      val stackSize: t            (* STACK   max stack size *)
      val virtualMemorySize: t    (* AS      virtual memory limit *)
   end
```

`MLton.Rlimit` provides a wrapper around the C `getrlimit` and `setrlimit` functions.

- `type rlim`
  the type of resource limits.
- `type t`
  the types of resources that can be inspected and modified.
- `get r`
  returns the current hard and soft limits for resource `r`. May raise `OS.SysErr`.
- `infinity`
  indicates that a resource is unlimited.
- `set (r, {hard, soft})`

sets the hard and soft limits for resource `r`. May raise `OS.SysErr`.

Last edited on 2005-12-01 22:54:25 by StephenWeeks.

MLtonRusage

```
signature MLTON_RUSAGE =
   sig
      type t = {utime: Time.time, (* user time *)
                stime: Time.time} (* system time *)

      val measureGC: bool -> unit
      val rusage: unit -> {children: t, gc: t, self: t}
   end
```

- `type t`
  corresponds to a subset of the C `struct rusage`.
- `measureGC b`
  controls whether garbage collection time is separately measured during program execution. This affects the behavior of both `rusage` and `Timer.checkCPUTimes`, both of which will return gc times of zero with `measureGC false`. Garbage collection time is always measured when either `gc-messages` or `gc-summary` is given as a runtime system option.
- `rusage ()`

corresponds to the C `getrusage` function. It returns the resource usage of the exited children, the garbage collector, and the process itself. The `self` component includes the usage of the `gc` component, regardless of whether `measureGC` is `true` or `false`. If `rusage` is used in a program, either directly, or indirectly via the `Timer` structure, then `measureGC true` is automatically called at the start of the program (it can still be disable by user code later).

Last edited on 2006-04-26 04:24:45 by StephenWeeks.

MLtonSignal

```
signature MLTON_SIGNAL =
   sig
      type t
      type signal = t

      structure Handler:
         sig
            type t

            val default: t
            val handler: (Thread.Runnable.t -> Thread.Runnable.t) -> t
            val ignore: t
            val isDefault: t -> bool
            val isIgnore: t -> bool
            val simple: (unit -> unit) -> t
         end

      structure Mask:
         sig
            type t

            val all: t
            val allBut: signal list -> t
            val block: t -> unit
            val getBlocked: unit -> t
            val isMember: t * signal -> bool
            val none: t
            val setBlocked: t -> unit
            val some: signal list -> t
            val unblock: t -> unit
         end

      val getHandler: t -> Handler.t
      val handled: unit -> Mask.t
      val prof: t
      val restart: bool ref
      val setHandler: t * Handler.t -> unit
      val suspend: Mask.t -> unit
      val vtalrm: t
   end
```

Signals handlers are functions from (runnable) threads to (runnable) threads. When a signal arrives, the corresponding signal handler is invoked, its argument being the thread that was interrupted by the signal. The signal handler runs asynchronously, in its own thread. The signal handler returns the thread that it would like to resume execution (this is often the thread that it was passed). It is an error for a signal handler to raise an exception that is not handled within the signal handler itself.

A signal handler is never invoked while the running thread is in a critical section (see MLtonThread). Invoking a signal handler implicitly enters a critical section and the normal return of a signal handler implicitly exits the critical section; hence, a signal handler is never interrupted by another signal handler.

- `type t`
  the type of signals.
- `type Handler.t`
  the type of signal handlers.

- `Handler.default`
  handles the signal with the default action.
- `Handler.handler f`
  returns a handler `h` such that when a signal `s` is handled by `h`, `f` will be passed the thread that was interrupted by `s` and should return the thread that will resume execution.
- `Handler.ignore`
  is a handler that will ignore the signal.
- `Handler.isDefault`
  returns true if the handler is the default handler.
- `Handler.isIgnore`
  returns true if the handler is the ignore handler.
- `Handler.simple f`
  returns a handler that executes `f ()` and does not switch threads.
- `type Mask.t`
  the type of signal masks, which are sets of blocked signals.
- `Mask.all`
  a mask of all signals.
- `Mask.allBut l`
  a mask of all signals except for those in `l`.
- `Mask.block m`
  blocks all signals in `m`.
- `Mask.getBlocked ()`
  gets the signal mask `m`, i.e. a signal is blocked if and only if it is in `m`.
- `Mask.isMember (m, s)`
  returns true if the signal `s` is in `m`.
- `Mask.none`
  a mask of no signals.
- `Mask.setBlocked m`
  sets the signal mask to `m`, i.e. a signal is blocked if and only if it is in `m`.
- `Mask.some l`
  a mask of the signals in `l`.
- `Mask.unblock m`
  unblocks all signals in `m`.
- `getHandler s`
  returns the current handler for signal `s`.
- `handled ()`
  returns the signal mask `m` corresponding to the currently handled signals; i.e., a signal is handled if and only if it is in `m`.
- `prof`
  `SIGPROF`, the profiling signal.
- `restart`
  dynamically determines the behavior of interrupted system calls; when `true`, interrupted system calls are restarted; when `false`, interrupted system calls raise `OS.SysError`.
- `setHandler (s, h)`
  sets the handler for signal `s` to `h`.
- `suspend m`
  temporarily sets the signal mask to `m` and suspends until an unmasked signal is received and handled, at which point `suspend` resets the mask and returns.
- `vtalrm`

`SIGVTALRM`, the signal for virtual timers.

Interruptible System Calls

Signal handling interacts in a non-trivial way with those functions in the Basis Library that correspond directly to interruptible system calls (a subset of those functions that may raise `OS.SysError`). The desire is that these functions should have predictable semantics. The principal concerns are:

1. System calls that are interrupted by signals should, by default, be restarted; the alternative is to raise

   ```
   OS.SysError (Posix.Error.errorMsg Posix.Error.intr,
                SOME Posix.Error.intr)
   ```

   This behavior is determined dynamically by the value of `Signal.restart`.
2. Signal handlers should always get a chance to run (when outside a critical region). If a system call is interrupted by a signal, then the signal handler will run before the call is restarted or `OS.SysError` is raised; that is, before the `Signal.restart` check.
3. A system call that must be restarted while in a critical section will be restarted with the handled signals blocked (and the previously blocked signals remembered). This encourages the system call to complete, allowing the program to make progress towards leaving the critical section where the signal can be handled. If the system call completes, the set of blocked signals are restored to those previously blocked.

Last edited on 2005-12-02 04:22:43 by StephenWeeks.

MLtonSocket

```
signature MLTON_SOCKET =
   sig
      structure Address:
         sig
            type t = word
         end
      structure Ctl:
         sig
            val getERROR: ('a, 'b) Socket.sock -> (string * int option) option
         end
      structure Host:
         sig
            type t = {name: string}

            val getByAddress: Address.t -> t option
            val getByName: string -> t option
         end
      structure Port:
         sig
            type t = int
         end

      type t

      val accept: t -> Address.t * Port.t * TextIO.instream * TextIO.outstream
      val connect: string * Port.t -> TextIO.instream * TextIO.outstream
      val fdToSock: Posix.FileSys.file_desc -> ('a, 'b) Socket.sock
      val listen: unit -> Port.t * t
      val listenAt: Port.t -> t
      val shutdownRead: TextIO.instream -> unit
      val shutdownWrite: TextIO.outstream -> unit
   end
```

This module contains a bare minimum of functionality to do TCP/IP programming. This module is implemented on top of the `Socket` module of the Standard Basis Library. We encourage you to use the standard `Socket` module, since we may eliminate `MLton.Socket` some day.

- `type Address.t`
  the type of IP addresses.
- `Ctl.getERROR s`
  like the Basis Library's `Socket.Ctl.getERROR`, except that it returns more information. `NONE` means that there was no error, and `SOME` means that there was an error, and provides the error message and error code, if any.
- `Host.getByAddress a`
  looks up the hostname (using `gethostbyaddr`) corresponding to `a`.
- `Host.getByName s`
  looks up the hostname (using `gethostbyname`) corresponding to `s`.
- `type Port.t`
  the type of TCP ports.
- `type t`
  the type of sockets.
- `accept s`

accepts a connection on socket `s` and return the address and port of the connecting socket, as well as streams corresponding to the connection.

- `connect (h, p)`
  connects to host `h` on port `p`, returning the streams corresponding to the connection.
- `fdToSock fd`
  coerces a file descriptor to a socket.
- `listen ()`
  listens to a port chosen by the system. Returns the port and the socket.
- `listenAt p`
  listens to port `p`. Returns the socket.
- `shutdownRead ins`
  causes the read part of the socket associated with `ins` to be shutdown.
- `shutdownWrite out`

causes the write part of the socket associated with `out` to be shutdown.

Last edited on 2005-12-01 23:07:28 by StephenWeeks.

MLtonStructure

The `MLton` structure contains a lot of functionality that is not available in the Basis Library. As a warning, please keep in mind that the `MLton` structure and its substructures do change from release to release of MLton.

```
structure MLton:
   sig
      val eq: 'a * 'a -> bool
      val isMLton: bool
      val share: 'a -> unit
      val shareAll: unit -> unit
      val size: 'a -> int

      structure Array: MLTON_ARRAY
      structure BinIO: MLTON_BIN_IO
      structure Cont: MLTON_CONT
      structure Exn: MLTON_EXN
      structure Finalizable: MLTON_FINALIZABLE
      structure GC: MLTON_GC
      structure IntInf: MLTON_INT_INF
      structure Itimer: MLTON_ITIMER
      structure Platform: MLTON_PLATFORM
      structure Pointer: MLTON_POINTER
      structure ProcEnv: MLTON_PROC_ENV
      structure Process: MLTON_PROCESS
      structure Profile: MLTON_PROFILE
      structure Random: MLTON_RANDOM
      structure Rlimit: MLTON_RLIMIT
      structure Rusage: MLTON_RUSAGE
      structure Signal: MLTON_SIGNAL
      structure Socket: MLTON_SOCKET
      structure Syslog: MLTON_SYSLOG
      structure TextIO: MLTON_TEXT_IO
      structure Thread: MLTON_THREAD
      structure Vector: MLTON_VECTOR
      structure Weak: MLTON_WEAK
      structure Word: MLTON_WORD where type word = Word.word
      structure Word8: MLTON_WORD where type word = Word8.word
      structure World: MLTON_WORLD
   end
```

Substructures

- MLtonArray
- MLtonBinIO
- MLtonCont
- MLtonExn
- MLtonFinalizable
- MLtonGC
- MLtonIntInf
- MLtonIO
- MLtonItimer
- MLtonPlatform
- MLtonPointer

* MLtonProcEnv
* MLtonProcess
* MLtonRandom
* MLtonRlimit
* MLtonRusage
* MLtonSignal
* MLtonSocket
* MLtonSyslog
* MLtonTextIO
* MLtonThread
* MLtonVector
* MLtonWeak
* MLtonWord
* MLtonWorld

Values

* `eq (x, y)`
  returns true if `x` and `y` are equal as pointers. For simple types like `char`, `int`, and `word`, this is the same as equals. For arrays, datatypes, strings, tuples, and vectors, this is a simple pointer equality. The semantics is a bit murky.
* `isMLton`
  is always true in a MLton implementation, and is always false in a stub implementation.
* `share x`
  maximizes sharing in the heap for the object graph reachable from `x`.
* `shareAll ()`
  maximizes sharing in the heap by sharing space for equivalent immutable objects. A call to `shareAll` performs a major garbage collection, and takes time proportional to the size of the heap.
* `size x`

returns the amount of heap space (in bytes) taken by the value of `x`, including all objects reachable from `x` by following pointers. It takes time proportional to the size of `x`. See below for an example.

Example of MLton.size

This example, `size.sml`, demonstrates the application of `MLton.size` to many different kinds of objects.

```
fun 'a printSize (name: string, min: int, value: 'a): unit=
   if MLton.size value >= min
      then
         (print "The size of "
          ; print name
          ; print " is >= "
          ; print (Int.toString min)
          ; print " bytes.\n")
   else ()

val l = [1, 2, 3, 4]

val _ =
    (
     printSize ("a char", 0, #"c")
     ; printSize ("an int list of length 4", 48, l)
```

```
    ; printSize ("a string of length 10", 24, "0123456789")
    ; printSize ("an int array of length 10", 52, Array.tabulate (10, fn _ => 0))
    ; printSize ("a double array of length 10",
                 92, Array.tabulate (10, fn _ => 0.0))
    ; printSize ("an array of length 10 of 2-ples of ints",
                 92, Array.tabulate (10, fn i => (i, i + 1)))
    ; printSize ("a useless function", 0, fn _ => 13)
    )

(* This is here so that the list is "useful".
 * If it were removed, then the optimizer (remove-unused-constructors)
 * would remove l entirely.
 *)
val _ = if 10 = foldl (op +) 0 l
           then ()
        else raise Fail "bug"

local
   open MLton.Cont
in
   val rc: int option t option ref = ref NONE
   val _ =
      case callcc (fn k: int option t => (rc := SOME k; throw (k, NONE))) of
         NONE => ()
       | SOME i => print (concat [Int.toString i, "\n"])
end

val _ =
   (print "The size of a continuation option ref is "
    ; if MLton.size rc > 1000
         then print "> 1000.\n"
      else print "< 1000.\n")

val _ =
   case !rc of
      NONE => ()
    | SOME k => (rc := NONE; MLton.Cont.throw (k, SOME 13))
```

Compile and run as usual.

```
% mlton size.sml
% ./size
The size of a char is >= 0 bytes.
The size of an int list of length 4 is >= 48 bytes.
The size of a string of length 10 is >= 24 bytes.
The size of an int array of length 10 is >= 52 bytes.
The size of a double array of length 10 is >= 92 bytes.
The size of an array of length 10 of 2-ples of ints is >= 92 bytes.
The size of a useless function is >= 0 bytes.
The size of a continuation option ref is > 1000.
13
The size of a continuation option ref is < 1000.
```

Last edited on 2005-12-01 23:10:46 by StephenWeeks.

MLtonSyslog

```
signature MLTON_SYSLOG =
   sig
      type openflag

      val CONS     : openflag
      val NDELAY   : openflag
      val PERROR   : openflag
      val PID      : openflag

      type facility

      val AUTHPRIV : facility
      val CRON     : facility
      val DAEMON   : facility
      val KERN     : facility
      val LOCAL0   : facility
      val LOCAL1   : facility
      val LOCAL2   : facility
      val LOCAL3   : facility
      val LOCAL4   : facility
      val LOCAL5   : facility
      val LOCAL6   : facility
      val LOCAL7   : facility
      val LPR      : facility
      val MAIL     : facility
      val NEWS     : facility
      val SYSLOG   : facility
      val USER     : facility
      val UUCP     : facility

      type loglevel

      val EMERG    : loglevel
      val ALERT    : loglevel
      val CRIT     : loglevel
      val ERR      : loglevel
      val WARNING  : loglevel
      val NOTICE   : loglevel
      val INFO     : loglevel
      val DEBUG    : loglevel

      val closelog: unit -> unit
      val log: loglevel * string -> unit
      val openlog: string * openflag list * facility -> unit
   end
```

`MLton.Syslog` is a complete interface to the system logging facilities. See `man 3 syslog` for more details.

- `closelog ()`
  closes the connection to the system logger.
- `log (l, s)`
  logs message `s` at a loglevel `l`.
- `openlog (name, flags, facility)`

opens a connection to the system logger. `name` will be prefixed to each message, and is typically set to the program name.

Last edited on 2005-12-01 23:11:30 by StephenWeeks.

MLtonTextIO

```
signature MLTON_TEXT_IO = MLTON_IO
```

See MLtonIO.

Last edited on 2005-12-01 23:11:52 by StephenWeeks.

MLtonThread

```
signature MLTON_THREAD =
   sig
      structure AtomicState:
         sig
            datatype t = NonAtomic | Atomic of int
         end

      val atomically: (unit -> 'a) -> 'a
      val atomicBegin: unit -> unit
      val atomicEnd: unit -> unit
      val atomicState: unit -> AtomicState.t

      structure Runnable:
         sig
            type t
         end

      type 'a t

      val atomicSwitch: ('a t -> Runnable.t) -> 'a
      val new: ('a -> unit) -> 'a t
      val prepend: 'a t * ('b -> 'a) -> 'b t
      val prepare: 'a t * 'a -> Runnable.t
      val switch: ('a t -> Runnable.t) -> 'a
   end
```

`MLton.Thread` provides access to MLton's user-level thread implementation (i.e. not OS-level threads). Threads are lightweight data structures that represent a paused computation. Runnable threads are threads that will begin or continue computing when `switch`ed to. `MLton.Thread` does not include a default scheduling mechanism, but it can be used to implement both preemptive and non-preemptive threads.

- `type AtomicState.t`
  the type of atomic states.
- `atomically f`
  runs `f` in a critical section.
- `atomicBegin ()`
  begins a critical section.
- `atomicEnd ()`
  ends a critical section.
- `atomicState ()`
  returns the current atomic state.
- `type Runnable.t`
  the type of threads that can be resumed.
- `type 'a t`
  the type of threads that expect a value of type `'a`.
- `atomicSwitch f`
  like `switch`, but assumes an atomic calling context. Upon `switch`ing back to the current thread, an implicit `atomicEnd` is performed.
- `new f`
  creates a new thread that, when run, applies `f` to the value given to the thread. `f` must terminate by `switch`ing to another thread or exiting the process.
- `prepend (t, f)`

creates a new thread (destroying `t` in the process) that first applies `f` to the value given to the thread and then continues with `t`. This is a constant time operation.

- `prepare (t, v)`
  prepares a new runnable thread (destroying `t` in the process) that will evaluate `t` on `v`.
- `switch f`

applies `f` to the current thread to get `rt`, and then start running thread `rt`. It is an error for `f` to perform another `switch`. `f` is guaranteed to run atomically.

Example of non-preemptive threads

```
structure Queue:
   sig
      type 'a t

      val new: unit -> 'a t
      val enque: 'a t * 'a -> unit
      val deque: 'a t -> 'a option
   end =
   struct
      datatype 'a t = T of {front: 'a list ref, back: 'a list ref}

      fun new() = T{front = ref [], back = ref []}

      fun enque(T{back, ...}, x) = back := x :: !back

      fun deque(T{front, back}) =
         case !front of
            [] => (case !back of
                      [] => NONE
                    | l => let val l = rev l
                           in case l of
                              [] => raise Fail "deque"
                            | x :: l => (back := []; front := l; SOME x)
                           end)
          | x :: l => (front := l; SOME x)
   end

structure Thread:
   sig
      val exit: unit -> 'a
      val run: unit -> unit
      val spawn: (unit -> unit) -> unit
      val yield: unit -> unit
   end =
   struct
      open MLton
      open Thread

      val topLevel: Thread.Runnable.t option ref = ref NONE

      local
         val threads: Thread.Runnable.t Queue.t = Queue.new()
      in
         fun ready (t: Thread.Runnable.t) : unit =
            Queue.enque(threads, t)
         fun next () : Thread.Runnable.t =
            case Queue.deque threads of
```

```
            NONE => valOf(!topLevel)
          | SOME t => t
      end

      fun 'a exit(): 'a = switch(fn _ => next())

      fun new(f: unit -> unit): Thread.Runnable.t =
         Thread.prepare
         (Thread.new (fn () => ((f() handle _ => exit())
                                 ; exit())),
          ())

      fun schedule t = (ready t; next())

      fun yield(): unit = switch(fn t => schedule (Thread.prepare (t, ())))

      val spawn = ready o new

      fun run(): unit =
         (switch(fn t =>
                 (topLevel := SOME (Thread.prepare (t, ()))
                  ; next()))
          ; topLevel := NONE)
   end

val rec loop =
   fn 0 => ()
    | n => (print(concat[Int.toString n, "\n"])
            ; Thread.yield()
            ; loop(n - 1))

val rec loop' =
   fn 0 => ()
    | n => (Thread.spawn(fn () => loop n); loop'(n - 2))

val _ = Thread.spawn(fn () => loop' 10)

val _ = Thread.run()

val _ = print "success\n"
```

Example of preemptive threads

```
structure Queue:
   sig
      type 'a t

      val new: unit -> 'a t
      val enque: 'a t * 'a -> unit
      val deque: 'a t -> 'a option
   end =
   struct
      datatype 'a t = T of {front: 'a list ref, back: 'a list ref}

      fun new () = T {front = ref [], back = ref []}

      fun enque (T {back, ...}, x) = back := x :: !back

      fun deque (T {front, back}) =
         case !front of
```

```
               [] => (case !back of
                         [] => NONE
                       | l => let val l = rev l
                              in case l of
                                    [] => raise Fail "deque"
                                  | x :: l => (back := []; front := l; SOME x)
                              end)
             | x :: l => (front := l; SOME x)
   end

structure Thread:
   sig
      val exit: unit -> 'a
      val run: unit -> unit
      val spawn: (unit -> unit) -> unit
      val yield: unit -> unit
   end =
   struct
      open Posix.Signal
      open MLton
      open Itimer Signal Thread

      val topLevel: Thread.Runnable.t option ref = ref NONE

      local
         val threads: Thread.Runnable.t Queue.t = Queue.new ()
      in
         fun ready t = Queue.enque (threads, t)
         fun next () =
            case Queue.deque threads of
               NONE => valOf (!topLevel)
             | SOME t => t
      end

      fun 'a exit (): 'a = switch (fn _ => next ())

      fun new (f: unit -> unit): Thread.Runnable.t =
         Thread.prepare
         (Thread.new (fn () => ((f () handle _ => exit ())
                                 ; exit ())),
          ())

      fun schedule t = (ready t; next ())

      fun yield (): unit = switch (fn t => schedule (Thread.prepare (t, ())))

      val spawn = ready o new

      fun setItimer t =
         Itimer.set (Itimer.Real,
                     {value = t,
                      interval = t})

      fun run (): unit =
         (switch (fn t =>
                  (topLevel := SOME (Thread.prepare (t, ()))
                   ; new (fn () => (setHandler (alrm, Handler.handler schedule)
                                    ; setItimer (Time.fromMilliseconds 20)))))
          ; setItimer Time.zeroTime
          ; ignore alrm
          ; topLevel := NONE)
```

```
   end

val rec delay =
   fn 0 => ()
    | n => delay (n - 1)

val rec loop =
   fn 0 => ()
    | n => (delay 500000; loop (n - 1))

val rec loop' =
   fn 0 => ()
    | n => (Thread.spawn (fn () => loop n); loop' (n - 1))

val _ = Thread.spawn (fn () => loop' 10)

val _ = Thread.run ()

val _ = print "success\n"
```

Last edited on 2005-12-02 03:52:11 by MatthewFluet.

MLtonVector

```
signature MLTON_VECTOR =
   sig
      val unfoldi: int * 'b * (int * 'b -> 'a * 'b) -> 'a vector
   end
```

- `unfoldi (n, b, f)`

constructs a vector v of a length `n`, whose elements v_i are determined by the equations $b_0 = b$ and $(v_i, b_{i+1}) = f(i, b_i)$.

Last edited on 2005-12-01 23:14:39 by StephenWeeks.

MLtonWeak

```
signature MLTON_WEAK =
   sig
      type 'a t

      val get: 'a t -> 'a option
      val new: 'a -> 'a t
   end
```

A weak pointer is a pointer to an object that is nulled if the object becomes unreachable due to garbage collection. The weak pointer does not itself cause the object it points to be retained by the garbage collector -- only other strong pointers can do that. For objects that are not allocated in the heap, like integers, a weak pointer will always be nulled. So, if `w: int Weak.t` then `Weak.get w = NONE`.

- `type 'a t`
  the type of weak pointers to objects of type `'a`
- `get w`
  returns `NONE` if the object pointed to by `w` no longer exists. Otherwise, returns `SOME` of the object pointed to by `w`.
- `new x`

returns a weak pointer to `x`.

Last edited on 2006-08-21 19:41:38 by StephenWeeks.

MLtonWord

```
signature MLTON_WORD =
   sig
      type t

      val rol: t * word -> t
      val ror: t * word -> t
   end
```

- `type t`
  the type of words. For `MLton.Word` this is `Word.word`, for `MLton.Word8` this is `Word8.word`.
- `rol (w, w')`
  rotates left (circular).
- `ror (w, w')`

rotates right (circular).

Last edited on 2005-12-01 23:15:27 by StephenWeeks.

MLtonWorld

```
signature MLTON_WORLD =
   sig
      datatype status = Clone | Original

      val load: string -> 'a
      val save: string -> status
      val saveThread: string * Thread.Runnable.t -> unit
   end
```

- `datatype status`
  specifies whether a world is original or restarted (a clone).
- `load f`
  loads the saved computation from file `f`.
- `save f`
  saves the entire state of the computation to the file `f`. The computation can then be restarted at a later time using `World.load` or the `load-world` runtime option. The call to `save` in the original computation returns `Original` and the call in the restarted world returns `Clone`.
- `saveThread (f, rt)`

saves the entire state of the computation to the file `f` that will resume with thread `rt` upon restart.

Example

Suppose that `save-world.sml` contains the following.

```
open MLton.World
val _ =
   case save "world" of
      Original => print "I am the original\n"
    | Clone => print "I am the clone\n"
```

Then, if we compile `save-world.sml` and run it, the `Original` branch will execute, and a file named `world` will be created.

```
% mlton save-world.sml
% save-world
I am the original
```

We can then load `world` using the `load-world` run time option.

```
% save-world @MLton load-world world --
I am the clone
```

Last edited on 2005-12-01 23:17:27 by StephenWeeks.

Machine

Machine is an IntermediateLanguage, translated from RSSA by ToMachine and used as input by the Codegen.

Description

Machine is an Untyped IntermediateLanguage, corresponding to a abstract register machine.

Implementation

machine.sig machine.fun

Type Checking

The Machine IntermediateLanguage has a primitive type checker, which only checks some liveness properties.

machine.sig machine.fun

Details and Notes

Last edited on 2005-12-01 19:27:42 by StephenWeeks.

ManualPage

MLton is run from the command line with a collection of options followed by a file name and a list of files to compile, assemble, and link with.

```
mlton [option ...] file.{c|cm|mlb|o|sml} [file.{c|o|s|S} ...]
```

The simplest case is to run `mlton foo.sml`, where `foo.sml` contains a valid SML program, in which case MLton compiles the program to produce an executable `foo`. Since MLton does not support separate compilation, the program must be the entire program you wish to compile. However, the program may refer to signatures and structures defined in the Basis Library.

Larger programs, spanning many files, can be compiled with the ML Basis system. In this case, `mlton foo.mlb` will compile the complete SML program described by the basis `foo.mlb`, which may specify both SML files and additional bases.

MLton also supports a limited subset of SML/NJ CompilationManager (CM) files. For example, `mlton foo.cm` will compile the complete SML program consisting of the concatenation of all the SML files referred to (either directly or indirectly) by `foo.cm`.

Next Steps

- CompileTimeOptions
- RunTimeOptions

Last edited on 2005-12-01 19:31:43 by StephenWeeks.

MatchCompilation

Match compilation is the process of translating an SML match into a nested tree (or dag) of simple case expressions and tests.

MLton's match compiler is described here.

Match compilation in other compilers

- BaudinetMacqueen85
- Leroy90, pages 60-69.
- Scott00
- Sestoft96

Last edited on 2005-07-26 18:19:23 by StephenWeeks.

MatchCompile

MatchCompile is a translation pass, agnostic in the IntermediateLanguages between which it translates.

Description

Match compilation converts a case expression with nested patterns into a case expression with flat patterns.

Implementation

match-compile.sig match-compile.fun

Details and Notes

```
val matchCompile:
   {caseType: Type.t, (* type of entire expression *)
    cases: (NestedPat.t * ((Var.t -> Var.t) -> Exp.t)) vector,
    conTycon: Con.t -> Tycon.t,
    region: Region.t,
    test: Var.t,
    testType: Type.t,
    tyconCons: Tycon.t -> {con: Con.t, hasArg: bool} vector}
   -> Exp.t * (unit -> ((Layout.t * {isOnlyExns: bool}) vector) vector)
```

`matchCompile` is complicated by the desire for modularity between the match compiler and its caller. Its caller is responsible for building the right hand side of a rule `p => e`. On the other hand, the match compiler is responsible for destructing the test and binding new variables to the components. In order to connect the new variables created by the match compiler with the variables in the pattern `p`, the match compiler passes an environment back to its caller that maps each variable in `p` to the corresponding variable introduced by the match compiler.

The match compiler builds a tree of n-way case expressions by working from outside to inside and left to right in the patterns. For example,

```
case x of
  (_, C1 a) => e1
| (C2 b, C3 c) => e2
```

is translated to

```
let
   fun f1 a = e1
   fun f2 (b, c) = e2
in
  case x of
     (x1, x2) =>
       (case x1 of
          C2 b' => (case x2 of
                      C1 a' => f1 a'
                    | C3 c' => f2(b',c')
                    | _ => raise Match)
        | _ => (case x2 of
                  C1 a'' => f1 a''
```

```
                | _ => raise Match))
end
```

Here you can see the necessity of abstracting out the ride hand sides of the cases in order to avoid code duplication. Right hand sides are always abstracted. The simplifier cleans things up. You can also see the new (primed) variables introduced by the match compiler and how the renaming works. Finally, you can see how the match compiler introduces the necessary default clauses in order to make a match exhaustive, i.e. cover all the cases.

The match compiler uses `numCons` and `tyconCons` to determine the exhaustivity of matches against constructors.

Last edited on 2005-12-01 19:33:22 by StephenWeeks.

MatthewFluet

Matthew Fluet (mfluet@acm.org , http://www.cs.cornell.edu/People/fluet) is a PhD student in the Computer Science Department at Cornell University.

Current MLton projects:

- Migrating SSA optimizations to SSA2
- Improving CML implementation
- Porting ML-Doc
- Porting ML-NLFFI
- Porting ML-RISC

Last edited on 2005-12-01 19:37:05 by StephenWeeks.

MichaelNorrish

I am a researcher at NICTA, with a web-page here.

I'm interested in MLton because of the chance that it might be a good vehicle for future implementations of the HOL theorem-proving system. It's beginning to look as if one route forward will be to embed an SML interpreter into a MLton-compiled executable. I don't know if an extensible interpreter of the kind we're looking for already exists.

Last edited on 2005-04-05 06:48:34 by MichaelNorrish.

MikeThomas

Here is a picture at home in Brisbane, Queensland, Australia, taken in January 2004.

image

Last edited on 2004-10-27 18:15:50 by StephenWeeks.

MoinMoin

MoinMoin is the wiki engine used to implement this site.

You can find out technical specifics about this particular instance of MoinMoin at the SystemInfo page.

Last edited on 2004-10-25 20:51:11 by StephenWeeks.

Monomorphise

Monomorphise is a translation pass from the XML IntermediateLanguage to the SXML IntermediateLanguage.

Description

Monomorphisation eliminates polymorphic values and datatype declarations by duplicating them for each type at which they are used.

Consider the following XML program.

```
datatype 'a t = T of 'a
fun 'a f (x: 'a) = T x
val a = f 1
val b = f 2
val z = f (3, 4)
```

The result of monomorphising this program is the following SXML program:

```
datatype t1 = T1 of int
datatype t2 = T2 of int * int
fun f1 (x: t1) = T1 x
fun f2 (x: t2) = T2 x
val a = f1 1
val b = f1 2
val z = f2 (3, 4)
```

Implementation

monomorphise.sig monomorphise.fun

Details and Notes

The monomorphiser works by making one pass over the entire program. On the way down, it creates a cache for each variable declared in a polymorphic declaration that maps a lists of type arguments to a new variable name. At a variable reference, it consults the cache (based on the types the variable is applied to). If there is already an entry in the cache, it is used. If not, a new entry is created. On the way up, the monomorphiser duplicates a variable declaration for each entry in the cache.

As with variables, the monomorphiser records all of the type at which constructors are used. After the entire program is processed, the monomorphiser duplicates each datatype declaration and its associated constructors.

The monomorphiser duplicates all of the functions declared in a `fun` declaration as a unit. Consider the following program

```
fun 'a f (x: 'a) = g x
and g (y: 'a) = f y
val a = f 13
val b = g 14
val c = f (1, 2)
```

and its monomorphisation

```
fun f1 (x: int) = g1 x
and g1 (y: int) = f1 y
fun f2 (x : int * int) = g2 x
and g2 (y : int * int) = f2 y
val a = f1 13
val b = g1 14
val c = f2 (1, 2)
```

Pathological datatype declarations

SML allows a pathological polymorphic datatype declaration in which recursive uses of the defined type constructor are applied to different type arguments than the definition. This has been disallowed by others on type theoretic grounds. A canonical example is the following.

```
datatype 'a t = A of 'a | B of ('a * 'a) t
val z : int t = B (B (A ((1, 2), (3, 4))))
```

The presence of the recursion in the datatype declaration might appear to cause the need for the monomorphiser to create an infinite number of types. However, due to the absence of polymorphic recursion in SML, there are in fact only a finite number of instances of such types in any given program. The monomorphiser translates the above program to the following one.

```
datatype t1 = B1 of t2
datatype t2 = B2 of t3
datatype t3 = A3 of (int * int) * (int * int)
val z : int t = B1 (B2 (A3 ((1, 2), (3, 4))))
```

It is crucial that the monomorphiser be allowed to drop unused constructors from datatype declarations in order for the translation to terminate.

Last edited on 2005-12-02 04:22:52 by StephenWeeks.

MoscowML

Moscow ML is a Standard ML Compiler. It is a byte-code compiler, so it compiles code quickly, but the code runs slowly. See Performance.

Last edited on 2004-12-30 20:11:52 by StephenWeeks.

Multi

Multi is an analysis pass for the SSA IntermediateLanguage, invoked from ConstantPropagation and LocalRef.

Description

This pass analyzes the control flow of a SSA program to determine which SSA functions and blocks might be executed more than once or by more than one thread. It also determines when a program uses threads and when functions and blocks directly or indirectly invoke `Thread_copyCurrent`.

Implementation

multi.sig multi.fun

Details and Notes

Last edited on 2005-12-01 23:18:59 by StephenWeeks.

Mutable

Mutable is an adjective meaning can be modified. In Standard ML, ref cells and arrays are mutable, while all other values are immutable.

Last edited on 2004-12-08 18:51:14 by StephenWeeks.

NumericLiteral

Numeric literals in Standard ML can be written in either decimal or hexadecimal notation. Sometimes it can be convenient to write numbers down in other bases. Fortunately, using Fold, it is possible to define a concise syntax for numeric literals that allows one to write numeric constants in any base and of various types (`int`, `IntInf.int`, `word`, and more).

We will define constants `I`, `II`, `W`, and `` ` `` so that, for example,

```
I 10 `1`2`3 $
```

denotes `123:int` in base 10, while

```
II 8 `2`3 $
```

denotes `19:IntInf.int` in base 8, and

```
W 2 `1`1`0`1 $
```

denotes `0w13: word`.

Here is the code.

```
structure Num =
   struct
      fun make (op *, op +, i2x) iBase =
          let
             val xBase = i2x iBase
          in
             Fold.fold
                ((i2x 0,
                  fn (i, x) =>
                     if 0 <= i andalso i < iBase then
                        x * xBase + i2x i
                     else
                        raise Fail (concat
                                       ["Num: ", Int.toString i,
                                        " is not a valid\
                                        \ digit in base ",
                                        Int.toString iBase])),
                 fst)
          end

      fun I  ? = make (op *, op +, id) ?
      fun II ? = make (op *, op +, IntInf.fromInt) ?
      fun W  ? = make (op *, op +, Word.fromInt) ?

      fun ` ? = Fold.step1 (fn (i, (x, step)) =>
                               (step (i, x), step)) ?

      val a = 10
      val b = 11
      val c = 12
      val d = 13
      val e = 14
      val f = 15
```

```
  end
```

where

```
fun fst (x, _) = x
```

The idea is for the fold to start with zero and to construct the result one digit at a time, with each stepper multiplying the previous result by the base and adding the next digit. The code is abstracted in two different ways for extra generality. First, the `make` function abstracts over the various primitive operations (addition, multiplication, etc) that are needed to construct a number. This allows the same code to be shared for constants `I`, `II`, `W` used to write down the various numeric types. It also allows users to add new constants for additional numeric types, by supplying the necessary arguments to make.

Second, the step function, `` ` ``, is abstracted over the actual construction operation, which is created by make, and passed along the fold. This allows the same constant, `` ` ``, to be used for all numeric types. The alternative approach, having a different step function for each numeric type, would be more painful to use.

On the surface, it appears that the code checks the digits dynamically to ensure they are valid for the base. However, MLton will simplify everything away at compile time, leaving just the final numeric constant.

Last edited on 2006-05-28 08:52:54 by VesaKarvonen.

OCaml

OCaml is a variant of ML and is similar to Standard ML.

OCaml and SML

Here's a comparison of some aspects of the OCaml and SML languages.

- Standard ML has a formal Definition, while OCaml is specified by its lone implementation and informal documentation.
- Standard ML has a number of compilers, while OCaml has only one.
- OCaml has built-in support for object-oriented programming, while Standard ML does not (however, see ObjectOrientedProgramming).
- Andreas Rossberg has a side-by-side comparison of the syntax of SML and OCaml.

OCaml and MLton

Here's a comparison of some aspects of OCaml and MLton.

- Performance
    - Both OCaml and MLton have excellent performance.
    - MLton performs extensive WholeProgramOptimization, which can provide substantial improvements in large, modular programs.
    - MLton uses native types, like 32-bit integers, without any penalty due to tagging or boxing. OCaml uses 31-bit integers with a penalty due to tagging, and 32-bit integers with a penalty due to boxing.
    - MLton uses native types, like 64-bit floats, without any penalty due to boxing. OCaml, in some situations, boxes 64-bit floats.
    - MLton represents arrays of all types unboxed. In OCaml, only arrays of 64-bit floats are unboxed, and then only when it is syntactically apparent.
    - MLton represents records compactly by reordering and packing the fields.
    - In MLton, polymorphic and monomorphic code have the same performance. In OCaml, polymorphism can introduce a performance penalty.
    - In MLton, module boundaries have no impact on performance. In OCaml, moving code between modules can cause a performance penalty.
- MLton's ForeignFunctionInterface is simpler than OCaml's.
- Tools
    - OCaml has a debugger, while MLton does not.
    - OCaml supports separate compilation, while MLton does not.
    - OCaml compiles faster than MLton.
    - MLton supports profiling of both time and allocation.
- Libraries
    - OCaml has more available libraries.
- Community
    - OCaml has a larger community than MLton.
    - MLton has a very responsive developer list.

Last edited on 2005-12-02 04:23:05 by StephenWeeks.

ObjectOrientedProgramming

Standard ML does not have explicit support for object-oriented programming. Here are some papers that show how to express certain object-oriented concepts in SML.

- OO Programming styles in ML
- Object-oriented programming and Standard ML
- mGTK: An SML binding of Gtk+

Last edited on 2005-12-01 23:20:26 by StephenWeeks.

OpenGL

There are at least two interfaces to OpenGL for MLton/SML, both of which should be considered alpha quality.

- MikeThomas built a low-level interface, directly translating many of the functions, covering GL, GLU, and GLUT. This is available in the MLton Sources: opengl . The code contains a number of small, standard OpenGL examples translated to SML.
- ChrisClearwater has written at least an interface to GL, and possibly more. See

http://mlton.org/pipermail/mlton/2005-January/026669.html

Contact us for more information or an update on the status of these projects.

Last edited on 2005-09-06 23:29:26 by MatthewFluet.

OperatorPrecedence

Standard ML has a built in notion of precedence for certain symbols. Every program that includes the Basis Library automatically gets the following infix declarations. Higher number indicates higher precedence.

```
infix 7 * / mod div
infix 6 + - ^
infixr 5 :: @
infix 4 = <> > >= < <=
infix 3 := o
infix 0 before
```

Last edited on 2005-12-02 04:23:19 by StephenWeeks.

OptionalArguments

Standard ML does not have built-in support for optional arguments. Nevertheless, using Fold, it is easy to define functions that take optional arguments.

For example, suppose that we have the following definition of a function f.

```
fun f (i, r, s) =
   concat [Int.toString i, ", ", Real.toString r, ", ", s]
```

Using the OptionalArg structure described below, we can define a function f', an optionalized version of f, that takes 0, 1, 2, or 3 arguments. Embedded within f' will be default values for i, r, and s. If f' gets no arguments, then all the defaults are used. If f' gets one argument, then that will be used for i. Two arguments will be used for i and r respectively. Three arguments will override all default values. Calls to f' will look like the following.

```
f' $
f' `2 $
f' `2 `3.0 $
f' `2 `3.0 `"four" $
```

The optional argument indicator, "`", is not special syntax -- it is a normal SML value, defined in the OptionalArg structure below.

Here is the definition of f' using the OptionalArg structure, in particular, OptionalArg.make and OptionalArg.D.

```
val f' =
   fn z =>
   let open OptionalArg in
      make (D 1) (D 2.0) (D "three") $
   end (fn i & r & s => f (i, r, s))
   z
```

The definition of f' is eta expanded as with all uses of fold. A call to OptionalArg.make is supplied with a variable number of defaults (in this case, three), the end-of-arguments terminator, $, and the function to run, taking its arguments as an n-ary product. In this case, the function simply converts the product to an ordinary tuple and calls f. Often, the function body will simply be written directly.

In general, the definition of an optional-argument function looks like the following.

```
val f =
   fn z =>
   let open OptionalArg in
      make (D <default1>) (D <default2>) ... (D <defaultn>) $
   end (fn x1 & x2 & ... & xn =>
        <function code goes here>)
   z
```

Here is the definition of OptionalArg.

```
structure OptionalArg =
   struct
      val make =
```

```
         fn z =>
         Fold.fold
         ((id, fn (f, x) => f x),
          fn (d, r) => fn func =>
          Fold.fold ((id, d ()), fn (f, d) =>
                    let
                       val d & () = r (id, f d)
                    in
                       func d
                    end))
         z

      fun D d = Fold.step0 (fn (f, r) =>
                           (fn ds => f (d & ds),
                            fn (f, a & b) => r (fn x => f a & x, b)))

      val ` =
         fn z =>
         Fold.step1 (fn (x, (f, _ & d)) => (fn d => f (x & d), d))
         z
   end
```

`OptionalArg.make` uses a nested fold. The first `fold` accumulates the default values in a product, associated to the right, and a reversal function that converts a product (of the same arity as the number of defaults) from right associativity to left associativity. The accumulated defaults are used by the second fold, which recurs over the product, replacing the appropriate component as it encounters optional arguments. The second fold also constructs a "fill" function, `f`, that is used to reconstruct the product once the end-of-arguments is reached. Finally, the finisher reconstructs the product and uses the reversal function to convert the product from right associative to left associative, at which point it is passed to the user-supplied function.

Much of the complexity comes from the fact that while recurring over a product from left to right, one wants it to be right-associative, e.g. look like

```
a & (b & (c & d))
```

but the user function in the end wants the product to be left associative, so that the product argument pattern can be written without parentheses (since `&` is left associative).

Labelled optional arguments

In addition to the positional optional arguments described above, it is sometimes useful to have labelled optional arguments. These allow one to define a function, `f`, with defaults, say `a` and `b`. Then, a caller of `f` can supply values for `a` and `b` by name. If no value is supplied then the default is used.

Labelled optional arguments are a simple extension of FunctionalRecordUpdate using post composition. Suppose, for example, that one wants a function `f` with labelled optional arguments `a` and `b` with default values `0` and `0.0` respectively. If one has a functional-record-update function `updateAB` for records with `a` and `b` fields, then one can define `f` in the following way.

```
val f =
   fn z =>
   Fold.post
   (updateAB {a = 0, b = 0.0},
    fn {a, b} => print (concat [Int.toString a, " ",
```

```
                                Real.toString b, "\n"]))
   z
```

The idea is that `f` is the post composition (using `Fold.post`) of the actual code for the function with a functional-record updator that starts with the defaults.

Here are some example calls to `f`.

```
val () = f $
val () = f (U#a 13) $
val () = f (U#a 13) (U#b 17.5) $
val () = f (U#b 17.5) (U#a 13) $
```

Notice that a caller can supply neither of the arguments, either of the arguments, or both of the arguments, and in either order. All that matter is that the arguments be labelled correctly (and of the right type, of course).

Here is another example.

```
val f =
   fn z =>
   Fold.post
   (updateBCD {b = 0, c = 0.0, d = "<>"},
    fn {b, c, d} =>
    print (concat [Int.toString b, " ",
                   Real.toString c, " ",
                   d, "\n"]))
   z
```

Here are some example calls.

```
val () = f $
val () = f (U#d "goodbye") $
val () = f (U#d "hello") (U#b 17) (U#c 19.3) $
```

Last edited on 2006-03-21 23:48:03 by VesaKarvonen.

OrphanedPages

Pages that no other page links to. Also see WantedPages.

1. CompilingWithSMLNJ
2. FirstClassPolymorphism
3. LanguageChanges
4. MLmon
5. PortStatus
6. PrintfGentle
7. Roadmap
8. ShowProf
9. Stabilizers
10. StandardMLGotchas
11. Survey
12. SurveyDone
13. Variant
14. ZZZOrphanedPages

Last edited on 2004-11-09 14:46:17 by StephenWeeks.

OtherSites

Other sites that have a MLton page (or more).

- Advogato
- Debian GNU/Linux (developer)
- FreeBSD
- freshmeat
- freshports
- GNU
- icewalkers
- Ubuntu
- wikipedi

Last edited on 2006-06-16 19:17:13 by StephenWeeks.

Overloading

In Standard ML, constants (like `13`, `0w13`, `13.0`) are overloaded, meaning that they can denote a constant of the appropriate type as determined by context. SML defines the overloading classes Int, Real, and Word, which denote the sets of types that integer, real, and word constants may take on. In MLton, these are defined as follows.

Int	`Int2.int`, `Int3.int`, ... `Int32.int`, `Int64.int`, `LargeInt.int`, `IntInf.int`
Real	`Real32.real`, `Real64.real`, `LargeReal.real`
Word	`Word2.word`, `Word3.word`, ... `Word32.word`, `Word64.int`, `LargeWord.word`

The Definition allows flexibility in how much context is used to resolve overloading. It says that the context is *no larger than the smallest enclosing structure-level declaration*, but that *an implementation may require that a smaller context determines the type*. MLton uses the largest possible context allowed by SML in resolving overloading. If the type of a constant is not determined by context, then it takes on a default type. In MLton, these are defined as follows.

Int	Int32.int
Real	Real64.real
Word	Word64.word

Other implementations may use a smaller context or different default types.

Also see

- discussion of overloading in the Basis Library

Examples

- The following program is rejected.

```
structure S:
   sig
      val x: Word8.word
   end =
   struct
      val x = 0w0
   end
```

The smallest enclosing structure declaration for `0w0` is `val x = 0w0`. Hence, `0w0` receives the default type for words, which is `Word32.word`.

Last edited on 2005-12-02 01:19:19 by StephenWeeks.

PackedRepresentation

PackedRepresentation is an analysis pass for the SSA2 IntermediateLanguage, invoked from ToRSSA.

Description

This pass analyzes a SSA2 program to compute a packed representation for each object.

Implementation

representation.sig packed-representation.fun

Details and Notes

Has a special case to make sure that `true` is represented as `1` and `false` is represented as `0`.

Last edited on 2005-12-01 23:24:19 by StephenWeeks.

PageSize

List of all pages, sorted by their size:

1. 30144 Fold
2. 27007 References
3. 19518 TypeIndexedValues
4. 14012 BasisLibrary
5. 11993 WesleyTerpstra/MoinEditorBackup
6. 11959 Performance
7. 11462 MLtonProcess
8. 11164 CommonArg
9. 10772 ConcurrentMLImplementation
10. 8502 CompileTimeOptions
11. 8426 MichaelNorrish/MoinEditorBackup
12. 8379 ValueRestriction
13. 8317 SyntacticConventions
14. 8291 PrintfGentle
15. 8242 InfixingOperators
16. 7268 FunctionalRecordUpdate
17. 6796 ForLoops
18. 6725 MLtonThread
19. 6674 PortingMLton
20. 6421 MLNLFFIImplementation
21. 6382 Bugs20041109
22. 6344 SMLNJDeviations
23. 6051 Elaborate
24. 5977 OptionalArguments
25. 5951 PolymorphicEquality
26. 5838 MLtonFinalizable
27. 5713 AdmitsEquality
28. 5553 MLtonSignal
29. 5536 Fixpoints
30. 5337 Features
31. 5326 CompilationManager
32. 5162 XML
33. 5094 MLtonStructure
34. 4709 Regions
35. 4702 CallGraph
36. 4651 TypeChecking
37. 4577 Credits
38. 4456 TypeVariableScope
39. 4449 StandardMLGotchas
40. 4440 MLBasisExamples
41. 4301 SelfCompiling
42. 4244 Fold01N
43. 4212 MLtonProfile
44. 4092 Printf
45. 3997 SMLNJLibrary
46. 3961 UniversalType

47. 3766 MatthewFluet/MoinEditorBackup
48. 3674 Users
49. 3593 VariableArityPolymorphism
50. 3518 EqualityTypeVariable
51. 3515 Download
52. 3497 Enscript
53. 3346 HenryCejtin/MoinEditorBackup
54. 3284 ForeignFunctionInterfaceTypes
55. 3193 MLBasisSyntaxAndSemantics
56. 3114 RunTimeOptions
57. 2997 NumericLiteral
58. 2995 ArrayLiteral
59. 2971 FirstClassPolymorphism
60. 2967 Monomorphise
61. 2965 ForeignFunctionInterfaceSyntax
62. 2910 HowProfilingWorks
63. 2821 PropertyList
64. 2740 Stabilizers
65. 2662 Installation
66. 2574 CompilerOverview
67. 2572 MLtonSocket
68. 2547 UnresolvedBugs
69. 2522 MLRISCLibrary
70. 2501 MatchCompile
71. 2495 CrossCompiling
72. 2493 GenerativeException
73. 2451 ProfilingTime
74. 2450 ChrisClearwater/MoinEditorBackup
75. 2433 ShowBasis
76. 2355 OCaml
77. 2342 CallingFromCToSML
78. 2322 MLBasisAnnotations
79. 2288 History
80. 2238 MLtonPointer
81. 2212 RefFlatten
82. 2159 MLBasisAnnotationExamples
83. 2146 ProfilingCounts
84. 2130 TomMurphy/MoinEditorBackup
85. 2129 LanguageChanges
86. 2106 RunningOnMinGW
87. 2085 CKitLibrary
88. 2021 MLBasisAvailableLibraries
89. 1976 SimplifyTypes
90. 1923 JesperLouisAndersen/MoinEditorBackup
91. 1921 WikiTool
92. 1919 MLBasis
93. 1918 XMLShrink
94. 1917 CompilingWithSMLNJ
95. 1900 Libraries
96. 1883 FAQ
97. 1876 ProductType

98. 1853 SMLofNJStructure
99. 1845 Unicode
100. 1841 Emacs
101. 1835 RunningOnNetBSD
102. 1831 ConcurrentML
103. 1798 CallingFromSMLToC
104. 1791 Overloading
105. 1787 Defunctorize
106. 1751 JesperLouisAndersen
107. 1665 CallingFromSMLToCFunctionPointer
108. 1600 MLtonSyslog
109. 1562 DefineTypeBeforeUse
110. 1554 SSA2
111. 1523 ScopeInference
112. 1497 Shrink
113. 1487 MLmon
114. 1485 Contact
115. 1474 MLtonWorld
116. 1470 Zone
117. 1454 Projects
118. 1441 KnownCase
119. 1434 EqualityType
120. 1423 MLtonRlimit
121. 1403 ProfilingAllocation
122. 1392 LocalRef
123. 1380 StephenWeeks/MoinEditorBackup
124. 1348 TypeConstructor
125. 1339 SSA
126. 1330 ToMachine
127. 1324 Sources
128. 1314 Preferences
129. 1313 MLtonRusage
130. 1289 MLBasisPathMap
131. 1285 ShowProf
132. 1280 Utilities
133. 1259 ProfilingTheStack
134. 1213 ManualPage
135. 1212 Restore
136. 1209 MLNLFFI
137. 1209 WebSite
138. 1203 StandardMLPortability
139. 1203 Useless
140. 1194 ImplementSuffix
141. 1157 MLtonPlatform
142. 1139 SSASimplify
143. 1136 RunningOnCygwin
144. 1133 DeadCode
145. 1129 MLtonCont
146. 1116 Lazy
147. 1110 LambdaFree
148. 1106 CommonBlock

149. 1104 Closure
150. 1103 MLtonRandom
151. 1102 MLtonIO
152. 1083 PortStatus
153. 1044 WikiMacros
154. 1038 SXML
155. 1024 RSSA
156. 1003 CommonSubexp
157. 1002 WikiSandBox
158. 998 WholeProgramOptimization
159. 972 MLtonIntInf
160. 960 ReleaseChecklist
161. 957 StandardML
162. 951 MLtonExn
163. 946 UnsafeStructure
164. 940 Identifier
165. 938 PolyEqual
166. 909 License
167. 863 MLtonWeak
168. 861 Flatten
169. 860 MLtonGC
170. 837 GnuMP
171. 835 VesaKarvonen
172. 827 SXMLSimplify
173. 827 LineDirective
174. 813 SSA2Simplify
175. 792 RunningOnSparc
176. 787 Documentation
177. 784 WesleyTerpstra
178. 784 TalkMLtonHistory
179. 784 XMLSimplifyTypes
180. 757 GenerativeDatatype
181. 757 Inline
182. 751 Profiling
183. 736 EtaExpansion
184. 722 Drawbacks
185. 720 Contify
186. 718 Reachability
187. 717 RemoveUnused
188. 717 RayRacine
189. 686 RayRacine/MoinEditorBackup
190. 686 IntroduceLoops
191. 682 GarbageCollection
192. 678 Redundant
193. 677 RSSASimplify
194. 670 AST
195. 660 SureshJagannathan/MoinEditorBackup
196. 659 RunningOnSolaris
197. 654 EditingPages
198. 645 CompilerPassTemplate
199. 644 ToSSA2

200. 617 OpenGL
201. 614 StandardMLBooks
202. 613 FrontEnd
203. 612 Swerve
204. 607 XMLSimplify
205. 596 OtherSites
206. 590 DefinitionOfStandardML
207. 589 Home
208. 576 Machine
209. 571 ForeignFunctionInterface
210. 562 Developers
211. 561 RedundantTests
212. 560 MLKit
213. 555 VesaKarvonen/MoinEditorBackup
214. 551 Multi
215. 551 SureshJagannathan
216. 543 Bug
217. 533 Experimental
218. 523 LLVM
219. 521 IntermediateLanguage
220. 520 CreatingPages
221. 507 StandardMLImplementations
222. 505 SMLNJ
223. 503 ClosureConvert
224. 501 DeepFlatten
225. 495 AccessControl
226. 494 PhantomType
227. 485 mGTK
228. 480 MichaelNorrish
229. 479 Chunkify
230. 477 PackedRepresentation
231. 463 LoopInvariant
232. 462 CoreML
233. 456 Bugs20051202
234. 456 MLtonItimer
235. 453 RSSAShrink
236. 438 LocalFlatten
237. 433 Globalize
238. 432 PlatformSpecificNotes
239. 431 ConstantPropagation
240. 421 Poplog
241. 412 StandardMLTutorials
242. 410 AndreiFormiga/MoinEditorBackup
243. 401 RunningOnPowerPC
244. 400 TalkStandardML
245. 400 RecentChanges
246. 397 TalkMLtonFeatures
247. 395 RunningOnOpenBSD
248. 387 MatthewFluet
249. 386 Polyvariance
250. 382 PrecedenceParse

251. 376 Eclipse
252. 376 RunningOnFreeBSD
253. 374 MLtonWord
254. 370 MLtonProcEnv
255. 369 TalkWholeProgram
256. 367 ParallelMove
257. 364 ObjectOrientedProgramming
258. 364 OperatorPrecedence
259. 361 ToRSSA
260. 359 Development
261. 358 AllocateRegisters
262. 347 SXMLShrink
263. 340 MatchCompilation
264. 329 ImperativeTypeVariable
265. 324 ImplementHandlers
266. 323 MLtonVector
267. 319 MLtonArray
268. 313 CMinusMinus
269. 309 AndreiFormiga
270. 306 TalkMLtonApproach
271. 302 ImplementProfiling
272. 295 ImplementExceptions
273. 284 InsertSignalChecks
274. 281 InsertLimitChecks
275. 277 TalkFolkLore
276. 270 TalkHowHigherOrder
277. 266 Pronounce
278. 265 ML
279. 263 Subversion
280. 261 MLj
281. 257 SMLNET
282. 254 StandardMLHistory
283. 244 TalkFromSMLTo
284. 242 HaMLet
285. 242 TalkHowPolymorphism
286. 238 RunningOnDarwin
287. 234 eXene
288. 230 Immutable
289. 230 TalkHowModules
290. 220 CoreMLSimplify
291. 218 MoscowML
292. 215 Roadmap
293. 209 Glade
294. 209 fxp
295. 200 Talk
296. 196 MoinMoin
297. 192 HenryCejtin
298. 190 Variant
299. 188 TomMurphy
300. 184 SpaceSafety
301. 177 Serialization

302. 176 TalkDiveIn
303. 176 StephenWeeks
304. 175 JohnnyAndersen
305. 165 Mutable
306. 152 ZZZOrphanedPages
307. 152 StyleGuide
308. 145 TemporaryUpload
309. 141 TrustedGroup
310. 136 Survey
311. 136 TalkTemplate
312. 129 PolyML
313. 127 LambdaCalculus
314. 125 WantedPages
315. 120 WikiName
316. 118 Alice
317. 112 MikeThomas
318. 92 TILT
319. 86 OldPages
320. 78 OrphanedPages
321. 73 MLtonTextIO
322. 72 MLtonBinIO
323. 69 PageSize
324. 46 RunningOnLinux
325. 45 SurveyDone
326. 30 SystemInfo
327. 15 Index
328. 1 ChrisClearwater

(last modified 2004-10-25 16:35:07)

ParallelMove

ParallelMove is a rewrite pass, agnostic in the IntermediateLanguage which it produces.

Description

This function computes a sequence of individual moves to effect a parallel move (with possibly overlapping froms and tos).

Implementation

parallel-move.sig parallel-move.fun

Details and Notes

Last edited on 2005-12-01 23:25:40 by StephenWeeks.

Performance

This page compares the performance of a number of SML compilers on a range of benchmarks. For a performance comparison of many different languages, including MLton, see the Computer Language Shootout.

This page compares the following SML compiler versions.

- MLton 20051202
- ML Kit 4.1.4
- Moscow ML 2.00
- Poly/ML 4.1.3
- SML/NJ 110.57

There are tables for run time, code size, and compile time.

Setup

All benchmarks were compiled and run on a 2.6 GHz Celeron with 2G of RAM. The benchmarks were compiled with the default settings for all the compilers, except for Moscow ML, which was passed the `-orthodox -standalone -toplevel` switches. The Poly/ML executables were produced by `use`'ing the file, followed by a `PolyML.commit`. The SML/NJ executables were produced by wrapping the entire program in a `local` declaration whose body performs an `SMLofNJ.exportFn`.

For more details, or if you want to run the benchmarks yourself, please see the benchmark directory of our Sources.

All of the benchmarks are available for download from this page. Some of the benchmarks were obtained from the SML/NJ benchmark suite. Some of the benchmarks expect certain input files to exist in the DATA subdirectory.

- hamlet.sml (hamlet-input.sml)
- ray.sml (ray)
- raytrace.sml (chess.gml)
- vliw.sml (ndotprod.s)

Run-time ratio

The following table gives the ratio of the run time of each benchmark when compiled by another compiler to the run time when compiled by MLton. That is, the larger the number, the slower the generated code runs. A number larger than one indicates that the corresponding compiler produces code that runs more slowly than MLton. If an entry is *, that means that the corresponding compiler failed to compile the benchmark or that the benchmark failed to run.

benchmark	MLton	ML-Kit	MosML	Poly/ML	SML/NJ
barnes-hut	1.0	*	*	*	1.6
boyer	1.0	*	10.1	1.9	3.1
checksum	1.0	*	*	*	*

count-graphs	1.0	7.3	60.7	4.2	3.8
DLXSimulator	1.0	*	*	*	*
fft	1.0	1.2	*	24.2	0.8
fib	1.0	0.9	5.0	1.2	1.3
flat-array	1.0	2.2	35.0	1041.6	13.4
hamlet	1.0	*	*	*	3.1
imp-for	1.0	2.8	63.0	5.1	5.6
knuth-bendix	1.0	*	19.8	4.8	4.6
lexgen	1.0	2.5	5.0	1.7	1.5
life	1.0	1.7	30.6	7.7	1.4
logic	1.0	*	9.4	1.2	2.1
mandelbrot	1.0	4.2	34.0	51.1	1.3
matrix-multiply	1.0	8.3	42.5	13.2	5.3
md5	1.0	*	*	*	*
merge	1.0	*	*	1.1	7.9
mlyacc	1.0	1.5	8.2	1.2	2.2
model-elimination	1.0	*	*	*	2.6
mpuz	1.0	2.3	78.2	4.6	4.1
nucleic	1.0	*	*	23.5	0.8
output1	1.0	30.7	61.4	16.2	14.4
peek	1.0	15.2	176.9	17.9	11.3
psdes-random	1.0	5.0	*	*	2.7
ratio-regions	1.0	2.0	34.7	2.1	5.4
ray	1.0	*	14.8	22.3	0.8
raytrace	1.0	*	*	*	3.3
simple	1.0	1.7	19.3	7.3	2.4
smith-normal-form	1.0	*	*	*	>1000
tailfib	1.0	1.0	51.9	3.2	1.4
tak	1.0	1.2	17.0	1.3	2.0
tensor	1.0	*	*	*	7.4
tsp	1.0	3.4	31.8	*	17.7
tyan	1.0	*	15.7	1.0	1.6
vector-concat	1.0	1.2	20.4	2.0	20.4
vector-rev	1.0	2.2	41.9	2.3	152.4
vliw	1.0	*	*	*	2.5
wc-input1	1.0	11.1	*	7.5	17.2
wc-scanStream	1.0	22.1	*	203.7	11.5
zebra	1.0	3.9	30.2	3.4	8.5
zern	1.0	*	*	*	2.6

Note: for SML/NJ, the smith-normal-form benchmark was killed after running for over 25,000 seconds.

Code size

The following table gives the code size of each benchmark in bytes. The size for MLton and the ML Kit is the sum of text and data for the standalone executable as reported by `size`. The size for Moscow ML is the size in bytes of the executable `a.out`. The size for Poly/ML is the difference in size of the database before the session start and after the commit. The size for SML/NJ is the size of the heap file created by `exportFn` and does not include the size of the SML/NJ runtime system (approximately 100K). A * in an entry means that the compiler failed to compile the benchmark.

benchmark	MLton	ML-Kit	MosML	Poly/ML	SML/NJ
barnes-hut	103,231	*	*	*	433,216
boyer	138,518	163,204	116,300	122,880	526,376
checksum	52,794	*	*	*	*
count-graphs	66,838	84,124	84,613	98,304	454,776
DLXSimulator	129,398	*	*	*	*
fft	64,797	80,240	84,046	65,536	434,256
fib	47,738	18,588	79,892	49,152	415,488
flat-array	47,762	23,820	80,034	49,152	410,680
hamlet	1,256,813	*	*	*	1,412,360
imp-for	47,626	19,372	80,040	57,344	400,424
knuth-bendix	109,126	93,400	88,439	180,224	431,144
lexgen	203,559	208,332	104,883	196,608	501,824
life	66,130	78,084	83,390	65,536	414,760
logic	106,614	116,880	87,251	114,688	440,360
mandelbrot	47,690	77,004	81,340	57,344	404,520
matrix-multiply	49,181	87,016	82,417	57,344	435,256
md5	77,646	*	*	*	*
merge	49,318	24,296	80,090	49,152	400,432
mlyacc	507,431	473,748	148,286	2,850,816	820,336
model-elimination	638,084	*	*	*	1,009,880
mpuz	50,594	73,232	82,382	81,920	408,616
nucleic	199,181	258,552	*	221,184	487,480
output1	80,720	63,336	80,187	49,152	399,400
peek	76,302	62,092	81,621	57,344	403,544
psdes-random	48,402	25,196	*	*	421,944
ratio-regions	73,914	95,924	87,482	73,728	443,448
ray	183,243	108,848	89,859	147,456	493,712
raytrace	265,332	*	*	*	636,112
simple	222,914	192,032	94,396	475,136	756,840
smith-normal-form	181,686	*	*	131,072	558,224
tailfib	47,434	18,804	79,943	57,344	399,400
tak	47,818	18,580	79,908	57,344	411,392
tensor	97,677	*	*	*	450,672

tsp	82,190	97,716	86,146	*	425,024
tyan	134,910	137,800	91,586	196,608	477,272
vector-concat	49,018	23,924	80,194	49,152	410,680
vector-rev	48,246	24,104	80,078	57,344	410,680
vliw	393,762	*	*	*	731,304
wc-input1	101,850	129,212	85,771	49,152	404,520
wc-scanStream	109,106	129,708	85,947	49,152	405,544
zebra	141,146	41,532	83,422	90,112	419,896
zern	91,087	*	*	*	479,384

Compile time

The following table gives the compile time of each benchmark in seconds. A * in an entry means that the compiler failed to compile the benchmark.

benchmark	MLton	ML-Kit	MosML	Poly/ML	SML/NJ
barnes-hut	8.28	*	*	*	1.37
boyer	8.14	8.99	0.39	0.12	3.20
checksum	5.45	*	*	*	*
count-graphs	6.12	2.06	0.14	0.05	0.90
DLXSimulator	9.81	*	*	*	*
fft	5.95	1.32	0.11	0.05	0.69
fib	5.45	0.60	0.05	0.02	0.22
flat-array	5.33	0.61	0.04	0.01	0.25
hamlet	85.70	*	*	*	88.87
imp-for	5.37	0.73	0.05	0.01	0.25
knuth-bendix	7.09	4.11	0.19	0.12	1.60
lexgen	11.02	7.21	0.40	0.26	3.63
life	5.84	2.16	0.10	0.04	0.64
logic	7.02	4.82	0.22	0.09	1.68
mandelbrot	5.41	0.75	0.06	0.02	0.29
matrix-multiply	5.39	0.77	0.06	0.01	0.30
md5	6.01	*	*	*	*
merge	5.41	0.62	0.06	0.02	0.26
mlyacc	24.70	40.69	3.35	1.08	18.04
model-elimination	25.04	*	*	*	28.79
mpuz	5.41	1.07	0.07	0.03	0.45
nucleic	14.24	24.79	*	0.36	2.78
output1	6.05	0.68	0.05	0.01	0.23
peek	6.04	0.70	0.05	0.02	0.25
psdes-random	5.39	0.75	*	*	64.13
ratio-regions	6.63	4.02	0.21	0.11	1.50

ray	9.51	3.02	0.15	0.08	1.03
raytrace	13.92	*	*	*	5.08
simple	11.40	13.19	0.43	0.21	3.76
smith-normal-form	8.90	*	*	0.10	2.25
tailfib	5.35	0.64	0.05	0.02	0.24
tak	5.36	0.62	0.05	0.01	0.22
tensor	8.75	*	*	*	2.81
tsp	6.50	1.93	0.15	*	0.66
tyan	8.86	6.25	0.30	0.17	2.28
vector-concat	5.52	0.68	0.05	0.01	0.25
vector-rev	5.33	0.64	0.05	0.02	0.26
vliw	18.28	*	*	*	13.12
wc-input1	6.85	0.68	0.07	0.02	0.27
wc-scanStream	7.07	0.69	0.06	0.02	0.29
zebra	8.57	2.30	0.09	0.04	0.78
zern	6.20	*	*	*	0.65

Last edited on 2006-09-17 05:14:27 by StephenWeeks.

PhantomType

A phantom type is a type that has no run-time representation, but is used to force the type checker to ensure invariants at compile time. This is done by augmenting a type with additional arguments (phantom type variables) and expressing constraints by choosing phantom types to stand for the phantom types in the types of values.

References

- Blume01
    - dimensions
    - C type system
- FluetPucella02
    - subtyping
- socket module in Basis Library

Last edited on 2005-12-02 04:23:48 by StephenWeeks.

PlatformSpecificNotes

Here are notes about using MLton on the following platforms.

Operating Systems

- Cygwin
- Darwin
- FreeBSD
- Linux
- MinGW
- NetBSD
- OpenBSD
- Solaris

Architectures

- PowerPC
- Sparc

Also see

- PortingMLton

Last edited on 2005-12-01 23:27:45 by StephenWeeks.

PolyEqual

PolyEqual is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This pass implements polymorphic equality.

Implementation

poly-equal.sig poly-equal.fun

Details and Notes

For each datatype, tycon, and vector type, it builds and equality function and translates calls to `MLton_equal` into calls to that function.

Also generates calls to `IntInf_equal` and `Word_equal`.

For tuples, it does the equality test inline; i.e., it does not create a separate equality function for each tuple type.

All equality functions are created only if necesary, i.e., if equality is actually used at a type.

Optimizations:

- for datatypes that are enumerations, do not build a case dispatch, just use `MLton_eq`, as the backend will represent these as ints
- deep equality always does an `MLton_eq` test first

Last edited on 2005-12-01 23:28:42 by StephenWeeks.

PolyML

Poly/ML is a Standard ML Compiler.

Also see

- Matthews95

Last edited on 2006-09-04 20:25:25 by StephenWeeks.

PolymorphicEquality

Polymorphic equality is a built-in function in Standard ML that compares two values of the same type for equality. It is specified as

```
val = : ''a * ''a -> bool
```

The `''a` in the specification are equality type variables, and indicate that polymorphic equality can only be applied to values of an equality type. It is not allowed in SML to rebind =, so a programmer is guaranteed that = always denotes polymorphic equality.

1. Equality of ground types
2. Equality of reals
3. Equality of functions
4. Equality of immutable types
5. Equality of mutable values
6. Equality of datatypes
7. Implementation
8. Also see

Equality of ground types

Ground types like `char`, `int`, and `word` may be compared (to values of the same type). For example, `13 = 14` is type correct and yields `false`.

Equality of reals

The one ground type that can not be compared is `real`. So, `13.0 = 14.0` is not type correct. One can use `Real.==` to compare reals for equality, but beware that this has different algebraic properties than polymorphic equality.

See http://mlton.org/basis/real.html for a discussion of why `real` is not an equality type.

Equality of functions

Comparison of functions is not allowed.

Equality of immutable types

Polymorphic equality can be used on immutable values like tuples, records, lists, and vectors. For example,

```
(1, 2, 3) = (4, 5, 6)
```

is a type-correct expression yielding `false`, while

```
[1, 2, 3] = [1, 2, 3]
```

is type correct and yields `true`.

Equality on immutable values is computed by structure, which means that values are compared by recursively descending the data structure until ground types are reached, at which point the ground types are compared with primitive equality tests (like comparison of characters). So, the expression

```
[1, 2, 3] = [1, 1 + 1, 1 + 1 + 1]
```

is guaranteed to yield `true`, even though the lists may occupy different locations in memory.

Because of structural equality, immutable values can only be compared if their components can be compared. For example, `[1, 2, 3]` can be compared, but `[1.0, 2.0, 3.0]` can not. The SML type system uses equality types to ensure that structural equality is only applied to valid values.

Equality of mutable values

In contrast to immutable values, polymorphic equality of mutable values (like ref cells and arrays) is performed by pointer comparison, not by structure. So, the expression

```
ref 13 = ref 13
```

is guaranteed to yield `false`, even though the ref cells hold the same contents.

Because equality of mutable values is not structural, arrays and refs can be compared *even if their components are not equality types*. Hence, the following expression is type correct (and yields true).

```
let
   val r = ref 13.0
in
   r = r
end
```

Equality of datatypes

Polymorphic equality of datatypes is structural. Two values of the same datatype are equal if they are of the same variant and if the variant's arguments are equal (recursively). So, with the datatype

```
datatype t = A | B of t
```

then `B (B A) = B A` is type correct and yields `false`, while `A = A` and `B A = B A` yield `true`.

As polymorphic equality descends two values to compare them, it uses pointer equality whenever it reaches a mutable value. So, with the datatype

```
datatype t = A of int ref | ...
```

then `A (ref 13) = A (ref 13)` is type correct and yields `false`, because the pointer equality on the two ref cells yields `false`.

One weakness of the SML type system is that datatypes do not inherit the special property of the `ref` and `array` type constructors that allows them to be compared regardless of their component type. For example, after declaring

```
datatype 'a t = A of 'a ref
```

one might expect to be able to compare two values of type `real t`, because pointer comparison on a ref cell would suffice. Unfortunately, the type system can only express that a user-defined datatype admits equality or not. In this case, `t` admits equality, which means that `int t` can be compared but that `real t` can not. We can confirm this with the program

```
datatype 'a t = A of 'a ref
fun f (x: real t, y: real t) = x = y
```

on which MLton reports the following error.

```
Error: z.sml 2.34.
  Function applied to incorrect argument.
    expects: [<equality>] * [<equality>]
    but got: [<non-equality>] * [<non-equality>]
    in: = (x, y)
```

Implementation

Polymorphic equality is implemented by recursively descending the two values being compared, stopping as soon as they are determined to be unequal, or exploring the entire values to determine that they are equal. Hence, polymorphic equality can take time proportional to the size of the smaller value.

MLton uses some optimizations to improve performance.

- When computing structural equality, first do a pointer comparison. If the comparison yields `true`, then stop and return `true`, since the structural comparison is guaranteed to do so. If the pointer comparison fails, then recursively descend the values.
- If a datatype is an enum (e.g. `datatype t = A | B | C`), then a single comparison suffices to compare values of the datatype. No case dispatch is required to determine whether the two values are of the same variant.
- When comparing a known constant non-value-carrying variant, use a single comparison. For example, the following code will compile into a single comparison for `A = x`.

  ```
  datatype t = A | B | C of ...
  ... if A = x then ...
  ```
- When comparing a small constant `IntInf.int` to another `IntInf.int`, use a single comparison against the constant. No case dispatch is required.

Also see

- AdmitsEquality
- EqualityType
- EqualityTypeVariable

Last edited on 2005-12-01 23:31:02 by StephenWeeks.

Polyvariance

Polyvariance is an optimization pass for the SXML IntermediateLanguage, invoked from SXMLSimplify.

Description

This pass duplicates a higher-order, `let` bound function at each variable reference, if the cost is smaller than some threshold.

Implementation

polyvariance.sig polyvariance.fun

Details and Notes

Last edited on 2005-12-01 23:31:34 by StephenWeeks.

Poplog

POPLOG is a development environment that includes implementations of a number of languages, including Standard ML.

While POPLOG is actively developed, the ML support predates SML'97, and there is no support for the BasisLibrary.

Here is a document on Mixed-language programming in ML and Pop-11.

Last edited on 2005-10-09 23:12:14 by StephenWeeks.

PortStatus

This table shows who's working on which port of the x86_64 branch and what the status is.

amd64	Linux	Matthew	(`-m32`) done amd64-linux.log (Fedora Core 4)
HPPA	HP-UX	Ville	runtime compiles
PowerPC	AIX	Ville	
PowerPC	Darwin	Wesley	done
PowerPC	Linux	Wesley	done
Sparc	Solaris	Stephen	done sparc-solaris.log (SunOS 5.8)
x86	Cygwin	Stephen	runtime compiles
x86	Darwin		
x86	FreeBSD	Matthew	done x86-freebsd.log (FreeBSD 6.1)
x86	Linux	Matthew	done x86-linux.log (Red Hat Enterprise Linux AS 4)
x86	MinGW	Wesley	done
x86	NetBSD	Matthew	done x86-netbsd.log (NetBSD 3.0)
x86	OpenBSD	Matthew	done x86-openbsd.log (OpenBSD 3.9)

Logs produced with
`(gmake all-no-docs ; ./bin/regression) 2>&1 | tee x86-netbsd.log`.

Last edited on 2006-07-15 19:09:35 by StephenWeeks.

PortingMLton

Porting MLton to a new target platform (architecture or OS) involves the following steps.

1. Make the necessary changes to the scripts, runtime system, Basis Library implementation, and compiler.
2. Get the regressions working using a cross compiler.
3. Cross compile MLton and bootstrap on the target.

MLton has a native code generator only for X86, so, if you are porting to another architecture, you must use the C code generator. These notes do not cover building a new native code generator.

Some of the following steps will not be necessary if MLton already supports the architecture or operating system you are porting to.

What code to change

- Scripts.
    - In `bin/platform`, add new cases to define `$HOST_OS` and `$HOST_ARCH`.
    - In `bin/upgrade-basis`,
        - add new cases to set `$arch` and `$os`.
        - add new cases in the code for `MLton.Platform` to define `Arch.t`, `OS.t`, `Arch.all`, and `OS.all`.
- Runtime system.

  The goal of this step is to be able to successfully run `make` in the `runtime` directory on the target machine.
    - In `platform.h`, add a new case to include `platform/<os>.h`.
    - In `platform/<os>.h`:
        - include platform-specific includes.
        - define `MLton_Platform_OS_host`.
        - define all of the `HAS_*` macros.
    - In `platform/<os>.c` implement any platform-dependent functions that the runtime needs.
    - In `basis/Real/class.c`, add the architecture specific code to implement `Real<N>.class` (i.e. to determine the class of a floating point number. It would be nice to implement this code (portably) in the Basis Library implementation some day.
    - Add rounding mode control to IEEEReal.c for the new arch.
    - Compile and install the GnuMP. This varies from platform to platform. In `platform/<os>.h`, you need to include the appropriate `gmp.h`.
    - Make sure the definition of `ReturnToC` in `include/x86-main.h` is correct.
- Basis Library implementation (`basis-library/*`)
    - In `misc/primitive.sml`,
        - Add a new variant to the `MLton.Platform.Arch.t` datatype.
        - Add a new variant to the `MLton.Platform.OS.t` datatype.
        - modify the constants that define `host` to match with `MLton_Platform_OS_host`, as set in `runtime/platform/<os>.h`.
    - In `mlton/platform.{sig,sml}` add a new variant.
    - In `sml-nj/sml-nj.sml`, modify `getOSKind`.

- Look at all the uses of `MLton.Platform` in the Basis Library implementation and see if you need to do anything special. You might use the following command to see where to look.

```
find basis-library -type f | xargs grep 'MLton\.Platform'
```

  If in doubt, leave the code alone and wait to see what happens when you run the regression tests. Here's some that will likely need to be modified.
    - `real/pack-real.sml`: definition of isBigEndian
- Compiler.
  - In `lib/mlton-stubs/` run `make links` to ensure that `platform.sig` has the changes made to the basis. Then, update `mlton.sml` to add any new variants in `MLton.Platform`.

The string used to identify a particular architecture or operating system must be the same (except for possibly case of letters) in the scripts, runtime, and basis library. In `mlton/main/main.fun`, MLton itself uses the basis library conversions to and from strings:

```
MLton.Platform.{Arch,OS}.{from,to}String
```

If the there is a mismatch, you may see the error message `strange arch` or `strange os`.

Running the regressions with a cross compiler

When porting to a new platform, it is always best to get all (or as many as possible) of the regressions working before moving to a self compile. It is easiest to do this by modifying and rebuilding the compiler on a working machine and then running the regressions with a cross compiler. It is not easy to build a gcc cross compiler, so we recommend generating the C and assembly on a working machine (using MLton's `-target` and `-stop g` flags, copying the generated files to the target machine, then compiling and linking there.

1. Remake the compiler on a working machine.
2. Use `bin/add-cross` to add support for the new target. In particular, this should create `build/lib/<target>/` with the platform-specific necessary cross-compilation information.
3. Run the regression tests with the cross-compiler. To cross-compile all the tests, do

```
bin/regression -cross <target>
```

   This will create all the executables. Then, copy `bin/regression` and the `regression` directory to the target machine, and do

```
bin/regression -run-only
```

   This should run all the tests.

Repeat this step, interleaved with appropriate compiler modifications, until all the regressions pass.

Bootstrap

Once you've got all the regressions working, you can build MLton for the new target. As with the regressions, the idea for bootstrapping is to generate the C and assembly on a working machine, copy it to the target machine, and then compile and link there. Here's the sequence of steps.

1. On a working machine, with the newly rebuilt compiler, in the `mlton` directory, do:

   ```
   mlton -stop g -target <target> mlton.cm
   ```
2. Copy to the target machine.
3. On the target machine, move the libraries to the right place. That is, in `build/lib`, do:

   ```
   rm -rf self target-map
   mv <target> self
   ```
4. On the target machine, compile and link MLton. That is, in the mlton directory, do something like:

   ```
   gcc -c -Ibuild/lib/include -O1 -w mlton/mlton.*.[cS]
   gcc -o build/lib/mlton-compile \
          -Lbuild/lib/self \
          -L/usr/local/lib \
          mlton.*.o \
          -lmlton -lgmp -lgdtoa -lm
   ```
5. At this point, MLton should be working and you can finish the rest of a usual make on the target machine.

   ```
   make script constants targetmap mlbpathmap world libraries tools install
   ```

There are other details to get right, like making sure that the tools directories were clean so that the tools are rebuilt on the new platform, but hopefully this structure works. Once you've got a compiler on the target machine, you should test it by running all the regressions normally (i.e. without the `-cross` flag) and by running a couple rounds of self compiles.

Also see

The above description is based on the following emails sent to the MLton list.

- http://mlton.org/pipermail/mlton/2002-October/013110.html
- http://mlton.org/pipermail/mlton/2004-July/016029.html

Last edited on 2006-04-30 14:19:01 by VilleLaurikari.

PrecedenceParse

PrecedenceParse is an analysis/rewrite pass for the AST IntermediateLanguage, invoked from Elaborate.

Description

This pass rewrites AST function clauses, expressions, and patterns to resolve OperatorPrecedence.

Implementation

precedence-parse.sig precedence-parse.fun

Details and Notes

Last edited on 2005-12-01 23:34:53 by StephenWeeks.

Printf

Programmers coming from C or Java often ask if Standard ML has a `printf` function. It does not. However, it is possible to implement your own version with only a few lines of code.

Here is a definition for `printf` and `fprintf`, along with format specifiers for booleans, integers, and reals.

```
structure Printf =
   struct
      fun $ (_, f) = f (fn p => p ()) ignore
      fun fprintf out f = f (out, id)
      val printf = fn z => fprintf TextIO.stdOut z
      fun one ((out, f), make) g =
         g (out, fn r =>
            f (fn p =>
               make (fn s =>
                     r (fn () => (p (); TextIO.output (out, s))))))
      fun ` x s = one (x, fn f => f s)
      fun spec to x = one (x, fn f => f o to)
      val B = fn z => spec Bool.toString z
      val I = fn z => spec Int.toString z
      val R = fn z => spec Real.toString z
   end
```

Here's an example use.

```
val () = printf `"Int="I`"  Bool="B`"  Real="R`"\n" $ 1 false 2.0
```

This prints the following.

```
Int=1  Bool=false  Real=2.0
```

In general, a use of `printf` looks like

```
printf <spec1> ... <specn> $ <arg1> ... <argm>
```

where each `<speci>` is either a specifier like `B`, `I`, or `R`, or is an inline string, like `` `"foo" ``. A backtick (`` ` ``) must precede each inline string. Each `<argi>` must be of the appropriate type for the corresponding specifier.

SML `printf` is more powerful than its C counterpart in a number of ways. In particular, the function produced by `printf` is a perfectly ordinary SML function, and can be passed around, used multiple times, etc. For example:

```
val f: int -> bool -> unit = printf `"Int="I`"  Bool="B`"\n" $
val () = f 1 true
val () = f 2 false
```

The definition of `printf` is even careful to not print anything until it is fully applied. So, examples like the following will work as expected.

```
val f: int -> bool -> unit = printf `"Int="I`"  Bool="B`"\n" $ 13
val () = f true
val () = f false
```

It is also easy to define new format specifiers. For example, suppose we wanted format specifiers for characters and strings.

```
val C = fn z => spec Char.toString z
val S = fn z => spec (fn s => s) z
```

One can define format specifiers for more complex types, e.g. pairs of integers.

```
val I2 =
   fn z =>
   spec (fn (i, j) =>
         concat ["(", Int.toString i, ", ", Int.toString j, ")"])
   z
```

Here's an example use.

```
val () = printf `"Test "I2`"  a string "S`"\n" $ (1, 2) "hello"
```

Printf via fold

Printf is best viewed as a special case of variable-argument Fold that inductively builds a function as it processes its arguments. Here is the definition of a `Printf` structure in terms of fold. The structure is equivalent to the above one, except that it uses the standard `$` instead of a specialized one.

```
structure Printf =
   struct
      fun fprintf out =
         Fold.fold ((out, id), fn (_, f) => f (fn p => p ()) ignore)

      val printf = fn z => fprintf TextIO.stdOut z

      fun one ((out, f), make) =
         (out, fn r =>
          f (fn p =>
             make (fn s =>
                   r (fn () => (p (); TextIO.output (out, s))))))

      val ` =
         fn z => Fold.step1 (fn (s, x) => one (x, fn f => f s)) z

      fun spec to = Fold.step0 (fn x => one (x, fn f => f o to))

      val B = fn z => spec Bool.toString z
      val I = fn z => spec Int.toString z
      val R = fn z => spec Real.toString z
   end
```

Viewing `printf` as a fold opens up a number of possibilities. For example, one can name parts of format strings using the fold idiom for naming sequences of steps.

```
val IB = fn u => Fold.fold u `"Int="I`" Bool="B
val () = printf IB`"  "IB`"\n" $ 1 true 3 false
```

One can even parameterize over partial format strings.

```
fun XB X = fn u => Fold.fold u `"X="X`" Bool="B
```

```
val () = printf (XB I)`"  "(XB R)`"\n" $ 1 true 2.0 false
```

Also see

- Functional Unparsing

Last edited on 2006-03-21 22:06:10 by StephenWeeks.

PrintfGentle

This page provides a gentle introduction and derivation of Printf, with sections and arrangement more suitable to a talk.

Introduction

SML does not have `printf`. Could we define it ourselves?

```
val () = printf ("here's an int %d and a real %f.\n", 13, 17.0)
val () = printf ("here's three values (%d, %f, %f).\n", 13, 17.0, 19.0)
```

What could the type of `printf` be?

This obviously can't work, because SML functions take a fixed number of arguments. Actually they take one argument, but if that's a tuple, it can only have a fixed number of components.

From tupling to currying

What about currying to get around the typing problem?

```
val () = printf "here's an int %d and a real %f.\n" 13 17.0
val () = printf "here's three values (%d, %f, %f).\n" 13 17.0 19.0
```

That fails for a similar reason. We need two types for `printf`.

```
val printf: string -> int -> real -> unit
val printf: string -> int -> real -> real -> unit
```

This can't work, because `printf` can only have one type. SML doesn't support programmer-defined overloading.

Overloading and dependent types

Even without worrying about number of arguments, there is another problem. The type of `printf` depends on the format string.

```
val () = printf "here's an int %d and a real %f.\n" 13 17.0
val () = printf "here's a real %f and an int %d.\n" 17.0 13
```

Now we need

```
val printf: string -> int -> real -> unit
val printf: string -> real -> int -> unit
```

Again, this can't possibly working because SML doesn't have overloading, and types can't depend on values.

Idea: express type information in the format string

If we express type information in the format string, then different uses of `printf` can have different types.

```
type 'a t  (* the type of format strings *)
val printf: 'a t -> 'a
infix D F
val fs1: (int -> real -> unit) t = "here's an int "D" and a real "F".\n"
val fs2: (int -> real -> real -> unit) t =
   "here's three values ("D", "F", "F").\n"
val () = printf fs1 13 17.0
val () = printf fs2 13 17.0 19.0
```

Now, our two calls to `printf` type check, because the format string specializes `printf` to the appropriate type.

The types of format characters

What should the type of format characters `D` and `F` be? Each format character requires an additional argument of the appropriate type to be supplied to `printf`.

Idea: guess the final type that will be needed for `printf` the format string and verify it with each format character.

```
type ('a, 'b) t   (* 'a = rest of type to verify, 'b = final type *)
val ` : string -> ('a, 'a) t  (* guess the type, which must be verified *)
val D: (int -> 'a, 'b) t * string -> ('a, 'b) t  (* consume an int *)
val F: (real -> 'a, 'b) t * string -> ('a, 'b) t  (* consume a real *)
val printf: (unit, 'a) t -> 'a
```

Don't worry. In the end, type inference will guess and verify for us.

Understanding guess and verify

Now, let's build up a format string and a specialized `printf`.

```
infix D F
val f0 = `"here's an int "
val f1 = f0 D " and a real "
val f2 = f1 F ".\n"
val p = printf f2
```

These definitions yield the following types.

```
val f0: (int -> real -> unit, int -> real -> unit) t
val f1: (real -> unit, int -> real -> unit) t
val f2: (unit, int -> real -> unit) t
val p: int -> real -> unit
```

So, `p` is a specialized `printf` function. We could use it as follows

```
val () = p 13 17.0
val () = p 14 19.0
```

Type checking this using a functor

```
signature PRINTF =
   sig
      type ('a, 'b) t
      val ` : string -> ('a, 'a) t
      val D: (int -> 'a, 'b) t * string -> ('a, 'b) t
      val F: (real -> 'a, 'b) t * string -> ('a, 'b) t
      val printf: (unit, 'a) t -> 'a
   end

functor Test (P: PRINTF) =
   struct
      open P
      infix D F

      val () = printf (`"here's an int "D" and a real "F".\n") 13 17.0
      val () = printf (`"here's three values ("D", "F ", "F").\n") 13 17.0 19.0
   end
```

Implementing Printf

Think of a format character as a formatter transformer. It takes the formatter for the part of the format string before it and transforms it into a new formatter that first does the left hand bit, then does its bit, then continues on with the rest of the format string.

```
structure Printf: PRINTF =
   struct
      datatype ('a, 'b) t = T of (unit -> 'a) -> 'b

      fun printf (T f) = f (fn () => ())

      fun ` s = T (fn a => (print s; a ()))

      fun D (T f, s) =
         T (fn g => f (fn () => fn i =>
                       (print (Int.toString i); print s; g ())))

      fun F (T f, s) =
         T (fn g => f (fn () => fn i =>
                       (print (Real.toString i); print s; g ())))
   end
```

Testing printf

```
structure Z = Test (Printf)
```

User-definable formats

The definition of the format characters is pretty much the same. Within the `Printf` structure we can define a format character generator.

```
val newFormat: ('a -> string) -> ('a -> 'b, 'c) t * string -> ('b, 'c) t =
   fn toString => fn (T f, s) =>
   T (fn th => f (fn () => fn a => (print (toString a); print s ; th ())))
```

```
val D = fn z => newFormat Int.toString z
val F = fn z => newFormat Real.toString z
```

A core Printf

We can now have a very small PRINTF signature, and define all the format strings externally to the core module.

```
signature PRINTF =
   sig
      type ('a, 'b) t
      val ` : string -> ('a, 'a) t
      val newFormat: ('a -> string) -> ('a -> 'b, 'c) t * string -> ('b, 'c) t
      val printf: (unit, 'a) t -> 'a
   end

structure Printf: PRINTF =
   struct
      datatype ('a, 'b) t = T of (unit -> 'a) -> 'b

      fun printf (T f) = f (fn () => ())

      fun ` s = T (fn a => (print s; a ()))

      fun newFormat toString (T f, s) =
         T (fn th =>
            f (fn () => fn a =>
               (print (toString a)
                ; print s
                ; th ())))
   end
```

Extending to fprintf

One can implement fprintf by threading the outstream through all the transformers.

```
signature PRINTF =
   sig
      type ('a, 'b) t
      val ` : string -> ('a, 'a) t
      val fprintf: (unit, 'a) t * TextIO.outstream -> 'a
      val newFormat: ('a -> string) -> ('a -> 'b, 'c) t * string -> ('b, 'c) t
      val printf: (unit, 'a) t -> 'a
   end

structure Printf: PRINTF =
   struct
      type out = TextIO.outstream
      val output = TextIO.output

      datatype ('a, 'b) t = T of (out -> 'a) -> out -> 'b

      fun fprintf (T f, out) = f (fn _ => ()) out

      fun printf t = fprintf (t, TextIO.stdOut)

      fun ` s = T (fn a => fn out => (output (out, s); a out))
```

```
    fun newFormat toString (T f, s) =
       T (fn g =>
          f (fn out => fn a =>
             (output (out, toString a)
              ; output (out, s)
              ; g out)))
 end
```

Notes

- Lesson: instead of using dependent types for a function, express the the dependency in the type of the argument.
- If `printf` is partially applied, it will do the printing then and there. Perhaps this could be fixed with some kind of terminator.

  A syntactic or argument terminator is not necessary. A formatter can either be eager (as above) or lazy (as below). A lazy formatter accumulates enough state to print the entire string. The simplest lazy formatter concatenates the strings as they become available:

```
structure PrintfLazyConcat: PRINTF =
   struct
      datatype ('a, 'b) t = T of (string -> 'a) -> string -> 'b

      fun printf (T f) = f print ""

      fun ` s = T (fn th => fn s' => th (s' ^ s))

      fun newFormat toString (T f, s) =
         T (fn th =>
            f (fn s' => fn a =>
               th (s' ^ toString a ^ s)))
   end
```

  It is somewhat more efficient to accumulate the strings as a list:

```
structure PrintfLazyList: PRINTF =
   struct
      datatype ('a, 'b) t = T of (string list -> 'a) -> string list -> 'b

      fun printf (T f) = f (List.app print o List.rev) []

      fun ` s = T (fn th => fn ss => th (s::ss))

      fun newFormat toString (T f, s) =
         T (fn th =>
            f (fn ss => fn a =>
               th (s::toString a::ss)))
   end
```

Last edited on 2005-07-13 21:21:04 by VesaKarvonen.

ProductType

Standard ML has special syntax for products (tuples). A product type is written as

```
t1 * t2 * ... * tN
```

and a product pattern is written as

```
(p1, p2, ..., pN)
```

In most situations the syntax is quite convenient. However, there are sitiuations where the syntax is cumbersome. There are also situations in which it is useful to construct and destruct n-ary products inductively, especially when using Fold.

In such situations, it is useful to have a binary product datatype with an infix constructor defined as follows.

```
datatype ('a, 'b) product = & of 'a * 'b
infix &
```

With these definitions, one can write an n-ary product as a nested binary product quite conveniently.

```
x1 & x2 & ... & xn
```

Because of left associativity, this is the same as

```
(((x1 & x2) & ...) & xn)
```

Because `&` is a constructor, the syntax can also be used for patterns.

The symbol `&` is inspired by the Curry-Howard isomorphism: the proof of a conjunction `(A & B)` is a pair of proofs `(a, b)`.

Example: parser combinators

A typical parser combinator library provides a combinator that has a type of the form.

```
'a parser * 'b parser -> ('a * 'b) parser
```

and produces a parser for the concatenation of two parsers. When more than two parsers are concatenated, the result of the resulting parser is a nested structure of pairs

```
(...((p1, p2), p3)..., pN)
```

which is somewhat cumbersome.

By using a product type, the type of the concatenation combinator then becomes

```
'a parser * 'b parser -> ('a, 'b) product parser
```

While this doesn't stop the nesting, it makes the pattern significantly easier to write. Instead of

```
(...((p1, p2), p3)..., pN)
```

the pattern is written as

```
p1 & p2 & p3 & ... & pN
```

which is considerably more concise.

Also see

- VariableArityPolymorphism

Last edited on 2006-03-21 22:04:54 by StephenWeeks.

Profiling

With MLton and `mlprof`, you can profile your program to find out bytes allocated, execution counts, or time spent in each function. To profile you program, compile with `-profile` *kind*, where *kind* is one of `alloc`, `count`, or `time`. Then, run the executable, which will write an `mlmon.out` file when it finishes. You can then run `mlprof` on the executable and the `mlmon.out` file to see the performance data.

Here are the three kinds of profiling that MLton supports.

- ProfilingAllocation
- ProfilingCounts
- ProfilingTime

Going further

- CallGraphs to visualize profiling data.
- ProfilingTheStack
- MLtonProfile to selectively profile parts of your program.
- HowProfilingWorks.

Last edited on 2004-11-01 18:55:47 by StephenWeeks.

ProfilingAllocation

With MLton and `mlprof`, you can profile your program to find out how many bytes each function allocates. To do so, compile your program with `-profile alloc`. For example, suppose that `list-rev.sml` is the following.

```
fun append (l1, l2) =
   case l1 of
      [] => l2
    | x :: l1 => x :: append (l1, l2)

fun rev l =
   case l of
      [] => []
    | x :: l => append (rev l, [x])

val l = List.tabulate (1000, fn i => i)
val _ = 1 + hd (rev l)
```

Compile and run `list-rev` as follows.

```
% mlton -profile alloc list-rev.sml
% ./list-rev
% mlprof -show-line true list-rev mlmon.out
6,030,136 bytes allocated (108,336 bytes by GC)
       function          cur
----------------------- -----
append  list-rev.sml: 1 97.6%
<gc>                     1.8%
<main>                   0.4%
rev  list-rev.sml: 6     0.2%
```

The data shows that most of the allocation is done by the `append` function defined on line 1 of `list-rev.sml`. The table also shows how special functions like `gc` and `main` are handled: they are printed with surrounding brackets. C functions are displayed similarly. In this example, the allocation done by the garbage collector is due to stack growth, which is usually the case.

The run-time performance impact of allocation profiling is noticeable, because it inserts additional C calls for object allocation.

Compile with `-profile alloc -profile-branch true` to find out how much allocation is done in each branch of a function; see ProfilingCounts for more details on `-profile-branch`.

Last edited on 2005-12-02 04:24:10 by StephenWeeks.

ProfilingCounts

With MLton and `mlprof`, you can profile your program to find out how many times each function is called and how many times each branch is taken. To do so, compile your program with

```
-profile
count -profile-branch true
```

. For example, suppose that `tak.sml`contains the following.

```
structure Tak =
   struct
      fun tak1 (x, y, z) =
         let
            fun tak2 (x, y, z) =
               if y >= x
                  then z
               else
                  tak1 (tak2 (x - 1, y, z),
                        tak2 (y - 1, z, x),
                        tak2 (z - 1, x, y))
         in
            if y >= x
               then z
            else
               tak1 (tak2 (x - 1, y, z),
                     tak2 (y - 1, z, x),
                     tak2 (z - 1, x, y))
         end
   end

val rec f =
   fn 0 => ()
    | ~1 => print "this branch is not taken\n"
    | n => (Tak.tak1 (18, 12, 6) ; f (n-1))

val _ = f 5000

fun uncalled () = ()
```

Compile with count profiling and run the program.

```
% mlton -profile count -profile-branch true tak.sml
% ./tak
```

Display the profiling data, along with raw counts and file positions.

```
% mlprof -raw true -show-line true tak mlmon.out
623,610,002 ticks
            function                cur       raw
--------------------------------- ----- -------------
Tak.tak1.tak2  tak.sml: 5         38.2% (238,530,000)
Tak.tak1.tak2.<true>  tak.sml: 7  27.5% (171,510,000)
Tak.tak1  tak.sml: 3              10.7%  (67,025,000)
Tak.tak1.<true>  tak.sml: 14      10.7%  (67,025,000)
Tak.tak1.tak2.<false>  tak.sml: 9 10.7%  (67,020,000)
Tak.tak1.<false>  tak.sml: 16      2.0%  (12,490,000)
```

```
f  tak.sml: 23                       0.0%       (5,001)
f.<branch>  tak.sml: 25              0.0%       (5,000)
f.<branch>  tak.sml: 23              0.0%           (1)
uncalled  tak.sml: 29                0.0%           (0)
f.<branch>  tak.sml: 24              0.0%           (0)
```

Branches are displayed with lexical nesting followed by `<branch>` where the function name would normally be, or `<true>` or `<false>` for if-expressions. It is best to run `mlprof` with `-show-line true` to help identify the branch.

One use of `-profile count` is as a code-coverage tool, to help find code in your program that hasn't been tested. For this reason, `mlprof` displays functions and branches even if they have a count of zero. As the above output shows, the branch on line 24 was never taken and the function defined on line 29 was never called. To see zero counts, it is best to run `mlprof` with `-raw true`, since some code (e.g. the branch on line 23 above) will show up with `0.0%` but may still have been executed and hence have a nonzero raw count.

Last edited on 2005-12-02 04:24:22 by StephenWeeks.

ProfilingTheStack

For all forms of Profiling, you can gather counts for all functions on the stack, not just the currently executing function. To do so, compile your program with `-profile-stack true`. For example, suppose that `list-rev.sml` contains the following.

```
fun append (l1, l2) =
   case l1 of
      [] => l2
    | x :: l1 => x :: append (l1, l2)

fun rev l =
   case l of
      [] => []
    | x :: l => append (rev l, [x])

val l = List.tabulate (1000, fn i => i)
val _ = 1 + hd (rev l)
```

Compile with stack profiling and then run the program.

```
% mlton -profile alloc -profile-stack true list-rev.sml
% ./list-rev
```

Display the profiling data.

```
% mlprof -show-line true list-rev mlmon.out
6,030,136 bytes allocated (108,336 bytes by GC)
       function          cur  stack  GC
----------------------- ----- ----- ----
append  list-rev.sml: 1 97.6% 97.6% 1.4%
<gc>                     1.8%  0.0% 1.8%
<main>                   0.4% 98.2% 1.8%
rev  list-rev.sml: 6     0.2% 97.6% 1.8%
```

In the above table, we see that `rev`, defined on line 6 of `list-rev.sml`, is only responsible for 0.2% of the allocation, but is on the stack while 97.6% of the allocation is done by the user program and while 1.8% of the allocation is done by the garbage collector.

The run-time performance impact of `-profile-stack true` can be noticeable since there is some extra bookkeeping at every nontail call and return.

Last edited on 2005-12-02 00:47:48 by StephenWeeks.

ProfilingTime

With MLton and `mlprof`, you can profile your program to find out how much time is spent in each function over an entire run of the program. To do so, compile your program with `-profile time`. For example, suppose that `tak.sml` contains the following.

```
structure Tak =
   struct
      fun tak1 (x, y, z) =
         let
            fun tak2 (x, y, z) =
               if y >= x
                  then z
               else
                  tak1 (tak2 (x - 1, y, z),
                        tak2 (y - 1, z, x),
                        tak2 (z - 1, x, y))
         in
            if y >= x
               then z
            else
               tak1 (tak2 (x - 1, y, z),
                     tak2 (y - 1, z, x),
                     tak2 (z - 1, x, y))
         end
   end

val rec f =
   fn 0 => ()
    | ~1 => print "this branch is not taken\n"
    | n => (Tak.tak1 (18, 12, 6) ; f (n-1))

val _ = f 5000

fun uncalled () = ()
```

Compile with time profiling and run the program.

```
% mlton -profile time tak.sml
% ./tak
```

Display the profiling data.

```
% mlprof tak mlmon.out
6.00 seconds of CPU time (0.00 seconds GC)
function     cur
------------- -----
Tak.tak1.tak2 75.8%
Tak.tak1      24.2%
```

This example shows how `mlprof` indicates lexical nesting: as a sequence of period-separated names indicating the structures and functions in which a function definition is nested. The profiling data shows that roughly three-quarters of the time is spent in the `Tak.tak1.tak2` function, while the rest is spent in `Tak.tak1`.

Display raw counts in addition to percentages with `-raw true`.

```
% mlprof -raw true tak mlmon.out
6.00 seconds of CPU time (0.00 seconds GC)
  function     cur    raw
------------- ----- -------
Tak.tak1.tak2 75.8% (4.55s)
Tak.tak1      24.2% (1.45s)
```

Display the file name and line number for each function in addition to its name with `-show-line true`.

```
% mlprof -show-line true tak mlmon.out
6.00 seconds of CPU time (0.00 seconds GC)
        function           cur
------------------------- -----
Tak.tak1.tak2  tak.sml: 5 75.8%
Tak.tak1  tak.sml: 3      24.2%
```

Time profiling is designed to have a very small performance impact. However, in some cases there will be a run-time performance cost, which may perturb the results. There is more likely to be an impact with `-codegen c` than `-codegen native`.

You can also compile with `-profile time -profile-branch true` to find out how much time is spent in each branch of a function; see ProfilingCounts for more details on `-profile-branch`.

Caveats

With `-profile time`, use of the following in your program will cause a run-time error, since they would interfere with the profiler signal handler.

- `MLton.Itimer.set (MLton.Itimer.Prof, ...)`
- `MLton.Signal.setHandler (MLton.Signal.prof, ...)`

Also, because of the random sampling used to implement `-profile time`, it is best to have a long running program (at least tens of seconds) in order to get reasonable time

---

Last edited on 2005-12-02 00:51:55 by StephenWeeks.

Projects

We have lots of ideas for projects to improve MLton, many of which we do not have time to implement, or at least haven't started on yet. Here is a list of some of those improvements, ranging from the easy (1 week) to the difficult (several months). If you have any interest in working on one of these, or some other improvement to MLton not listed here, please send mail to MLton@mlton.org.

- Port to new platform: Windows (native, not Cygwin or MinGW), ...
- Source-level debugger
- Heap profiler
- Interfaces to libraries: OpenGL, ...
- Additional constant types: Real80, ...
- An IDE (possibly integrated with Eclipse)
- Port MLRISC and use for code generation
- Optimizations
    - Improved closure representation
      Right now, MLton's closure conversion algorithm uses a simple flat closure to represent each function.
    - Elimination of array bounds checks in loops
    - Elimination of overflow checks on array index computations
    - Common-subexpression elimination of repeated array subscripts
    - Loop-invariant code motion, especially for tuple selects
    - Loop unrolling, especially for small loops
    - Auto-vectorization, for MMX/SSE/3DNow/AltiVec (see the work done on GCC)
    - Optimize `MLton_eq`: pointer equality is necessarily false when one of the arguments is freshly allocated in the block
- Analyses
    - Uncaught exception analysis

Last edited on 2006-06-22 02:51:58 by MatthewFluet.

Pronounce

Here is how "MLton" sounds.

"MLton" is pronounced in two syllables, with stress on the first syllable. The first syllable sounds like the word *mill* (as in "steel mill"), the second like the word *tin* (as in "cookie tin").

Last edited on 2005-12-02 00:54:13 by StephenWeeks.

PropertyList

A property list is a dictionary-like data structure into which properties (name-value pairs) can be inserted and from which properties can be looked up by name. The term comes from the Lisp language, where every symbol has a property list for storing information, and where the names are typically symbols and keys can be any type of value.

Here is an SML signature for property lists such that for any type of value a new property can be dynamically created to manipulate that type of value in a property list.

```
signature PROPERTY_LIST =
   sig
      type t

      val new: unit -> t
      val newProperty: unit -> {add: t * 'a -> unit,
                                peek: t -> 'a option}
   end
```

Here is a functor demonstrating the use of property lists. It first creates a property list, then two new properties (of different types), and adds a value to the list for each property.

```
functor Test (P: PROPERTY_LIST) =
   struct
      val pl = P.new ()

      val {add = addInt: P.t * int -> unit, peek = peekInt} = P.newProperty ()
      val {add = addReal: P.t * real -> unit, peek = peekReal} = P.newProperty ()

      val () = addInt (pl, 13)
      val () = addReal (pl, 17.0)
      val s1 = Int.toString (valOf (peekInt pl))
      val s2 = Real.toString (valOf (peekReal pl))
      val () = print (concat [s1, " ", s2, "\n"])
   end
```

Applied to an appropriate implementation `PROPERTY_LIST`, the `Test` functor will produce the following output.

```
13 17.0
```

Implementation

Because property lists can hold values of any type, their implementation requires a UniversalType. Given that, a property list is simply a list of elements of the universal type. Adding a property adds to the front of the list, and looking up a property scans the list.

```
functor PropertyList (U: UNIVERSAL_TYPE): PROPERTY_LIST =
   struct
      datatype t = T of U.t list ref

      fun new () = T (ref [])

      fun 'a newProperty () =
         let
```

```
            val (inject, out) = U.embed ()
            fun add (T r, a: 'a): unit = r := inject a :: (!r)
            fun peek (T r) =
               Option.map (valOf o out) (List.find (isSome o out) (!r))
         in
            {add = add, peek = peek}
         end
   end
```

If `U: UNIVERSAL_TYPE`, then we can test our code as follows.

```
structure Z = Test (PropertyList (U))
```

Of course, a serious implementation of property lists would have to handle duplicate insertions of the same property, as well as the removal of elements in order to avoid space leaks.

Also see

MLton relies heavily on property lists for attaching information to syntax tree nodes in its intermediate languages. See property-list.sig property-list.fun .

MLRISC uses property lists extensively.

Last edited on 2005-08-19 15:30:27 by MatthewFluet.

RSSA

RSSA is an IntermediateLanguage, translated from SSA2 by ToRSSA, optimized by RSSASimplify, and translated by ToMachine to Machine.

Description

RSSA is a IntermediateLanguage that makes representation decisions explicit.

Implementation

rssa.sig rssa.fun

Type Checking

The new type language is aimed at expressing bit-level control over layout and associated packing of data representations. There are singleton types that denote constants, other atomic types for things like integers and reals, and arbitrary sum types and sequence (tuple) types. The big change to the type system is that type checking is now based on subtyping, not type equality. So, for example, the singleton type 0xFFFFEEBB whose only inhabitant is the eponymous constant is a subtype of the type Word32.

Details and Notes

SSA is an abbreviation for Static Single Assignment. The RSSA IntermediateLanguage is a variant of SSA.

Last edited on 2005-12-02 03:28:34 by StephenWeeks.

RSSAShrink

RSSAShrink is an optimization pass for the RSSA IntermediateLanguage.

Description

This pass implements a whole family of compile-time reductions, like:

- constant folding, copy propagation
- inline the `Goto` to a block with a unique predecessor

Implementation

shrink.sig shrink.fun
shrink.sig shrink.fun

Details and Notes

Last edited on 2005-12-02 01:05:57 by StephenWeeks.

RSSASimplify

The optimization passes for the RSSA IntermediateLanguage are collected and controlled by the `Backend` functor (backend.sig , backend.fun).

The following optimization pass is implemented:

- RSSAShrink

The following implementation passes are implemented:

- ImplementHandlers
- ImplementProfiling
- InsertLimitChecks
- InsertSignalChecks

The optimization passes can be controlled from the command-line by the options

- `-diag-pass <pass>` -- keep diagnostic info for pass
- `-drop-pass <pass>` -- omit optimization pass
- `-keep-pass <pass>` -- keep the results of pass

Last edited on 2005-12-02 01:07:42 by StephenWeeks.

RayRacine

Using SML in some *Semantic Web* stuff. Anyone interested in similar, please contact me. GreyLensman on #sml on IRC or rracine at this domain adelphia with a dot here net.

Current areas of coding.

1. Pretty solid, high performance Rete implementation - base functionality is complete.
2. N3 parser - mostly complete
3. RDF parser based on fxg - not started.
4. Swerve HTTP server - 1/2 done.
5. SPARQL implementation - not started.
6. Persistent engine based on BerkelyDB - not started.
7. Native implementation of Postgresql protocol - underway, ways to go.
8. I also have a small change to the MLton compiler to add PackWord<N> - changes compile but needs some more work, clean-up and unit tests.

Last edited on 2005-12-02 03:28:00 by StephenWeeks.

Reachability

Reachability is a notion dealing with the graph of heap objects maintained at runtime. Nodes in the graph are heap objects and edges correspond to the pointers between heap objects. As the program runs, it allocates new objects (adds nodes to the graph), and those new objects can contain pointers to other objects (new edges in the graph). If the program uses mutable objects (refs or arrays), it can also change edges in the graph.

At any time, the program has access to some finite set of *root* nodes, and can only ever access nodes that are reachable by following edges from these root nodes. Nodes that are *unreachable* can be garbage collected.

Also see

- MLtonFinalizable
- MLtonWeak

Last edited on 2006-08-21 19:41:42 by StephenWeeks.

Redundant

Redundant is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

???

Implementation

redundant.sig redundant.fun

Details and Notes

The reason Redundant got put in was due to some output of the ClosureConvert pass converter where the environment record, or components of it, were passed around in several places. That may have been more relevant with polyvariant analyses (which are long gone). But it still seems possibly relevant, especially with more aggressive flattening, which should reveal some fields in nested closure records that are redundant.

Last edited on 2005-12-02 00:58:46 by StephenWeeks.

RedundantTests

RedundantTests is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This pass simplifies conditionals whose results are implied by a previous conditional test.

Implementation

redundant-tests.sig redundant-tests.fun

Details and Notes

An additional test will sometimes eliminate the overflow test when adding or subtracting 1. In particular, it will eliminate it in the following cases:

```
if x < y
  then ... x + 1 ...
else ... y - 1 ...
```

Last edited on 2005-12-02 00:59:16 by StephenWeeks.

RefFlatten

Refflatten is an optimization pass for the SSA2 IntermediateLanguage, invoked from SSA2Simplify.

Description

This pass flattens a `ref` cell into its containing object. The idea is to replace, where possible, a type like

```
(int ref * real)
```

with a type like

```
(int[m] * real)
```

where the `[m]` indicates a mutable field of a tuple.

Implementation

ref-flatten.sig ref-flatten.fun

Details and Notes

The savings is obvious, I hope. We avoid an extra heap-allocated object for the `ref`, which in the above case saves two words. We also save the time and code for the extra indirection at each get and set. There are lots of useful data structures (singly-linked and doubly-linked lists, union-find, fibonacci heaps, ...) that I believe we are paying through the nose right now because of the absence of ref flattening.

The idea is to compute for each occurrence of a `ref` type in the program whether or not that `ref` can be represented as an offset of some object (constructor or tuple). As before, a unification-based whole-program with deep abstract values makes sure the analysis is consistent.

The only syntactic part of the analysis that remains is the part that checks that for a variable bound to a value constructed by `Ref_ref`:

- the object allocation is in the same block. This is pretty draconian, and it would be nice to generalize it some day to allow flattening as long as the `ref` allocation and object allocation "line up one-to-one" in the same loop-free chunk of code.
- updates occur in the same block (and hence it is safe-for-space because the containing object is still alive). It would be nice to relax this to allow updates as long as it can be proved that the container is live.

Prevent flattening of `unit refs`.

RefFlatten is safe for space. The idea is to prevent a `ref` being flattened into an object that has a component of unbounded size (other than possibly the `ref` itself) unless we can prove that at each point the `ref` is live, then the containing object is live too. I used a pretty simple approximation to liveness.

Last edited on 2005-12-02 01:01:34 by StephenWeeks.

References

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A

- Compiling with Continuations (addall). ISBN 0521416957. Andrew W. Appel. Cambridge University Press, 1992.
- A Critique of Standard ML. Andrew W. Appel. JFP 1993.
- Shrinking Lambda Expressions in Linear Time. Andrew Appel and Trevor Jim. JFP 1997.
- Modern Compiler Implementation in ML (addall). ISBN 0521582741 Andrew W. Appel. Cambridge University Press, 1998.

B

- Tree Pattern Matching for ML. Marianne Baudinet, David MacQueen. 1985.
  Describes the match compiler used in an early version of SML/NJ.
- Compiling Standard ML to Java Bytecodes. Nick Benton, Andrew Kennedy, and George Russell. ICFP 1998.
- Interlanguage Working Without Tears: Blending SML with Java. Nick Benton and Andrew Kennedy. ICFP 1999.
- Exceptional Syntax. Nick Benton and Andrew Kennedy. JFP 2001.
- Adventures in Interoperability: The SML.NET Experience. Nick Benton, Andrew Kennedy, and Claudio Russo. PPDP 2004.
- Shrinking Reductions in SML.NET. Nick Benton, Andrew Kennedy, Sam Lindley and Claudio Russo. IFL 2004.
  Describes a linear-time implementation of an Appel-Jim shrinker, using a mutable IL, and shows that it yields nice speedups in SML.NET's compile times. There are also benchmarks showing that SML.NET when compiled by MLton runs roughly five times faster than when compiled by SML/NJ.
- Embedded Interpreters. Nick Benton. JFP 2005.
- The Edinburgh SML Library. Dave Berry. University of Edinburgh Technical Report ECS-LFCS-91-148, 1991.
- A semantics for ML concurrency primitives. Dave Berry, Robin Milner, and David N. Turner. POPL 1992.
- Lessons From the Design of a Standard ML Library. Dave Berry. JFP 1993.
- Compiling SML to Java Bytecode. Peter Bertelsen. Master's Thesis, 1998.
- OO Programming styles in ML. Bernard Berthomieu. LAAS Report #2000111, 2000.
- No-Longer-Foreign: Teaching an ML compiler to speak C "natively". Matthias Blume. BABEL 2001.
- Portable library descriptions for Standard ML. Matthias Blume. 2001.
- Destructors, Finalizers, and Synchronization. Hans Boehm. POPL 2003.

Discusses a number of issues in the design of finalizers. Many of the design choices are consistent with MLtonFinalizable.

C

- Flow-directed Closure Conversion for Typed Languages. Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. ESOP 2000.

Describes MLton's closure-conversion algorithm, which translates from its simply-typed higher-order intermediate language to its simply-typed first-order intermediate language.

- A Parallel, Real-Time Garbage Collector. Perry Cheng and Guy E. Blelloch. PLDI 2001.
- QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. Koen Claessen and John Hughes. ICFP 2000.
- Adding Threads to Standard ML. Eric C. Cooper and J. Gregory Morrisett. CMU Technical Report CMU-CS-90-186, 1990.

D

- Functional Unparsing. Olivier Danvy. BRICS Technical Report RS 98-12, 1998.
- Exhancements to eXene. Dustin B. Deboer. Master of Science Thesis, 2005.
  Describes ways to improve widget concurency, handling of input focus, X resources and selections.
- A concurrent, generational garbage collector for a multithreaded implementation of ML. Damien Doligez and Xavier Leroy. POPL 1993.
- Modular Type Classes. Derek Dreyer, Robert Harper, Manuel M.T. Chakravarty. University of Chicago Technical Report TR-2006-03, 2006.
- Extensional Polymorphism. Catherin Dubois, Francois Rouaix, and Pierre Weis. POPL 1995.

An extension of ML that allows the definition of ad-hoc polymorphic functions by inspecting the type of their argument.

E

- Garbage Collection Safety for Region-based Memory Management. Martin Elsman. TLDI 2003.
- Type-Specialized Serialization with Sharing Martin Elsman. University of Copenhagen. IT University Technical Report TR-2004-43, 2004.

F

- The Little MLer (addall). ISBN 026256114X. Matthias Felleisen and Dan Freidman. The MIT Press, 1998.
- Kill-Safe Synchronization Abstractions. Matthew Flatt and Robert Bruce Findler. PLDI 2004.
- Contification Using Dominators. Matthew Fluet and Stephen Weeks. ICFP 2001.
  Describes contification, a generalization of tail-recursion elimination that is an optimization operating on MLton's static single assignment (SSA) intermediate language.
- Phantom Types and Subtyping. Matthew Fluet and Riccardo Pucella. TCS 2002.
- Generic Polymorphism in ML. J. Furuse. JFLA 2001.

The formalism behind G'CAML, which has an approach to ad-hoc polymorphism based on Dubois94, the differences being in how type checking works an an improved compilation approach for typecase that does the matching at compile time, not run time.

G

- A Multi-threaded Higher-order User Interface Toolkit. Emden R. Gansner and John H. Reppy. User Interface Software, 1993.
- The Standard ML Basis Library. (addall) ISBN 0521794781. Emden R. Gansner and John H. Reppy. Cambridge University Press, 2004.

An introduction and overview of the Basis Library, followed by a detailed description of each module. The module descriptions are also available online.

- Region-based Memory Management in Cyclone. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. PLDI 2002.

H

- Combining Region Inference and Garbage Collection. Niels Hallenberg, Martin Elsman, and Mads Tofte. PLDI 2002.
- Introduction to Programming using SML (addall). ISBN 0201398206. Michael R. Hansen, Hans Rischel. Addison-Wesley, 1999.
- On the Practicality and Desirability of Highly-concurrent, Mostly-functional Programming. Carl H. Hauser, David B. Benson. ACSD 2004.
  Describes the use of Concurrent ML in implementing the Ped text editor. Argues that using large numbers of threads and message passing style are is a practical and effective ways of modularizing a program.
- Safe and Flexible Memory Management in Cyclone. Mike Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. University of Maryland Technical Report CS-TR-4514, 2003.
- Compiling HOL4 to Native Code. Joe Hurd. TPHOLs 2004.

Describes a port of HOL from Moscow ML to MLton, the difficulties encountered in compiling large programs, and the speedups achieved (roughly 10x).

I

J

- Garbage Collection: Algorithms for Automatic Memory Management (addall). ISBN 0471941484. Richard Jones. John Wiley & Sons, 1999.

K

- Mistakes and ambiguities in the definition of Standard ML. Stefan Kahrs. University of Edinburgh Technical Report ECS-LFCS-93-257, 1993.

  There are also the addenda published in 1996.
  Describes a number of problems with the 1990 Definition, many of which were fixed in the 1997 Definition.
- Pickler Combinators. Andrew Kennedy. JFP 2004.

L

- Faster Algorithms for Finding Minimal Consistent DFAs. Kevin Lang. 1999.
- mGTK: An SML binding of Gtk+. Ken Larsen and Henning Niss. USENIX Annual Technical Conference, 2004.
- The ZINC experiment: an economical implementation of the ML language. Xavier Leroy. Technical report 117, INRIA, 1990.
  A detailed explanation of the design and implementation of a bytecode compiler and interpreter for ML with a machine model aimed at efficient implementation.

- Polymorphism by name for references and continuations. Xavier Leroy. POPL 1993.
- MLRISC Annotations. Allen Leung and Lal George. 1998.

M

- Asynchronous exceptions in Haskell. Simon Marlow, Simon Peyton Jones, Andy Moran and John Reppy. PLDI 2001.
  An asynchronous exception is a signal that one thread can send to another, and is useful for the receiving thread to treat as an exception so that it can clean up locks or other state relevant to its current context.
  There are a couple of earlier versions of this paper floating around, from August and November 2000. Make sure and get the official version from May 2001 (linked above).
- A Distributed Concurrent Implementation of Standard ML. David Matthews. University of Edinburgh Technical Report ECS-LFCS-91-174, 1991.
- Papers on Poly/ML. David C. J. Matthews. University of Edinburgh Technical Report ECS-LFCS-95-335, 1995.
- That About Wraps it Up: Using FIX to Handle Errors Without Exceptions, and Other Programming Tricks. Bruce J. McAdam. University of Edinburgh Technical Report ECS-LFCS-97-375, 1997.
- A Just-In-Time backend for Moscow ML 2.00 in SML. Bjarke Meier, Kristian NÃ¸rgaard. Masters Thesis, 2003.
  A just-in-time compiler using GNU Lightning, showing a speedup of up to four times over Moscow ML's usual bytecode interpreter.
  The full report is only available in Danish.
- How ML Evolved. Robin Milner. Polymorphism--The ML/LCF/Hope Newsletter, 1983.
- Commentary on Standard ML (online pdf). (addall) ISBN 0262631327. Robin Milner and Mads Tofte. The MIT Press, 1990.
  Introduces and explains the notation and approach used in The Definition of Standard ML.
- The Definition of Standard ML. (addall) ISBN 0262631326. Robin Milner, Mads Tofte, and Robert Harper. The MIT Press, 1990.
  Superseded by The Definition of Standard ML (Revised). Accompanied by the Commentary on Standard ML.
- The Definition of Standard ML (Revised). (addall) ISBN 0262631814. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. The MIT Press, 1997.
  A terse and formal specification of Standard ML's syntax and semantics. Supersedes an older version.
- Principles and a Preliminary Design for ML2000. The ML2000 working group, 1999.
- Automatic Code Generation from Coloured Petri Nets for an Access Control System. Kjeld H. Mortensen. Workshop on Practical Use of Coloured Petri Nets and Design/CPN, 1999.
- Procs and locks: a portable multiprocessing platform for Standard ML of New Jersey. J. Gregory Morrisett and Andrew Tolmach. PPoPP 1993.
- ML Grid Programming with ConCert. Tom Murphy VII. ML 2006.

N

- fxp - Processing Structured Documents in SML. Andreas Neumann. Scottish Functional Programming Workshop, 1999.
  Describes fxp, an XML parser implemented in Standard ML.
- Parsing and Querying XML Documents in SML. Andreas Neumann. Doctoral Thesis, 1999.

O

- Purely Functional Data Structures. ISBN 0521663504. Chris Okasaki. Cambridge University Press, 1999.

P

- ML For the Working Programmer (addall) ISBN 052156543X. Larry C. Paulson. Cambridge University Press, 1996.
- The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation. Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson. FLOPS 2002.
  Describes a native x86 Erlang compiler and a comparison of many different native x86 compilers (including MLton) and their register usage and call stack implementations.
- Reactive Programming in Standard ML. Riccardo R. Puccella. 1998. ICCL 1998.

Q

R

- Concurrent Programming in ML. Norman Ramsey. Princeton University Technical Report CS-TR-262-90, 1990.
- Embedding an Interpreted Language Using Higher-Order Functions and Types. Norman Ramsey. IVME 2003.
- Widening Integer Arithmetic. Kevin Redwine and Norman Ramsey. CC 2004.
  Describes a method to implement numeric types and operations (like `Int31` or `Word17`) for sizes smaller than that provided by the processor.
- Synchronous Operations as First-Class Values. John Reppy. PLDI 1988.
- Concurrent Programming in ML (addall). ISBN 0521480892. John Reppy. Cambridge University Press, 1999.
  Covers ConcurrentML.
- Definitional Interpreters Revisited. John C. Reynolds. HOSC 1998.
- Definitional Interpreters for Higher-Order Programming Languages John C. Reynolds. HOSC 1998.
- Defects in the Revised Definition of Standard ML. Andreas Rossberg. 2001.

S

- Dual-Mode Garbage Collection. Patrick M. Sansom. Workshop on the Parallel Implementation of Functional Languages, 1991.
- When Do Match-Compilation Heuristics Matter. Kevin Scott and Norman Ramsey. University of Virginia Technical Report CS-2000-13, 2000.
  Modified SML/NJ to experimentally compare a number of match-compilation heuristics and showed that choice of heuristic usually does not significantly affect code size or run time.
- ML pattern match compilation and partial evaluation. Peter Sestoft. Partial Evaluation, 1996.
  Describes the derivation of the match compiler used in Moscow ML.
- Anthony L. Shipman. Unix System Programming with Standard ML, 2002.
- Calcul statique des applications de modules parametres. Julien Signoles. JFLA 2003.
  Describes a defunctorizer for OCaml, and compares it to existing defunctorizers, including MLton.
- A Separate Compilation Extension to Standard ML. David Swasey, Tom Murphy VII, Karl Crary and Robert Harper ML 2006.

T

- No Assembly Required: Compiling Standard ML to C. David Tarditi, Peter Lee, and Anurag Acharya. 1990.
- Object-oriented programming and Standard ML. Lars Thorup and Mads Tofte. ML, 1994.
- Type Inference for Polymorphic References. Mads Tofte. Information and Computation, 89(1), 1990.
- Combining Closure Conversion with Closure Analysis using Algebraic Types. Andrew Tolmach. TIC 1997.
  Describes a closure-conversion algorithm for a monomorphic IL. The algorithm uses a unification-based flow analysis followed by defunctionalization and is similar to the approach used in MLton CejtinEtAl00.
- From ML to Ada: Strongly-typed Language Interoperability via Source Translation. Andrew Tolmach and Dino Oliva. JFP 1998.

Describes a compiler for RML, a core SML-like language. The compiler is similar in structure to MLton, using monomorphisation, defunctionalization, and optimization on a first-order IL.

U

- Elements of ML Programming (addall). ISBN 0137903871. Jeffrey D. Ullman. Prentice-Hall, 1998.

V

W

- Managing Memory with Types. Daniel C. Wang. PhD Thesis.
  Chapter 6 describes an implementation of a type-preserving garbage collector for MLton.
- Type-Preserving Garbage Collectors. Daniel C. Wang and Andrew W. Appel. POPL 2001.
  Shows how to modify MLton to generate a strongly typed garbage collector as part of a program.
- Programming With Recursion Schemes. Daniel C. Wang and Tom Murphy VII.
  Describes a programming technique for data abstraction, along with benchmarks of MLton and other SML compilers.
- Recursion Schemes as Abstract Interfaces. Daniel C. Wang and Tom Murphy. JFP.
- Whole-Program Compilation in MLton. Stephen Weeks. ML 2006.
- Simple Imperative Polymorphism. Andrew Wright. LASC, 8(4):343-355, 1995.

The origin of the ValueRestriction.

X

Y

- Encoding Types in ML-like Languages. Zhe Yang. ICFP 1998.

Z

Abbreviations

- ACSD = International Conference on Application of Concurrency to System Design
- BABEL = Workshop on multi-language infrastructure and interoperability
- CC = International Conference on Compiler Construction
- ESOP = European Symposium on Programming
- FLOPS = Symposium on Functional and Logic Programming
- FPCA = Conference on Functional Programming and Computer Architecture
- HOSC = Higher-Order and Symbolic Computation
- ICCL = IEEE International Conference on Computer Languages
- ICFP = International Conference on Functional Programming
- IFL = International Workshop on Implementation and Application of Functional Languages
- IVME = Workshop on Interpreters, Virtual Machines and Emulators
- JFLA = Journees Francophones des Langages Applicatifs
- JFP = Journal of Functional Programming
- LASC = Lisp and Symbolic Computation
- ML = Workshop on ML
- PLDI = Conference on Programming Language Design and Implementation
- POPL = Symposium on Principles of Programming Languages
- PPDP = International Conference on Principles and Practice of Declarative Programming
- PPoPP = Principles and Practice of Parallel Programming
- TCS = IFIP International Conference on Theoretical Computer Science
- TIC = Types in Compilation
- TLDI = Workshop on Types in Language Design and Implementation
- TPHOLs = International Conference on Theorem Proving in Higher Order Logics

---

Last edited on 2006-10-19 19:33:38 by StephenWeeks.

Regions

In region-based memory management, the heap is divided into a collection of regions into which objects are allocated. At compile time, either in the source program or through automatic inference, allocation points are annotated with the region in which the allocation will occur. Typically, although not always, the regions are allocated and deallocated according to a stack discipline.

MLton does not use region-based memory management; it uses traditional GarbageCollection. We have considered integrating regions with MLton, but in our opinion it is far from clear that regions would provide MLton with improved performance, while they would certainly add a lot of complexity to the compiler and complicate reasoning about and achieving SpaceSafety. Region-based memory management and garbage collection have different strengths and weaknesses; it's pretty easy to come up with programs that do significantly better under regions than under GC, and vice versa. We believe that it is the case that common SML idioms tend to work better under GC than under regions.

One common argument for regions is that the region operations can all be done in (approximately) constant time; therefore, you eliminate GC pause times, leading to a real-time GC. However, because of space safety concerns (see below), we believe that region-based memory management for SML must also include a traditional garbage collector. Hence, to achieve real-time memory management for MLton/SML, we believe that it would be both easier and more efficient to implement a traditional real-time garbage collector than it would be to implement a region system.

Regions, the ML Kit, and space safety

The ML Kit pioneered the use of regions for compiling Standard ML. The ML Kit maintains a stack of regions at run time. At compile time, it uses region inference to decide when data can be allocated in a stack-like manner, assigning it to an appropriate region. The ML Kit has put a lot of effort into improving the supporting analyses and representations of regions, which are all necessary to improve the performance.

Unfortunately, under a pure stack-based region system, space leaks are inevitable in theory, and costly in practice. Data for which region inference can not determine the lifetime is moved into the *global region* whose lifetime is the entire program. There are two ways in which region inference will place an object to the global region.

- When the inference is too conservative, that is, when the data is used in a stack-like manner but the region inference can't figure it out.
- When data is not used in a stack-like manner. In this case, correctness requires region inference to place the object

This global region is a source of space leaks. No matter what region system you use, there are some programs such that the global region must exist, and its size will grow to an unbounded multiple of the live data size. For these programs one must have a GC to achieve space safety.

To solve this problem, the ML Kit has undergone work to combine garbage collection with region-based memory management. HallenbergEtAl02 and Elsman03 describe the addition of a garbage collector to the ML Kit's region-based system. These papers provide convincing evidence for space leaks in the global region. They show a number of benchmarks where the memory usage of the program running with just regions is a large multiple (2, 10, 50, even 150) of the program running with regions plus GC.

These papers also give some numbers to show the ML Kit with just regions does better than either a system with just GC or a combined system. Unfortunately, a pure region system isn't practical because of the lack of space safety. And the other performance numbers are not so convincing, because they compare to an old version of SML/NJ and not at all with MLton. It would be interesting to see a comparison with a more serious collector.

Regions, Garbage Collection, and Cyclone

One possibility is to take Cyclone's approach, and provide both region-based memory management and garbage collection, but at the programmer's option (GrossmanEtAl02, HicksEtAl03).

One might ask whether we might do the same thing -- i.e., provide a `MLton.Regions` structure with explicit region based memory management operations, so that the programmer could use them when appropriate. MatthewFluet has thought about this question

http://www.cs.cornell.edu/People/fluet/rgn-monad/index.html

Unfortunately, his conclusion is that the SML type system is too weak to support this option, although there might be a "poor-man's" version with dynamic checks.

Last edited on 2005-09-06 23:20:00 by MatthewFluet.

ReleaseChecklist

Before Packaging

- Wiki
  - check OrphanedPages and WantedPages.
  - spell check.
- Check man pages and set the dates if necessary.
- Update `doc/changelog` with a summary.

After Packaging

- tag the release in the SVN repository.
- `mlton.org`
  - `basis` gets a snapshot of http://standardml.org/Basis.
  - `changelog` gets a copy of `doc/changelog`.
  - Put up MLton Guide.
  - Home gets note of new release.
  - Download gets release notes and executables.
  - Experimental is cleared.
- Send mail to
  - MLton@mlton.org
  - MLton-user@mlton.org
  - sml-list@cs.cmu.edu (aka news:comp.lang.ml)
  - lwn@lwn.net (linux weekly news)
- Post to
  - news:comp.lang.functional
- Update OtherSites that have MLton pages.
- dupload Debian package.
- Generate new Performance numbers.

Last edited on 2005-12-03 04:51:01 by StephenWeeks.

RemoveUnused

RemoveUnused is an optimization pass for both the SSA and SSA2 IntermediateLanguages, invoked from SSASimplify and SSA2Simplify.

Description

This pass aggressively removes unused:

- datatypes
- datatype constructors
- datatype constructor arguments
- functions
- function arguments
- function returns
- blocks
- block arguments
- statements (variable bindings)
- handlers from non-tail calls (mayRaise analysis)
- continuations from non-tail calls (mayReturn analysis)

Implementation

remove-unused.sig remove-unused.fun remove-unused2.sig remove-unused2.fun

Details and Notes

Last edited on 2005-12-02 01:03:46 by StephenWeeks.

Restore

Restore is a rewrite pass for the SSA and SSA2 IntermediateLanguages, invoked from KnownCase and LocalRef.

Description

This pass restores the SSA condition for a violating SSA or SSA2 program; the program must satisfy:

Every path from the root to a use of a variable (excluding globals) passes through a def of that variable.

Implementation

restore.sig restore.fun
restore2.sig restore2.fun

Details and Notes

Based primarily on Section 19.1 of Modern Compiler Implementation in ML.

The main deviation is the calculation of liveness of the violating variables, which is used to predicate the insertion of phi arguments. This is due to the algorithm's bias towards imperative languages, for which it makes the assumption that all variables are defined in the start block and all variables are "used" at exit.

This is "optimized" for restoration of functions with small numbers of violating variables -- use bool vectors to represent sets of violating variables.

Also, we use a Promise.t to suspend part of the dominance frontier computation.

Last edited on 2005-12-02 03:19:52 by StephenWeeks.

Roadmap

This roadmap details what we hope will be in the next MLton release, currently slated for the second quarter of 2006.

- x86-64 native code generator
- support for creating shared libraries
- Unicode support

Last edited on 2005-12-02 21:03:07 by StephenWeeks.

RunTimeOptions

Executables produced by MLton take command line arguments that control the runtime system. These arguments are optional, and occur before the executable's usual arguments. To use these options, the first argument to the executable must be `@MLton`. The optional arguments then follow, must be terminated by `--`, and are followed by any arguments to the program. The optional arguments are *not* made available to the SML program via `CommandLine.arguments`. For example, a valid call to `hello-world` is:

```
hello-world @MLton gc-summary fixed-heap 10k -- a b c
```

In the above example, `CommandLine.arguments () = ["a", "b", "c"].`

It is allowed to have a sequence of `@MLton` arguments, as in:

```
hello-world @MLton gc-summary -- @MLton fixed-heap 10k -- a b c
```

Run-time options can also control MLton, as in

```
mlton @MLton fixed-heap 0.5g -- foo.sml
```

Options

- `fixed-heap` *x*`{k|K|m|M|g|G}`
  Use a fixed size heap of size *x*, where *x* is a real number and the trailing letter indicates its units.

`k` or `K`	1024
`m` or `M`	1,048,576
`g` or `G`	1,073,741,824

  A value of `0` means to use almost all the RAM present on the machine.

  The heap size used by `fixed-heap` includes all memory allocated by SML code, including memory for the stack (or stacks, if there are multiple threads). It does not, however, include any memory used for code itself or memory used by C globals, the C stack, or malloc.

- `gc-messages`
  Print a message at the start and end of every garbage collection.
- `gc-summary`
  Print a summary of garbage collection statistics upon program termination.
- `load-world` *world*
  Restart the computation with the file specified by *world*, which must have been created by a call to `MLton.World.save` by the same executable. See MLtonWorld.
- `max-heap` *x*`{k|K|m|M|g|G}`
  Run the computation with an automatically resized heap that is never larger than *x*, where *x* is a real number and the trailing letter indicates the units as with `fixed-heap`. The heap size for `max-heap` is accounted for as with `fixed-heap`.
- `no-load-world`
  Disable `load-world`. This can be used as an argument to the compiler via `-runtime no-load-world` to create executables that will not load a world. This may be useful to ensure that set-uid executables do not load some strange world.

- `ram-slop` *x*
  Multiply *x* by the amount of RAM on the machine to obtain what the runtime views as the amount of RAM it can use. Typically *x* is less than 1, and is used to account for space used by other programs running on the same machine.
- `stop`

Causes the runtime to stop processing `@MLton` arguments once the next `--` is reached. This can be used as an argument to the compiler via `-runtime stop` to create executables that don't process any `@MLton` arguments.

Last edited on 2005-12-02 06:13:52 by StephenWeeks.

RunningOnCygwin

MLton runs on the Cygwin emulation layer, which provides a Posix-like environment while running on Windows. To run MLton with Cygwin, you must first install Cygwin on your Windows machine. To do this, visit the Cygwin site from your Windows machine and run their `setup.exe` script. Then, you can unpack the MLton binary `tgz` in your Cygwin environment.

To run MLton cross-compiled executables on Windows, you must install the Cygwin `dll` on the Windows machine.

Known issues

- Time profiling is disabled.
- Cygwin's `mmap` emulation is less than perfect. Sometimes it interacts badly with `Posix.Process.fork`. For idiomatic uses of `fork` plus `exec`, you can instead use the `MLton.Process.spawn` family of functions, which work on all our platforms.
- Cygwin's `mmap` emulation does not make available as much contiguous virtual address space as using the Windows `VirtualAlloc` function. Earlier versions of MLton used `VirtualAlloc` instead of `mmap`, but that no longer works.

Also see

- RunningOnMinGW

Last edited on 2006-07-20 19:36:31 by StephenWeeks.

RunningOnDarwin

MLton runs fine on Darwin, which underlies Mac OSX.

- MLton requires the GnuMP library, which fink has here.

Also see

- RunningOnPowerPC

Last edited on 2005-12-02 01:14:31 by StephenWeeks.

RunningOnFreeBSD

MLton is available as a FreeBSD port.

Known issues

- Executables often run more slowly than on a comparable Linux machine. We conjecture that part of this is due to costs due to heap resizing and kernel zeroing of pages. Any help in solving the problem would be appreciated.

Last edited on 2006-07-20 19:36:37 by StephenWeeks.

RunningOnLinux

The are no known issues using MLton on Linux.

Last edited on 2004-11-02 00:56:09 by StephenWeeks.

RunningOnMinGW

MLton runs on MinGW, a library for porting Unix applications to Windows. Some library functionality is missing or changed.

- Many functions are unimplemented and will `raise SysErr`.
    - `MLton.Itimer.set`
    - `MLton.ProcEnv.setgroups`
    - `MLton.Process.kill`
    - `MLton.Process.reap`
    - `MLton.World.load`
    - `OS.FileSys.readLink`
    - `OS.IO.poll`
    - `OS.Process.terminate`
    - `Posix.FileSys.chown`
    - `Posix.FileSys.fchown`
    - `Posix.FileSys.fpathconf`
    - `Posix.FileSys.link`
    - `Posix.FileSys.mkfifo`
    - `Posix.FileSys.pathconf`
    - `Posix.FileSys.readlink`
    - `Posix.FileSys.symlink`
    - `Posix.IO.dupfd`
    - `Posix.IO.getfd`
    - `Posix.IO.getfl`
    - `Posix.IO.getlk`
    - `Posix.IO.setfd`
    - `Posix.IO.setfl`
    - `Posix.IO.setlkw`
    - `Posix.IO.setlk`
    - `Posix.ProcEnv.ctermid`
    - `Posix.ProcEnv.getegid`
    - `Posix.ProcEnv.geteuid`
    - `Posix.ProcEnv.getgid`
    - `Posix.ProcEnv.getgroups`
    - `Posix.ProcEnv.getlogin`
    - `Posix.ProcEnv.getpgrp`
    - `Posix.ProcEnv.getpid`
    - `Posix.ProcEnv.getppid`
    - `Posix.ProcEnv.getuid`
    - `Posix.ProcEnv.setgid`
    - `Posix.ProcEnv.setpgid`
    - `Posix.ProcEnv.setsid`
    - `Posix.ProcEnv.setuid`
    - `Posix.ProcEnv.sysconf`
    - `Posix.ProcEnv.times`
    - `Posix.ProcEnv.ttyname`
    - `Posix.Process.exece`
    - `Posix.Process.execp`
    - `Posix.Process.exit`

- `Posix.Process.fork`
- `Posix.Process.kill`
- `Posix.Process.pause`
- `Posix.Process.waitpid_nh`
- `Posix.Process.waitpid`
- `Posix.SysDB.getgrgid`
- `Posix.SysDB.getgrnam`
- `Posix.SysDB.getpwuid`
- `Posix.TTY.TC.drain`
- `Posix.TTY.TC.flow`
- `Posix.TTY.TC.flush`
- `Posix.TTY.TC.getattr`
- `Posix.TTY.TC.getpgrp`
- `Posix.TTY.TC.sendbreak`
- `Posix.TTY.TC.setattr`
- `Posix.TTY.TC.setpgrp`
- `Unix.kill`
- `Unix.reap`
- `UnixSock.fromAddr`
- `UnixSock.toAddr`

Last edited on 2006-09-07 00:27:10 by StephenWeeks.

RunningOnNetBSD

MLton runs fine on NetBSD.

Installing the correct packages for NetBSD

The NetBSD system installs 3rd party packages by a mechanism known as pkgsrc. This is a tree of Makefiles which when invoked downloads the source code, builds a package and installs it on the system. In order to run MLton on NetBSD, you will have to install several packages for it to work:

- `shells/bash`
- `devel/gmp`
- `devel/gmake`

In order to get graphical call-graphs of profiling information, you will need the additional package

- `graphics/graphviz`

To build the documentation for MLton, you need `htmldoc`.

Tips for compiling and using MLton on NetBSD

MLton can be a memory-hog on computers with little memory. While 640Mb of RAM ought to be enough to self-compile MLton one might want to do some tuning to the NetBSD VM subsystem in order to succeed. The notes presented here is what JesperLouisAndersen uses for compiling MLton on his laptop.

The NetBSD VM subsystem

NetBSD uses a VM subsystem named UVM. Tuning the VM system can be done via the `sysctl(8)`-interface with the "VM" MIB set.

Tuning the NetBSD VM subsystem for MLton

MLton uses a lot of anonymous pages when it is running. Thus, we will need to tune up the default of 80 for anonymous pages. Setting

```
sysctl -w vm.anonmax=95
sysctl -w vm.anonmin=50
sysctl -w vm.filemin=2
sysctl -w vm.execmin=2
sysctl -w vm.filemax=4
sysctl -w vm.execmax=4
```

makes it less likely for the VM system to swap out anonymous pages. For a full explanation of the above flags, see the documentation.

The result is that my laptop goes from a MLton compile where it swaps a lot to a MLton compile with no swapping.

Last edited on 2006-07-20 19:36:43 by StephenWeeks.

RunningOnOpenBSD

MLton runs fine on OpenBSD.

Known issues

- Our socket regression test fails. We suspect this is not a bug and is simply due to our test relying on a certain behavior when connecting to a socket that has not yet accepted, which is handled differently on OpenBSD than other platforms. Any help in understanding and resolving this issue is appreciated.

Last edited on 2006-07-20 19:36:47 by StephenWeeks.

RunningOnPowerPC

MLton runs fine on PowerPC.

Known issues

- When compiling for PowerPC, MLton doesn't support native code generation (`-codegen native`). Hence, performance is not as good as it might be and compile times are longer. Also, the quality of code generated by `gcc` is important. By default, MLton calls `gcc -O1`. You can change this by calling MLton with `-cc-opt -O2`.

Last edited on 2005-12-02 01:36:54 by StephenWeeks.

RunningOnSolaris

MLton runs fine on Solaris.

Known issues

- You must install the `binutils`, `gcc`, and `make` packages. You can find out how to get these at sunfreeware.com
- Making the documentation requires that you install `latex` and `dvips`, which are available in the `tetex` package. It also requires `hevea`, for which we haven't yet tracked down a package.
- Bootstrapping is so slow as to be impractical (many hours on a 500MHz UltraSparc). For this reason, we strongly recommend building with a Linux to Solaris cross compiler.

Also see

- RunningOnSparc

Last edited on 2005-12-02 04:24:41 by StephenWeeks.

RunningOnSparc

MLton runs fine on Sparc.

Known issues

- When compiling for Sparc, MLton doesn't support native code generation (`-codegen native`). Hence, performance is not as good as it might be and compile times are longer. Also, the quality of code generated by `gcc` is important. By default, MLton calls `gcc -O1`. You can change this by calling MLton with `-cc-opt -O2`. We have seen this speed up some programs by as much as 30%, especially those involving floating point; however, it can also more than double compile times.
- When compiling for Sparc, MLton uses `-align 8` by default. While this speeds up reals, it also may increase object sizes. If your program does not make significant use of reals, you might see a speedup with `-align 4`.

Last edited on 2005-12-02 01:38:25 by StephenWeeks.

SMLNET

SML.NET is a Standard ML Compiler that targets the .NET Common Language Runtime.

SML.NET is based on the MLj compiler.

BentonEtAl04 describes SML.NET.

Last edited on 2004-12-30 20:11:30 by StephenWeeks.

SMLNJ

SML/NJ is a Standard ML Compiler. It is a native code compiler that runs on a variety of platforms and has a number of libraries and tools.

We maintain a list of SML/NJ's deviations from the Definition of SML.

MLton has support for some features of SML/NJ in order to ease porting between MLton and SML/NJ.

- CompilationManager (CM)
- LineDirectives
- SMLofNJStructure
- UnsafeStructure

Last edited on 2004-12-30 20:12:30 by StephenWeeks.

SMLNJDeviations

Here are some deviations of SML/NJ from the Definition of SML. Some of these are documented in the SML '97 Conversion Guide. Since MLton does not deviate from the Definition, you should look here if you are having trouble porting a program from MLton to SML/NJ or vice versa. If you discover other deviations of SML/NJ that aren't listed here, please send mail to MLton@mlton.org.

- SML/NJ allows spaces in long identifiers, as in `S . x`. Section 2.5 of the Definition implies that `S . x` should be treated as three separate lexical items.
- SML/NJ rejects

  ```
  (op *)
  ```

  as an unmatched close comment.
- SML/NJ allows = to be rebound by the declaration:

  ```
  val op = = 13
  ```

  This is explicitly forbidden on page 5 of the Definition.
- SML/NJ allows rebinding `true`, `false`, `nil`, `::`, and `ref` by the declarations:

  ```
  fun true () = ()
  fun false () = ()
  fun nil () = ()
  fun op :: () = ()
  fun ref () = ()
  ```

  This is explicitly forbidden on page 9 of the Definition.
- SML/NJ extends the syntax of the language to allow vector expressions and patterns like the following:

  ```
  val v = #[1,2,3]
  val #[x,y,z] = v
  ```
- SML/NJ extends the syntax of the language to allow *or patterns* like the following:

  ```
  datatype foo = Foo of int | Bar of int
  val (Foo x | Bar x) = Foo 13
  ```
- SML/NJ allows higher-order functors, that is, functors can be components of structures and can be passed as functor arguments and returned as functor results. As a consequence, SML/NJ allows abbreviated functor definitions, as in the following:

  ```
  signature S =
    sig
      type t
      val x: t
    end
  functor F (structure A: S): S =
    struct
      type t = A.t * A.t
      val x = (A.x, A.x)
    end
  functor G = F
  ```
- SML/NJ extends the syntax of the language to allow functor and signature definitions to occur within the scope of `local` and `structure` declarations.

* SML/NJ allows duplicate type specifications in signatures when the duplicates are introduced by `include`, as in the following:

```
signature SIG1 =
   sig
      type t
      type u
   end
signature SIG2 =
   sig
      type t
      type v
   end
signature SIG =
   sig
      include SIG1
      include SIG2
   end
```

  This is disallowed by rule 77 of the Definition.
* SML/NJ allows sharing constraints between type abbreviations in signatures, as in the following:

```
signature SIG =
   sig
      type t = int * int
      type u = int * int
      sharing type t = u
   end
```

  These are disallowed by rule 78 of the Definition.
* SML/NJ disallows multiple `where type` specifications of the same type name, as in the following

```
signature S =
  sig
     type t
     type u = t
  end
  where type u = int
```

  This is allowed by rule 84 of the Definition.
* SML/NJ allows `and` in `sharing` specs in signatures, as in

```
signature S =
   sig
      type t
      type u
      type v
      sharing type t = u
      and type u = v
   end
```

* SML/NJ does not expand the `withtype` derived form as described by the Definition. According to page 55 of the Definition, the type bindings of a `withtype` declaration are substituted simultaneously in the connected datatype. Consider the following program.

```
type u = real
datatype a =
    A of t
```

```
  | B of u
withtype u = int
and t = u
```

According to the Definition, it should be expanded to the following.

```
type u = real
datatype a =
    A of u
  | B of int
```

However, SML/NJ expands `withtype` bindings sequentially, meaning that earlier bindings are expanded within later ones. Hence, the above program is expanded to the following.

```
type u = real
datatype a =
    A of int
  | B of int
```

- SML/NJ allows `withtype` specifications in signatures.
- SML/NJ allows a `where` structure specification that is similar to a `where type` specification. For example:

```
structure S = struct type t = int end
signature SIG =
  sig
     structure T : sig type t end
  end where T = S
```

This is equivalent to:

```
structure S = struct type t = int end
signature SIG =
  sig
     structure T : sig type t end
  end where type T.t = S.t
```

SML/NJ also allows a definitional structure specification that is similar to a definitional type specification. For example:

```
structure S = struct type t = int end
signature SIG =
  sig
     structure T : sig type t end = S
  end
```

This is equivalent to the previous examples and to:

```
structure S = struct type t = int end
signature SIG =
  sig
     structure T : sig type t end where type t = S.t
  end
```

- SML/NJ disallows binding non-datatypes with datatype replication. For example, it rejects the following program that should be allowed according to the Definition.

```
type ('a, 'b) t = 'a * 'b
```

```
datatype u = datatype t
```

This idiom can be useful when one wants to rename a type without rewriting all the type arguments. For example, the above would have to be written in SML/NJ as follows.

```
type ('a, 'b) t = 'a * 'b
type ('a, 'b) u = ('a, 'b) t
```

- SML/NJ disallows sharing a structure with one of its substructures. For example, SML/NJ disallows the following.

```
signature SIG =
   sig
      structure S:
         sig
            type t
            structure T: sig type t end
         end
      sharing S = S.T
   end
```

This signature is allowed by the Definition.

- SML/NJ disallows polymorphic generalization of refutable patterns. For example, SML/NJ disallows the following.

```
val [x] = [[]]
val _ = (1 :: x, "one" :: x)
```

Deviations from the Basis Library Specification

Here are some deviations of SML/NJ from the Basis Library Specification.

- SML/NJ exposes the equality of the `vector` type in structures such as `Word8Vector` that abstractly match `MONO_VECTOR`, which says `type vector`, not `eqtype vector`. So, for example, SML/NJ accepts the following program:

```
fun f (v: Word8Vector.vector) = v = v
```

Last edited on 2006-08-15 07:55:07 by VesaKarvonen.

SMLNJLibrary

The SML/NJ Library is a collection of libraries that are distributed with SML/NJ. Due to differences between SML/NJ and MLton, these libraries will not work out-of-the box with MLton.

As of 20050818, MLton includes a port of the SML/NJ Library, currently synchronized with SML/NJ version 110.57.

Usage

* You can import a sub-library of the SML/NJ Library into an MLB file with:

MLB file	Description
`$(SML_LIB)/smlnj-lib/Util/smlnj-lib.mlb`	Various utility modules, included collections, simple formating, ...
`$(SML_LIB)/smlnj-lib/Controls/controls-lib.mlb`	A library for managing control flags in an application.
`$(SML_LIB)/smlnj-lib/HashCons/hash-cons-lib.mlb`	Support for implementing hash-consed data structures.
`$(SML_LIB)/smlnj-lib/INet/inet-lib.mlb`	Networking utilities; supported on both Unix and Windows systems.
`$(SML_LIB)/smlnj-lib/Unix/unix-lib.mlb`	Utilities for Unix-based operating systems.
`$(SML_LIB)/smlnj-lib/PP/pp-lib.mlb`	Pretty-printing library.
`$(SML_LIB)/smlnj-lib/HTML/html-lib.mlb`	HTML parsing and pretty-printing library.
`$(SML_LIB)/smlnj-lib/RegExp/regexp-lib.mlb`	Regular expression library.
`$(SML_LIB)/smlnj-lib/Reactive/reactive-lib.mlb`	Reactive scripting library.

* If you are porting a project from SML/NJ's CompilationManager to MLton's ML Basis system using `cm2mlb`, note that the following maps are included by default:

```
$smlnj-lib.cm        $(SML_LIB)/smlnj-lib/Util
$controls-lib.cm     $(SML_LIB)/smlnj-lib/Controls
$hash-cons-lib.cm    $(SML_LIB)/smlnj-lib/HashCons
$inet-lib.cm         $(SML_LIB)/smlnj-lib/INet
$unix-lib.cm         $(SML_LIB)/smlnj-lib/Unix
$pp-lib.cm           $(SML_LIB)/smlnj-lib/PP
```

```
$html-lib.cm        $(SML_LIB)/smlnj-lib/HTML
$regexp-lib.cm      $(SML_LIB)/smlnj-lib/RegExp
$reactive-lib.cm    $(SML_LIB)/smlnj-lib/Reactive
```

This will automatically convert a `$/smlnj-lib.cm` import in an input `.cm` file into a `$(SML_LIB)/smlnj-lib/Util/smlnj-lib.mlb` import in the output `.mlb` file.

Details

The following changes were made to the SML/NJ Library, in addition to deriving the `.mlb` files from the `.cm` files:

- `Util/redblack-set-fn.sml` (modified): Rewrote use of `where` structure specification.
- `Util/redblack-map-fn.sml` (modified): Rewrote use of `where` structure specification.
- `Util/graph-scc.sml` (modified): Rewrote use of `where` structure specification.
- `Util/bit-array.sml` (modified): The computation of the `maxLen` is given by:

  ```
  val maxLen = 8*Word8Array.maxLen
  ```

  This is fine in SML/NJ where `Word8Array.maxLen` is 16777215, but in MLton, `Word8Array.maxLen` is equal to `valOf(Int.maxInt)`, so the computation overflows. To accommodate both SML/NJ and MLton, the computation is replaced by

  ```
  val maxLen = (8*Word8Array.maxLen) handle Overflow => Word8Array.maxLen
  ```
- `Util/engine.mlton.sml` (added, not exported): Implements `structure Engine`, providing time-limited, resumable computations using MLtonThread, MLtonSignal, and MLtonItimer.
- `Util/time-limit.mlton.sml` (added): Implements `structure TimeLimit` using `structure Engine`. The SML/NJ implementation of `structure TimeLimit` uses SML/NJ's first-class continuations, signals, and interval timer.
- `Util/time-limit.mlb` (added): Exports `structure TimeLimit`, which is *not* exported by `smlnj-lib.mlb`. Since MLton is very conservative in the presence of threads and signals, program performance may be adversely affected by unnecessarily including `structure TimeLimit`.
- `HTML/html-elements-fn.sml` (modified): Rewrote use of *or-patterns*.
- `HTML/html-attrs-fn.sml` (modified): Rewrote use of *or-patterns*.

Patch

- smlnj-lib.patch

Last edited on 2006-03-04 17:06:17 by MatthewFluet.

SMLofNJStructure

```
signature SML_OF_NJ =
   sig
      structure Cont:
         sig
            type 'a cont
            val callcc: ('a cont -> 'a) -> 'a
            val throw: 'a cont -> 'a -> 'b
         end
      structure SysInfo:
         sig
            exception UNKNOWN
            datatype os_kind = BEOS | MACOS | OS2 | UNIX | WIN32

            val getHostArch: unit -> string
            val getOSKind: unit -> os_kind
            val getOSName: unit -> string
         end

      val exnHistory: exn -> string list
      val exportFn: string * (string * string list -> OS.Process.status) -> unit
      val exportML: string -> bool
      val getAllArgs: unit -> string list
      val getArgs: unit -> string list
      val getCmdName: unit -> string
   end
```

`SMLofNJ` implements a subset of the structure of the same name provided in Standard ML of New Jersey. It is included to make it easier to port programs between the two systems. The semantics of these functions may be different than in SML/NJ.

- `structure Cont`
  implements continuations.
- `SysInfo.getHostArch ()`
  returns the string for the architecture.
- `SysInfo.getOSKind`
  returns the OS kind.
- `SysInfo.getOSName ()`
  returns the string for the host.
- `exnHistory`
  the same as `MLton.Exn.history`.
- `getCmdName ()`
  the same as `CommandLine.name ()`.
- `getArgs ()`
  the same as `CommandLine.arguments ()`.
- `getAllArgs ()`
  the same as `getCmdName()::getArgs()`.
- `exportFn f`
  saves the state of the computation to a file that will apply `f` to the command-line arguments upon restart.
- `exportML f`

saves the state of the computation to file `f` and continue. Returns `true` in the restarted computation and `false` in the continuing computation.

Last edited on 2005-12-02 02:31:55 by StephenWeeks.

SSA

SSA is an IntermediateLanguage, translated from SXML by ClosureConvert, optimized by SSASimplify, and translated by ToSSA2 to SSA2.

Description

SSA is a FirstOrder, SimplyTyped IntermediateLanguage. It is the main IntermediateLanguage used for optimizations.

An SSA program consists of a collection of datatype declarations, a sequence of global statements, and a collection of functions, along with a distinguished "main" function. Each function consists of a collection of basic blocks, where each basic block is a sequence of statements ending with some control transfer.

Implementation

ssa.sig ssa.fun
ssa-tree.sig ssa-tree.fun

Type Checking

Type checking of a SSA program verifies the following:

- no duplicate definitions (tycons, cons, vars, labels, funcs)
- no out of scope references (tycons, cons, vars, labels, funcs)
- variable definitions dominate variable uses
- case transfers are exhaustive and irredundant
- `Enter`/`Leave` profile statements match
- "traditional" well-typedness

type-check.sig type-check.fun

Details and Notes

SSA is an abbreviation for Static Single Assignment.

Last edited on 2005-12-02 04:25:39 by StephenWeeks.

SSA2

SSA2 is an IntermediateLanguage, translated from SSA by ToSSA2, optimized by SSA2Simplify, and translated by ToRSSA to RSSA.

Description

SSA2 is a FirstOrder, SimplyTyped IntermediateLanguage, a slight variant of the SSA IntermediateLanguage,

Like SSA, a SSA program consists of a collection of datatype declarations, a sequence of global statements, and a collection of functions, along with a distinguished "main" function. Each function consists of a collection of basic blocks, where each basic block is a sequence of statements ending with some control transfer.

Unlike SSA, SSA2 includes mutable fields in objects and makes the vector type constructor n-ary instead of unary. This allows optimizations like RefFlatten and DeepFlatten to be expressed.

Implementation

ssa2.sig ssa2.fun
ssa-tree2.sig ssa-tree2.fun

Type Checking

Type checking of a SSA2 program verfies the following:

- no duplicate definitions (tycons, cons, vars, labels, funcs)
- no out of scope references (tycons, cons, vars, labels, funcs)
- variable definitions dominate variable uses
- case transfers are exhaustive and irredundant
- `Enter`/`Leave` profile statements match
- "traditional" well-typedness

type-check2.sig type-check2.fun

Details and Notes

SSA is an abbreviation for Static Single Assignment.

Last edited on 2005-12-02 03:19:44 by StephenWeeks.

SSA2Simplify

The optimization passes for the SSA2 IntermediateLanguage are collected and controlled by the Simplify2 functor (simplify2.sig , simplify2.fun).

The following optimization passes are implemented:

- DeepFlatten
- RefFlatten
- RemoveUnused
- Zone

There are additional analysis and rewrite passes that augment many of the other optimization passes:

- Restore
- Shrink

The optimization passes can be controlled from the command-line by the options

- diag-pass <pass> -- keep diagnostic info for pass
- drop-pass <pass> -- omit optimization pass
- keep-pass <pass> -- keep the results of pass
- loop-passes <n> -- loop optimization passes
- ssa2-passes <passes> -- ssa optimization passes

Last edited on 2005-08-19 15:27:05 by MatthewFluet.

SSASimplify

The optimization passes for the SSA IntermediateLanguage are collected and controlled by the `Simplify` functor (simplify.sig , simplify.fun).

The following optimization passes are implemented:

- CommonArg
- CommonBlock
- CommonSubexp
- ConstantPropagation
- Contify
- Flatten
- Inline
- IntroduceLoops
- KnownCase
- LocalFlatten
- LocalRef
- LoopInvariant
- Redundant
- RedundantTests
- RemoveUnused
- SimplifyTypes
- Useless

The following implementation pass is implemented:

- PolyEqual

There are additional analysis and rewrite passes that augment many of the other optimization passes:

- Multi
- Restore
- Shrink

The optimization passes can be controlled from the command-line by the options:

- `diag-pass <pass>` -- keep diagnostic info for pass
- `drop-pass <pass>` -- omit optimization pass
- `keep-pass <pass>` -- keep the results of pass
- `loop-passes <n>` -- loop optimization passes
- `ssa-passes <passes>` -- ssa optimization passes

Last edited on 2005-08-19 15:26:49 by MatthewFluet.

SXML

SXML is an IntermediateLanguage, translated from XML by Monomorphise, optimized by SXMLSimplify, and translated by ClosureConvert to SSA.

Description

SXML is a simply-typed version of XML.

Implementation

sxml.sig sxml.fun
sxml-tree.sig

Type Checking

SXML shares the type checker for XML.

Details and Notes

There are only two differences between XML and SXML. First, SXML `val`, `fun`, and `datatype` declarations always have an empty list of type variables. Second, SXML variable references always have an empty list of type arguments. Constructors uses can only have a nonempty list of type arguments if the constructor is a primitive.

Although we could rely on the type system to enforce these constraints by parameterizing the XML signature, StephenWeeks did so in a previous version of the compiler, and the software engineering gains were not worth the effort.

Last edited on 2005-12-02 02:42:31 by StephenWeeks.

SXMLShrink

SXMLShrink is an optimization pass for the SXML IntermediateLanguage, invoked from SXMLSimplify.

Description

This pass performs optimizations based on a reduction system.

Implementation

shrink.sig shrink.fun

Details and Notes

SXML shares the XMLShrink simplifier.

Last edited on 2005-12-02 02:42:47 by StephenWeeks.

SXMLSimplify

The optimization passes for the SXML IntermediateLanguage are collected and controlled by the SxmlSimplify functor (sxml-simplify.sig , sxml-simplify.fun).

The following optimization passes are implemented:

- Polyvariance
- SXMLShrink

The following implementation passes are implemented:

- ImplementExceptions
- ImplementSuffix

The following optimization passes are not implemented, but might prove useful:

- Uncurry
- LambdaLift

The optimization passes can be controlled from the command-line by the options

- diag-pass <pass> -- keep diagnostic info for pass
- drop-pass <pass> -- omit optimization pass
- keep-pass <pass> -- keep the results of pass
- sxml-passes <passes> -- sxml optimization passes

Last edited on 2005-08-19 15:25:57 by MatthewFluet.

ScopeInference

Scope inference is an analysis/rewrite pass for the AST IntermediateLanguage, invoked from Elaborate.

Description

This pass adds free type variables to the `val` or `fun` declaration where they are implicitly scoped.

Implementation

scope.sig scope.fun

Details and Notes

Scope inference determines for each type variable, the declaration where it is bound. Scope inference is a direct implementation of the specification given in section 4.6 of the Definition. Recall that a free occurrence of a type variable `'a` in a declaration `d` is *unguarded* in `d` if `'a` is not part of a smaller declaration. A type variable `'a` is implicitly scoped at `d` if `'a` is unguarded in `d` and `'a` does not occur unguarded in any declaration containing `d`.

The first pass of scope inference walks down the tree and renames all explicitly bound type variables in order to avoid name collisions. It then walks up the tree and adds to each declaration the set of unguarded type variables occurring in that declaration. At this point, if declaration `d` contains an unguarded type variable `'a` and the immediately containing declaration does not contain `'a`, then `'a` is implicitly scoped at `d`. The final pass walks down the tree leaving a `'a` at the a declaration where it is scoped and removing it from all enclosed declarations.

Last edited on 2005-12-02 01:43:12 by StephenWeeks.

SelfCompiling

If you want to compile MLton, you must first get the Sources. You can compile with either MLton or SML/NJ, but we strongly recommend using MLton, since it generates a much faster and more robust executable.

Compiling with MLton

To compile with MLton, you need the binary versions of `mlton`, `mllex`, and `mlyacc` that come with the MLton binary package. To be safe, you should use the same version of MLton that you are building. However, older versions may work, as long as they don't go back too far. To build MLton, run `make` from within the root directory of the sources. This will build MLton first with the already installed binary version of MLton and will then rebuild MLton with itself.

First, the `Makefile` calls `mllex` and `mlyacc` to build the lexer and parser, and then calls `mlton` to compile itself. When making MLton using another version the `Makefile` automatically uses `mlton-stubs.cm`, which will put in enough stubs to emulate the `MLton` structure. Once MLton is built, the `Makefile` will rebuild MLton with itself, this time using `mlton.cm` and the real `MLton` structure from the Basis Library. This second round of compilation is essential in order to achieve a fast and robust MLton.

Compiling MLton requires at least 512M of actual RAM, and 1G is preferable. If your machine has less than 512M, self-compilation will likely fail, or at least take a very long time due to paging. Even if you have enough memory, there simply may not be enough available, due to memory consumed by other processes. In this case, you may see an `Out of memory` message, or self-compilation may become extremely slow. The only fix is to make sure that enough memory is available.

Possible Errors

- If you have errors running `latex`, you can skip building the documentation by using `make all-no-docs`.
- The C compiler may not be able to find the GnuMP header file, `gmp.h` leading to an error like the following.

  ```
  platform/darwin.h:26:36: /usr/local/include/gmp.h: No such file or directory
  ```

  The solution is to install (or build) the GnuMP on your machine. If you install it at a different location, put the new path in `runtime/platform/<os>.h`.
- The following error indicates that a binary version of MLton could not be found in your path.

  ```
  .../upgrade-basis: mlton: command not found
  Error: cannot upgrade basis because the compiler doesn't work
  make[3]: *** [upgrade-basis.sml] Error 1
  ```

  You need to have `mlton` in your path to build MLton from source.

  During the build process, there are various times that the `Makefiles` look for a `mlton` in your path and in `src/build/bin`. It is OK if the latter doesn't exist when the build starts; it is the target being built. While not finding `build/bin/mlton` also results in `mlton: command not found` error messages, such errors are benign and will not abort the build. Failure to find a `mlton` in your path will abort the build.

* Mac OS X executables do not seem to like static libraries to have a different path location at runtime compared to when the executable was built. For example, the binary package for Mac OS X unpacks to `/usr`. If you try to install it in `/usr/local` you may get the following errors:

  ```
  /usr/bin/ld: table of contents for archive:
  /usr/local/lib/mlton/self/libmlton.a is out of date;
  rerun ranlib(1) (can't load from it)
  ```

  Although running `ranlib` seems like the right thing to do, it doesn't actually resolve the problem. Best bet is to install in `/usr` and then either live with this location, or build MLton yourself and install in `/usr/local`.

Compiling with SML/NJ

To compile with SML/NJ, run `make nj-mlton` from within the root directory of the sources. You must use a recent version of SML/NJ. First, the `Makefile` calls `mllex` and `mlyacc` to build the lexer and parser. Then, it calls SML/NJ with the appropriate `sources.cm` file. Building with SML/NJ takes some time (roughly 10 minutes on a 1.6GHz machine). Unless you are doing compiler development and need rapid recompilation, we recommend compiling with MLton.

Last edited on 2005-12-02 01:44:46 by StephenWeeks.

Serialization

Standard ML does not have built-in support for serialization. Here are papers that describes a user-level approach.

- Elsman04
- Kennedy04

Last edited on 2005-12-02 01:46:34 by StephenWeeks.

ShowBasis

MLton has a flag, `-show-basis` *file*, that causes MLton to pretty print to *file* the basis defined by the input program. For example, if `foo.sml` contains

```
fun f x = x + 1
```

then `mlton -show-basis foo.basis foo.sml` will create `foo.basis` with the following contents.

```
val f: int -> int
```

If you only want to see the basis and do not wish to compile the program, you can call MLton with `-stop tc`.

Displaying signatures

When displaying signatures, MLton prefixes types defined in the signature them with `?.` to distinguish them from types defined in the environment. For example,

```
signature SIG =
   sig
      type t
      val x: t * int -> unit
   end
```

is displayed as

```
signature SIG =
   sig
      type t = ?.t
      val x: (?.t * int) -> unit
   end
```

Notice that `int` occurs without the `?.` prefix.

MLton also uses a canonical name for each type in the signature, and that name is used everywhere for that type, no matter what the input signature looked like. For example:

```
signature SIG =
   sig
      type t
      type u = t
      val x: t
      val y: u
   end
```

is displayed as

```
signature SIG =
   sig
      type t = ?.t
      type u = ?.t
      val x: ?.t
```

```
      val y: ?.t
   end
```

Canonical names are always relative to the "top" of the signature, even when used in nested substructures. For example:

```
signature S =
   sig
      type t
      val w: t
      structure U:
         sig
            type u
            val x: t
            val y: u
         end
      val z: U.u
   end
```

is displayed as

```
signature S =
   sig
      type t = ?.t
      val w: ?.t
      val z: ?.U.u
      structure U:
         sig
            type u = ?.U.u
            val x: ?.t
            val y: ?.U.u
         end
   end
```

Displaying structures

When displaying structures, MLton uses signature constraints wherever possible, combined with `where type` clauses to specify the meanings of the types defined within the signature.

```
signature SIG =
   sig
      type t
      val x: t
   end
structure S: SIG =
   struct
      type t = int
      val x = 13
   end
structure S2:> SIG = S
```

is displayed as

```
structure S: SIG
             where type t = int
structure S2: SIG
              where type t = S2.t
```

```
signature SIG =
   sig
      type t = ?.t
      val x: ?.t
   end
```

Last edited on 2005-12-02 01:48:03 by StephenWeeks.

ShowProf

If an executable is compiled for profiling, then it accepts a special command-line runtime system argument, `show-prof`, that outputs information about the source functions that are profiled. Normally, this information is used by `mlprof`. This page documents the `show-prof` output format, and is intended for those working on the profiler internals.

The `show-prof` output is ASCII, and consists of a sequence of lines.

- The magic number of the executable.
- The number of source names in the executable.
- A line for each source name giving the name of the function, a tab, the filename of the file containing the function, a colon, a space, and the line number that the function starts on in that file.
- The number of (split) source functions.
- A line for each (split) source function, where each line consists of a source-name index (into the array of source names) and a successors index (into the array of split-source sequences, defined below).
- The number of split-source sequences.
- A line for each split-source sequence, where each line is a space separated list of (split) source functions.

The latter two arrays, split sources and split-source sequences, define a directed graph, which is the call-graph of the program.

Last edited on 2006-10-23 23:29:54 by StephenWeeks.

Shrink

Shrink is a rewrite pass for the SSA and SSA2 IntermediateLanguages, invoked from every optimization pass (see SSASimplify and SSA2Simplify).

Description

This pass implements a whole family of compile-time reductions, like:

- `#1(a, b) --> a`
- `case C x of C y => e  --> let y = x in e`
- constant folding, copy propagation
- eta blocks
- tuple reconstruction elimination

Implementation

shrink.sig shrink.fun
shrink.sig shrink.fun

Details and Notes

The Shrink pass is run after every SSA and SSA2 optimization pass.

The Shrink implementation also includes functions to eliminate unreachable blocks from a SSA or SSA2 program or function. The Shrink pass does not guarantee to eliminate all unreachable blocks. Doing so would unduly complicate the implementation, and it is almost always the case that all unreachable blocks are eliminated. However, a small number of optimization passes require that the input have no unreachable blocks (essentially, when the analysis works on the control flow graph and the rewrite iterates on the vector of blocks). These passes explicitly call `eliminateDeadBlocks`.

The Shrink pass has a special case to turn a non-tail call where the continuation and handler only do `Profile` statements into a tail call where the `Profile` statements precede the tail call.

Last edited on 2005-12-02 04:24:49 by StephenWeeks.

SimplifyTypes

SimplifyTypes is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This pass computes a "cardinality" of each datatype, which is an abstraction of the number of values of the datatype.

- Zero means the datatype has no values (except for bottom).
- One means the datatype has one value (except for bottom).
- Many means the datatype has many values.

This pass removes all datatypes whose cardinality is Zero or One and removes:

- components of tuples
- function args
- constructor args

which are such datatypes.

This pass marks constructors as one of:

- Useless: it never appears in a `ConApp`.
- Transparent: it is the only variant in its datatype and its argument type does not contain any uses of `array` or `vector`.
- Useful: otherwise

This pass also removes Useless and Transparent constructors.

Implementation

simplify-types.sig simplify-types.fun

Details and Notes

This pass must happen before polymorphic equality is implemented because

1. it will make polymorphic equality faster because some types are simpler
2. it removes uses of polymorphic equality that must return true

We must keep track of Transparent constructors whose argument type uses `array` because of datatypes like the following:

```
datatype t = T of t vector
```

Such a datatype has Cardinality.Many, but we cannot eliminate the datatype and replace the lhs by the rhs, i.e. we must keep the circularity around.

Must do similar things for `vectors`.

Also, to eliminate as many Transparent constructors as possible, for something like the following,

```
datatype t = T of u array
     and u = U of t vector
```

we (arbitrarily) expand one of the datatypes first. The result will be something like

```
datatype u = U of u array array
```

where all uses of `t` are replaced by `u array`.

Last edited on 2005-12-02 04:25:02 by StephenWeeks.

Sources

We maintain our sources with Subversion. You can view them on the web or access them with a subversion client. Anonymous read access is available via

```
svn://mlton.org/mlton
```

We use the standard repository layout, so you can check out the latest revision with

```
svn co svn://mlton.org/mlton/trunk mlton
```

Committers (you know who you are) can access via

```
svn+ssh://mlton.org/svnroot/
```

Committers can check out the trunk with

```
svn co svn+ssh://mlton.org/svnroot/mlton/trunk mlton
```

Commit email

All commits are sent to `MLton-commit@mlton.org` (subscribe, archive), which is only for commit email. Discussion should go to MLton@mlton.org.

If the first line of a commit log message begins with "`MAIL `", then the commit message will be sent with the subject as the rest of that first line, and will also be sent to [mailto:MLton@mlton.org MLton@mlton.org].

Changelog

See the changelog for a list of changes and bug fixes.

CVS

Prior to 20050730, we used CVS. We left the CVS server up until 20060809, at which point it was taken down.

Last edited on 2006-08-10 21:16:02 by StephenWeeks.

SpaceSafety

Informally, space safety is a property of a language implementation that asymptotically bounds the space used by a running program.

References

- Chapter 12 of Appel92

Last edited on 2004-12-30 20:20:57 by StephenWeeks.

Stabilizers

Installation

- Stabilizers currently require the MLton sources, this should be fixed by the next release

License

- Stabilizers are released under the MLton License

Instructions

- Download and build a source copy of MLton
- Extract the tar.gz file attached to this wiki page
- Some examples are provided in the "examples/" sub directory, more examples will be added to this page in the following week

Bug reports / Suggestions

- Please send any errors you encounter to schatzp and lziarek at cs.purdue.edu
- We are looking to expand the usability of stabilizers
- Please send any suggestions and desired functionality to the above email addresses

Note

- This is an alpha release. We expect to have another release shortly with added functionality soon
- More documentation, such as signatures and descriptions of functionality, will be forthcoming

Documentation

```
signature STABLE =
  sig
     type checkpoint

     val stable: ('a -> 'b) -> ('a -> 'b)
     val stabilize: unit -> 'a

     val stableCP: (('a -> 'b) * (unit -> unit)) ->
                   (('a -> 'b) *  checkpoint)
     val stabilizeCP: checkpoint -> unit

     val unmonitoredAssign: ('a ref * 'a) -> unit
     val monitoredAssign: ('a ref * 'a) -> unit
  end
```

`Stable` provides functions to manage stable sections.

- `type checkpoint`
  handle used to stabilize contexts other than the current one.
- `stable f`

returns a function identical to `f` that will execute within a stable section.

- `stabilize ()`
  unrolls the effects made up to the current context to at least the nearest enclosing {{stable}}} section. These effects may have propagated to other threads, so all affected threads are returned to a globally consistent previous state. The return is undefined because control cannot resume after stabilize is called.
- `stableCP (f, comp)`
  returns a function `f'` and checkpoint tag `cp`. Function `f'` is identical to `f` but when applied will execute within a stable section. `comp` will be executed if `f'` is later stabilized. `cp` is used by `stabilizeCP` to stabilize a given checkpoint.
- `stabilizeCP cp`
  same as stabilize except that the (possibly current) checkpoint to stabilize is provided.
- `unmonitoredAssign (r, v)`
  standard assignment (`:=`). The version of CML distributed rebinds `:=` to a monitored version so interesting effects can be recorded.
- `monitoredAssign (r, v)`

the assignment operator that should be used in programs that use stabilizers. `:=` is rebound to this by including CML.

stabilizers_alpha_2006-10-09.tar.gz

Last edited on 2006-10-09 20:34:58 by PhilipSchatz.

StandardML

Standard ML (SML) is a programming language that combines excellent support for rapid prototyping, modularity, and development of large programs, with performance approaching that of C.

SML Resources

- Tutorials
- Books
- Implementations

Aspects of SML

- DefineTypeBeforeUse
- EqualityType
- EqualityTypeVariable
- GenerativeDatatype
- GenerativeException
- Identifier
- OperatorPrecedence
- Overloading
- PolymorphicEquality
- ValueRestriction

Using SML

- Fixpoints
- ForLoops
- FunctionalRecordUpdate
- InfixingOperators
- Lazy
- ObjectOrientedProgramming
- OptionalArguments
- Printf
- PropertyList
- Serialization
- StyleGuide
- UniversalType

Programming in SML

- Emacs
- Enscript

Notes

- History of SML
- Regions

Related Languages

- Alice
- OCaml

Last edited on 2006-08-03 11:19:55 by VesaKarvonen.

StandardMLBooks

Introductory Books

- Elements of ML Programming
- ML For the Working Programmer
- Introduction to Programming using SML
- The Little MLer

Applications

- Unix System Programming with Standard ML

Reference Books

- The Standard ML Basis Library
- The Definition of Standard ML (Revised)

Related Topics

- Concurrent Programming in ML
- Purely Functional Data Structures

Last edited on 2005-05-19 19:50:12 by StephenWeeks.

StandardMLGotchas

This page contains brief explanations of some recurring sources of confusion and problems that SML newbies encounter.

The and keyword

It is a common mistake to misuse the `and` keyword or to not know how to introduce mutually recursive definitions. The purpose of the `and` keyword is to introduce mutually recursive definitions of functions and datatypes. For example,

```
fun isEven 0w0 = true
  | isEven 0w1 = false
  | isEven n = isOdd (n-0w1)
and isOdd 0w0 = false
  | isOdd 0w1 = true
  | isOdd n = isEven (n-0w1)
```

and

```
datatype decl = VAL of id * pat * expr
           (* | ... *)
     and expr = LET of decl * expr
           (* | ... *)
```

You can also use `and` as a shorthand in a couple of other places, but it is not obligatory.

Constructed patterns

It is a common mistake to forget to parenthesize constructed patterns in `fun` bindings. Consider the following invalid definition:

```
fun length nil = 0
  | length h :: t = 1 + length t
```

The pattern `h :: t` needs to be parenthesized:

```
fun length nil = 0
  | length (h :: t) = 1 + length t
```

The parentheses are needed, because a `fun` definition may specify multiple curried formal parameters.

The same applies to nonfix constructors. For example, the parentheses in

```
fun valOf NONE = raise Option
  | valOf (SOME x) = x
```

are required. However, the outermost constructed pattern in a `fn` or `case` expression need not be parenthesized. Both

```
val valOf = fn NONE => raise Option
             | SOME x => x
```

and

```
fun valOf x = case x of
                NONE => raise Option
              | SOME x => x
```

are fine.

Declarations and expressions

It is a common mistake to confuse expressions and declarations. Normally a SML source file should only contain declarations. The following are declarations:

```
datatype dt = ...
fun f ... = ...
functor Fn (...) = ...
infix ...
infixr ...
local ... in ... end
nonfix ...
open ...
signature SIG = ...
structure Struct = ...
type t = ...
val v = ...
```

Note that

```
let ... in ... end
```

isn't a declaration.

To specify a side-effecting computation in a source file, you can write:

```
val () = ...
```

Equality types

SML has a fairly intricate built-in notion of equality. See EqualityType and EqualityTypeVariable for a thorough discussion.

Nested cases

It is a common mistake to write nested case expressions without the necessary parentheses. See UnresolvedBugs for a discussion.

(op *)

It used to be a common mistake to parenthesize `op *` as `(op *)`. Before SML'97, `*)` was considered a comment terminator in SML and caused a syntax error. At the time of writing, SML/NJ still rejects the code. An extra space may be used for portability: `(op * )`. However, parenthesizing `op` is redundant, even though it is a widely used convention.

Overloading

A number of standard operators (+, -, ~, *, <, >, ...) and numeric constants are overloaded for some of the numeric types (`int`, `real`, `word`). It is a common surprise that definitions using overloaded operators such as

```
fun min (x, y) = if y < x then y else x
```

are not overloaded themselves. SML doesn't really support (user-defined) overloading or other forms of ad hoc polymorphism. In cases such as the above where the context doesn't resolve the overloading, expressions using overloaded operators or constants get assigned a default type. The above definition gets the type

```
val min : int * int -> int
```

See Overloading and TypeIndexedValues for further discussion.

Semicolons

It is a common mistake to use redundant semicolons in SML code. This is probably caused by the fact that in a SML REPL, a semicolon (and enter) is used to signal the implementation that it should evaluate the preceding chunk of code as a unit. In SML source files, semicolons are really needed in only two places. Namely, in expressions of the form

```
(exp ; ... ; exp)
```

and

```
let ... in exp ; ... ; exp end
```

Stale bindings

Unresolved records

Value restriction

See ValueRestriction.

Last edited on 2006-08-14 10:17:53 by VesaKarvonen.

StandardMLHistory

Standard ML grew out of ML in the early 1980s.

For an excellent overview of SML's history, see Appendix F of the Definition.

For an overview if its history before 1982, see How ML Evolved.

Last edited on 2005-06-20 21:44:44 by StephenWeeks.

StandardMLImplementations

There are a number of implementations of Standard ML, from interpreters, to byte-code compilers, to incremental compilers, to whole-program compilers.

- HaMLet
- ML Kit
- MLton
- Moscow ML
- Poly/ML
- Poplog
- SML/NJ
- SML.NET
- TILT

Not Actively Maintained

- Edinburgh ML
- MLj
- MLWorks
- TIL

Last edited on 2005-12-02 02:39:34 by StephenWeeks.

StandardMLPortability

Technically, SML'97 as defined in the Definition requires only a a minimal initial basis, which, while including the types `int`, `real`, `char`, and `string`, need have no operations on those base types. Hence, the only observable output of an SML'97 program is termination or raising an exception. Most SML compilers should agree there, to the degree each agrees with the Definition. See UnresolvedBugs for MLton's very few corner cases.

Realistically, a program needs to make use of the Basis Library. Within the Basis Library, there are numerous places where the behavior is implementation dependent. For a trivial example:

```
val _ = valOf (Int.maxInt)
```

may either raise the `Option` exception (if `Int.maxInt == NONE`) or may terminate normally. The default Int/Real/Word sizes are the biggest implementation dependent aspect; so, one implementation may raise `Overflow` while another can accommodate the result. Also, maximum array and vector lengths are implementation dependent. Interfacing with the operating system is a bit murky, and implementations surely differ in handling of errors there.

Last edited on 2005-12-02 04:25:49 by StephenWeeks.

StandardMLTutorials

- A Gentle Introduction to ML. Andrew Cummings.
- Programming in Standard ML '97: An Online Tutorial. Stephen Gilmore.
- Programming in Standard ML. Robert Harper.
- Essentials of Standard ML Modules. Mads Tofte.

Last edited on 2005-05-10 15:17:53 by StephenWeeks.

StephenWeeks

I am a consultant based in the San Francisco Bay Area.

home page

You can email me at sweeks@sweeks.com.

My license plate.

image

Last edited on 2004-11-10 21:59:17 by StephenWeeks.

StyleGuide

These conventions are chosen so that inertia is towards modularity, code reuse and finding bugs early, *not* to save typing.

- SyntacticConventions

Last edited on 2004-11-14 23:23:24 by StephenWeeks.

Subversion

Subversion is a version control system designed to replace CVS. The MLton project uses Subversion to maintain its source code.

- Version Control with Subversion, a free online book

Last edited on 2005-07-30 21:29:05 by StephenWeeks.

SureshJagannathan

I am an Associate Professor at the Department of Computer Science at Purdue University. My research focus is in programming language design and implementation, concurrency, and distributed systems. I am interested in various aspects of MLton, mostly related to (in no particular order): (1) control-flow analysis (2) representation strategies (e.g., flattening), (3) IR formats, and (4) extensions for distributed programming.

Please see my Home page for more details.

Last edited on 2004-11-20 21:09:49 by SureshJagannathan.

Survey

The 2005 MLton Survey is closed. Please check this space in January 2006 for our next survey. Thanks to all who responded.

Last edited on 2005-02-07 01:08:25 by StephenWeeks.

SurveyDone

Success. Thank you for submitting a survey.

Last edited on 2005-01-05 19:41:21 by StephenWeeks.

Swerve

Swerve is an HTTP server written in SML, originally developed with SML/NJ. RayRacine ported Swerve to MLton in January 2005.

download the port

Excerpt from the included `README`:

Total testing of this port consisted of a successful compile, startup, and serving one html page with one gif image. Given that the original code was throughly designed and implemented in a thoughtful manner and I expect it is quite usable modulo a few minor bugs introduced by my porting effort.

Last edited on 2005-10-24 00:55:35 by PhilipSchatz.

SyntacticConventions

Here are a number of syntactic conventions useful for programming in SML.

1. General
2. Identifiers
3. Types
4. Core
5. Signatures
6. Structures
7. Functors

General

- A line of code never exceeds 80 columns.
- Only split a syntactic entity across multiple lines if it doesn't fit on one line within 80 columns.
- Use alphabetical order wherever possible.
- Avoid redundant parentheses.
- When using `:`, there is no space before the colon, and a single space after it.

Identifiers

- Variables, record labels and type constructors begin with and use small letters, using capital letters to separate words.

```
cost
maxValue
```

- Variables that represent collections of objects (lists, arrays, vectors, ...) are often suffixed with an `s`.

```
xs
employees
```

- Constructors, structure identifiers, and functor identifiers begin with a capital letter.

```
Queue
LinkedList
```

- Signature identifiers are in all capitals, using _ to separate words.

```
LIST
BINARY_HEAP
```

Types

- Alphabetize record labels. In a record type, there are spaces after colons and commas, but not before colons or commas, or at the delimiters `{` and `}` .

```
{bar: int, foo: int}
```

- Only split a record type across multiple lines if it doesn't fit on one line. If a record type must be split over multiple lines, put one field per line.

```
{bar: int,
 foo: real * real,
```

```
 zoo: bool}
```

- In a tuple type, there are spaces before and after each `*`.

  ```
  int * bool * real
  ```
- Only split a tuple type across multiple lines if it doesn't fit on one line. In a tuple type split over multiple lines, there is one type per line, and the `*`s go at the beginning of the lines.

  ```
  int
  * bool
  * real
  ```

  It may also be useful to parenthesize to make the grouping more apparent.

  ```
  (int
   * bool
   * real)
  ```
- In an arrow type split over multiple lines, put the arrow at the beginning of its line.

  ```
  int * real
  -> bool
  ```

  It may also be useful to parenthesize to make the grouping more apparent.

  ```
  (int * real
   -> bool)
  ```
- Avoid redundant parentheses.
    - ♦ Arrow types associate to the right, so write

      ```
      a -> b -> c
      ```

      not

      ```
      a -> (b -> c)
      ```
    - ♦ Type constructor application associates to the left, so write

      ```
      int ref list
      ```

      not

      ```
      (int ref) list
      ```
    - ♦ Type constructor application binds more tightly than a tuple type, so write

      ```
      int list * bool list
      ```

      not

      ```
      (int list) * (bool list)
      ```
    - ♦ Tuple types bind more tightly than arrow types, so write

      ```
      int * bool -> real
      ```

      not

      ```
      (int * bool) -> real
      ```

Core

- A core expression or declaration split over multiple lines does not contain any blank lines.
- A record field selector has no space between the `#` and the record label. So, write

  ```
  #foo
  ```

  not

  ```
  # foo
  ```
- A tuple has a space after each comma, but not before, and not at the delimiters `( )`.

  ```
  (e1, e2, e3)
  ```
- A tuple split over multiple lines has one element per line, and the commas go at the end of the lines.

  ```
  (e1,
   e2,
   e3)
  ```
- A list has a space after each comma, but not before, and not at the delimiters `[ ]`.

  ```
  [e1, e2, e3]
  ```
- A list split over multiple lines has one element per line, and the commas at the end of the lines.

  ```
  [e1,
   e2,
   e3]
  ```
- A record has spaces before and after =, a space after each comma, and no space at the delimiters. Field names appear in alphabetical order.

  ```
  {bar = 13, foo = true}
  ```
- A sequence expression has a space after each semicolon, but not before.

  ```
  (e1; e2; e3)
  ```
- A sequence expression split over multiple lines has one expression per line, and the semicolons at the beginning of lines. Lisp and Scheme programmers may find this hard to read at first.

  ```
  (e1
   ; e2
   ; e3)
  ```

  Rationale: this makes it easy to visually spot the beginning of each expression, which becomes more valuable as the expressions themselves are split across multiple lines.
- An application expression has a space between the function and the argument. There are no parens unless the argument is a tuple (in which case the parens are really part of the tuple, not the application).

  ```
  f a
  f (a1, a2, a3)
  ```
- Avoid redundant parentheses. Application associates to left, so write

  ```
  f a1 a2 a3
  ```

  not

```
((f a1) a2) a3
```

* Infix operators have a space before and after the operator.

```
x + y
x * y - z
```

* Avoid redundant parentheses. Use OperatorPrecedence. So, write

```
x + y * z
```

not

```
x + (y * z)
```

* An `andalso` expression split over multiple lines has the `andalso` at the beginning of subsequent lines.

```
e1
andalso e2
andalso e3
```

* A `case` expression is indented as follows

```
case e1 of
   p1 => e1
 | p2 => e2
 | p3 => e3
```

* A `datatype`'s constructors are alphabetized.

```
datatype t = A | B | C
```

* A `datatype` declaration has a space before and after each `|`.

```
datatype t = A | B of int | C
```

* A `datatype` split over multiple lines has one constructor per line, with the `|` at the beginning of lines and the constructors beginning 3 columns to the right of the `datatype`.

```
datatype t =
   A
 | B
 | C
```

* A `fun` declaration may start its body on the subsequent line, indented 3 spaces.

```
fun f x y =
   let
      val z = x + y + z
   in
      z
   end
```

* An `if` expression is indented as follows.

```
if e1
   then e2
else e3
```

* A sequence of `if-then-elses` is indented as follows.

```
if e1
   then e2
else if e3
   then e4
```

```
else if e5
   then e6
else e7
```

- A `let` expression has the `let`, `in`, and `end` on their own lines, starting in the same column. Declarations and the body are indented 3 spaces.

```
let
   val x = 13
   val y = 14
in
   x + y
end
```

- A `local` declaration has the `local`, `in`, and `end` on their own lines, starting in the same column. Declarations are indented 3 spaces.

```
local
   val x = 13
in
   val y = x
end
```

- An `orelse` expression split over multiple lines has the `orelse` at the beginning of subsequent lines.

```
e1
orelse e2
orelse e3
```

- A `val` declaration has a space before and after the `=`.

```
val p = e
```

- A `val` declaration can start the expression on the subsequent line, indented 3 spaces.

```
val p =
   if e1 then e2 else e3
```

Signatures

- A `signature` declaration is indented as follows.

```
signature FOO =
   sig
      val x: int
   end
```

- A `val` specification has a space after the colon, but not before.

```
val x: int
```

Exception: in the case of operators (like `+`), there is a space before the colon to avoid lexing the colon as part of the operator.

```
val + : t * t -> t
```

- Alphabetize specifications in signatures.

```
sig
   val x: int
   val y: bool
```

```
end
```

Structures

- A `structure` declaration has a space on both sides of the =.

```
structure Foo = Bar
```

- A `structure` declaration split over multiple lines is indented as follows.

```
structure S =
   struct
      val x = 13
   end
```

- Declarations in a `struct` are separated by blank lines.

```
struct
   val x =
      let
         y = 13
      in
         y + 1
      end

   val z = 14
end
```

Functors

- A `functor` declaration has spaces after each `:` (or `:>`) but not before, and a space before and after the =. It is indented as follows

```
functor Foo (S: FOO_ARG): FOO =
   struct
       val x = S.x
   end
```

Exception: a functor declaration in a file to itself can omit the indentation to save horizontal space.

```
functor Foo (S: FOO_ARG): FOO =
struct

val x = S.x

end
```

In this case, there should be a blank line after the `struct` and before the `end`.

Last edited on 2006-08-24 23:26:33 by MichaelNorrish.

SystemInfo

Python Version
: 2.3.5 (#2, Sep 4 2005, 22:01:42) [GCC 3.3.5 (Debian 1:3.3.5-13)]

MoinMoin Version
: Release 1.2.3 [Revision 1.186]

Number of pages
: 328

Number of system pages
: 2

Number of backup versions
: 1639

Accumulated page sizes
: 651728

Entries in edit log
: 2136 (212477 bytes)

Event log
: 64168355 bytes

Global extension macros
: AbandonedPages, BR, FootNote, Form, FullSearch, GetText, Include, Navigation, OrphanedPages, PageHits, PageSize, RandomPage, RandomQuote, RecentChanges, ShowSmileys, StatsChart, SystemAdmin, TableOfContents, TeudView, WantedPages

Local extension macros
: Cite, Div, DownloadSVN, Form, Improvement, IncludeSVN, Input, Span, TextArea, ViewCVS, ViewCVSDir, ViewSVN, ViewSVNDir

Global extension actions
: AttachFile, DeletePage, LikePages, LocalSiteMap, RenamePage, SpellCheck, links, rss_rc, titleindex

Local extension actions
: AllLinks

Installed processors
: CSV, Colorize

Last edited on 2004-10-26 01:42:46 by StephenWeeks.

TILT

TILT is a Standard ML Compiler.

Last edited on 2006-08-18 18:34:10 by StephenWeeks.

Talk

The MLton Standard ML Compiler

Henry Cejtin, Matthew Fluet, Suresh Jagannathan, Stephen Weeks

Next

Last edited on 2004-12-01 16:48:10 by MatthewFluet.

TalkDiveIn

Dive In

- to Development
- to Documentation
- to Download

Prev

Last edited on 2005-11-14 23:13:23 by MatthewFluet.

TalkFolkLore

Folk Lore

- Defunctorization and monomorphisation are feasible
- Global control-flow analysis is feasible
- Early closure conversion is feasible

Prev Next

Last edited on 2004-12-01 18:35:55 by MatthewFluet.

TalkFromSMLTo

From Standard ML to S-T F-O IL

* What issues arise when translating from Standard ML into an intermediate language?

Prev Next

Last edited on 2004-12-01 18:39:02 by MatthewFluet.

TalkHowHigherOrder

Higher-order Functions

- How does one represent SML's higher-order functions?
- MLton's answer: defunctionalize

Prev Next

See ClosureConvert.

Last edited on 2004-12-01 18:36:01 by MatthewFluet.

TalkHowModules

Modules

- How does one represent SML's modules?
- MLton's answer: defunctorize

Prev Next

See Elaborate.

Last edited on 2004-12-01 18:36:07 by MatthewFluet.

TalkHowPolymorphism

Polymorphism

- How does one represent SML's polymorphism?
- MLton's answer: monomorphise

Prev Next

See Monomorphise.

Last edited on 2004-12-01 18:36:12 by MatthewFluet.

TalkMLtonApproach

MLton's Approach

- whole-program optimization using a simply-typed, first-order intermediate language
- ensures programs are not penalized for exploiting abstraction and modularity

Prev Next

Last edited on 2004-12-01 18:36:17 by MatthewFluet.

TalkMLtonFeatures

MLton Features

- Supports full Standard ML language and Basis Library
- Generates standalone executables
- Extensions
  - Foreign function interface (SML to C, C to SML)
  - ML Basis system for programming in the very large
  - Extension libraries

Prev Next

See Features.

Last edited on 2005-01-28 21:49:50 by MatthewFluet.

TalkMLtonHistory

MLton History

April 1997	Stephen Weeks wrote a defunctorizer for SML/NJ
Aug. 1997	Begin independent compiler (`smlc`)
Oct. 1997	Monomorphiser
Nov. 1997	Polyvariant higher-order control-flow analysis (10,000 lines)
March 1999	First release of MLton (48,006 lines)
Jan. 2002	MLton at 102,541 lines
Jan. 2003	MLton at 112,204 lines
Jan. 2004	MLton at 122,299 lines
Nov. 2004	MLton at 141,311 lines

Prev Next

See History.

Last edited on 2004-12-01 18:42:32 by MatthewFluet.

TalkStandardML

Standard ML

- a high-level language makes
  - a programmer's life easier
  - a compiler writer's life harder
- perceived overheads of features discourage their use
  - higher-order functions
  - polymorphic datatypes
  - separate modules

Prev Next

Also see Standard ML.

Last edited on 2005-01-18 15:02:29 by MatthewFluet.

TalkTemplate

Title

- Bullet
- Bullet

Prev Next

Last edited on 2004-12-01 18:59:26 by MatthewFluet.

TalkWholeProgram

Whole Program Compiler

- Each of these techniques requires whole-program analysis
- But, additional benefits:
    - eliminate (some) variablity in programming styles
    - specialize representations
    - simplifies and improves runtime system

Prev Next

Last edited on 2004-12-01 18:40:55 by MatthewFluet.

ToMachine

ToMachine is a translation pass from the RSSA IntermediateLanguage to the Machine IntermediateLanguage.

Description

This pass converts from a RSSA program into a Machine program.

It uses AllocateRegisters, Chunkify, and ParallelMove.

Implementation

backend.sig backend.fun

Details and Notes

Because the MLton runtime system is shared by all codegens, it is most convenient to decide on stack layout *before* any codegen takes over. In particular, we compute all the stack frame info for each RSSA function, including stack size, garbage collector masks for each frame, etc. To do so, the Machine IntermediateLanguage imagines an abstract machine with an infinite number of (pseudo-)registers of every size. A liveness analysis determines, for each variable, whether or not it is live across a point where the runtime system might take over (for example, any garbage collection point) or a non-tail call to another RSSA function. Those that are live go on the stack, while those that aren't live go into psuedo-registers. From this information, we know all we need to about each stack frame. On the downside, nothing further on is allowed to change this stack info; it is set in stone.

Last edited on 2005-12-02 03:34:28 by StephenWeeks.

ToRSSA

ToRSSA is a translation pass from the SSA2 IntermediateLanguage to the RSSA IntermediateLanguage.

Description

This pass converts a SSA2 program into a RSSA program.

It uses PackedRepresentation.

Implementation

ssa-to-rssa.sig ssa-to-rssa.fun

Details and Notes

Last edited on 2005-12-02 02:51:27 by StephenWeeks.

ToSSA2

ToSSA2 is a translation pass from the SSA IntermediateLanguage to the SSA2 IntermediateLanguage.

Description

This pass is a simple conversion from a SSA program into a SSA2 program.

The only interesting portions of the translation are:

- an SSA `ref` type becomes an object with a single mutable field
- `array`, `vector`, and `ref` are eliminated in favor of select and updates
- `Case` transfers separate discrimination and constructor argument selects

Implementation

ssa-to-ssa2.sig ssa-to-ssa2.fun

Details and Notes

Last edited on 2005-12-02 02:53:59 by StephenWeeks.

TomMurphy

Tom Murphy VII is a long time MLton user and occasional contributor. He works on programming languages for his PhD work at Carnegie Mellon in Pittsburgh, USA.

Home page

Last edited on 2005-09-27 05:20:33 by TomMurphy.

TrustedGroup

This list of users is for AccessControl.

- HenryCejtin
- JesperLouisAndersen
- MatthewFluet
- StephenWeeks
- VilleLaurikari

Last edited on 2006-06-10 16:06:57 by StephenWeeks.

TypeChecking

MLton's type checker follows the Definition of SML closely, so you may find differences between MLton and other SML compilers that do not follow the Definition so closely. In particular, SML/NJ has many deviations from the Definition -- please see SMLNJDeviations for those that we are aware of.

In some respects MLton's type checker is more powerful than other SML compilers, so there are programs that MLton accepts that are rejected by some other SML compilers. These kinds of programs fall into a few simple categories.

- MLton resolves flexible record patterns using a larger context than many other SML compilers. For example, MLton accepts the following.

```
fun f {x, ...} = x
val _ = f {x = 13, y = "foo"}
```

- MLton uses as large a context as possible to resolve the type of variables constrained by the value restriction to be monotypes. For example, MLton accepts the following.

```
structure S:
   sig
      val f: int -> int
   end =
   struct
      val f = (fn x => x) (fn y => y)
   end
```

Type error messages

To aid in the understanding of type errors, MLton's type checker displays type errors differently than other SML compilers. In particular, when two types are different, it is important for the programmer to easily understand why they are different. So, MLton displays only the differences between two types that don't match, using underscores for the parts that match. For example, if a function expects `real * int` but gets `real * real`, the type error message would look like

```
expects: _ * [int]
but got: _ * [real]
```

As another aid to spotting differences, MLton places brackets `[]` around the parts of the types that don't match. A common situation is when a function receives a different number of arguments than it expects, in which case you might see an error like

```
expects: [int * real]
but got: [int * real * string]
```

The brackets make it easy to see that the problem is that the tuples have different numbers of components -- not that the components don't match. Contrast that with a case where a function receives the right number of arguments, but in the wrong order.

```
expects: [int] * [real]
but got: [real] * [int]
```

Here the brackets make it easy to see that the components do not match.

We appreciate feedback on any type error messages that you find confusing, or suggestions you may have for improvements to error messages.

The shortest/most-recent rule for type names

In a type error message, MLton often has a number of choices in deciding what name to use for a type. For example, in the following type-incorrect program

```
type t = int
fun f (x: t) = x
val _ = f "foo"
```

MLton reports

```
Error: z.sml 3.9.
  Function applied to incorrect argument.
    expects: [t]
    but got: [string]
    in: f "foo"
```

MLton could have reported `expects: [int]` instead of `expects: [t]`. However, MLton uses the shortest/most-recent rule in order to decide what type name to display. This rule means that, at the point of the error, MLton first looks for the shortest name for a type in terms of number of structure identifiers (e.g. `foobar` is shorter than `A.t`). Next, if there are multiple names of the same length, then MLton uses the most recently defined name. It is this tiebreaker that causes MLton to prefer `t` to `int` in the above example.

In signature matching, most recently defined is taken to include all of the definitions introduced by the structure. For example

```
structure S:
   sig
      val x: int
   end =
   struct
      type t = int
      val x = "foo"
   end
```

MLton reports the error message

```
Error: z.sml 2.4.
  Variable type in structure disagrees with signature.
    variable: x
    structure: [string]
    signature: [t]
```

in which the `[t]` refers to the type defined in the structure, since that is more recent than the definition of `int`.

In signatures with type equations, this can be somewhat confusing. For example.

```
structure S:
   sig
      type t
      type u = t
```

```
end =
struct
   type t = int
   type u = char
end
```

MLton reports the error

```
Error: z.sml 2.4.
  Type definition in structure disagrees with signature.
    type: u
    structure: [u]
    signature: [t]
```

This error reflects the fact that the signature requires type `u` to equal `t`, but that in the structure, `u` is defined to be `char`, whose most-recent name is `u`, while the signature requires `u` to be `int`, whose most-recent name is `t`.

Last edited on 2005-12-02 04:26:13 by StephenWeeks.

TypeConstructor

In Standard ML, a type constructor is a function from types to types. Type constructors can be *nullary*, meaning that they take no arguments, as in `char`, `int`, and `real`. Type constructors can be *unary*, meaning that they take one argument, as in `array`, `list`, and `vector`. A program can define a new type constructor in two ways: a `type` definition or a `datatype` declaration. User-defined type constructors can can take any number of arguments.

```
datatype t = T of int * real            (* 0 arguments *)
type 'a t = 'a * int                    (* 1 argument *)
datatype ('a, 'b) t = A | B of 'a * 'b  (* 2 arguments *)
type ('a, 'b, 'c) t = 'a * ('b  -> 'c)  (* 3 arguments *)
```

Here are the syntax rules for type constructor application.

- Type constructor application is written in postfix. So, one writes `int list`, not `list int`.
- Unary type constructors drop the parens, so one writes `int list`, not `(int) list`.
- Nullary type constructors drop the argument entirely, so one writes `int`, not `() int`.
- N-ary type constructors use tuple notation; for example, `(int, real) t`.
- Type constructor application associates to the left. So, `int ref list` is the same as `(int ref) list`.

Last edited on 2005-12-02 04:26:23 by StephenWeeks.

TypeIndexedValues

Standard ML does not support ad hoc polymorphism. This presents a challenge to programmers. The problem is that at first glance there seems to be no practical way to implement something like a function for converting a value of any type to a string or a function for computing a hash value for a value of any type. Fortunately there are ways to implement type-indexed values in SML as discussed in Yang98. Various articles such as Danvy98, Ramsey03, Elsman04, Kennedy04, and Benton05 also contain examples of type-indexed values.

Example: Converting any SML value to (roughly) SML syntax

NOTE: This is example is work-in-progress. Please send your comments to the MLton-user@mlton.org list.

Consider the problem of converting any SML value to a textual presentation that matches the syntax of SML as closely as possible. One solution is a type-indexed function that maps a given type to a function that maps any value (of the type) to its textual presentation. A type-indexed function like this can be useful for a variety of purposes. For example, one could use it to show debugging information. We'll call this function "`show`".

We'll do a fairly complete implementation of `show`. We do not distinguish infix and nonfix constructors, but that is not an intrinsic property of SML datatypes. We also don't reconstruct a type name for the value, although it would be particularly useful for functional values. To reconstruct type names, some changes would be needed and the reader is encouraged to consider how to do that. A more realistic implementation would use some pretty printing combinators to compute a layout for the result. This should be a relatively easy change (given a suitable pretty printing library). Cyclic values (through references and arrays) do not have a standard textual presentation and it is impossible to convert arbitrary functional values (within SML) to a meaningful textual presentation. Finally, it would also make sense to show sharing of references and arrays. We'll leave these improvements to an actual library implementation.

We'll make use of the fixpoint framework, which is actually also a simple type-indexed function. The following code assumes that the `Fix` structure and utilities are in scope.

Signature

Let's consider the design of the `SHOW` signature:

```
infixr -->

signature SHOW =
   sig
      type 'a t       (* complete type-index *)
      type 'a s       (* incomplete sum *)
      type ('a, 'k) p (* incomplete product *)
      type u          (* tuple or unlabelled product *)
      type l          (* record or labelled product *)

      val show : 'a t -> 'a -> string

      (* user-defined types *)
      val inj : ('a -> 'b) -> 'b t -> 'a t

      (* tuples and records *)
      val * : ('a, 'k) p * ('b, 'k) p -> (('a, 'b) product, 'k) p
```

```
      val U :            'a t -> ('a, u) p
      val L : string -> 'a t -> ('a, l) p

      val tuple  : ('a, u) p -> 'a t
      val record : ('a, l) p -> 'a t

      (* datatypes *)
      val + : 'a s * 'b s -> (('a, 'b) sum) s

      val C0 : string -> unit s
      val C1 : string -> 'a t -> 'a s

      val data : 'a s -> 'a t

      val Y : 'a t Fix.t

      (* exceptions *)
      val exn : exn t
      val regExn : (exn -> ('a * 'a s) option) effect

      (* some built-in type constructors *)
      val refc : 'a t -> 'a ref t
      val array : 'a t -> 'a array t
      val list : 'a t -> 'a list t
      val vector : 'a t -> 'a vector t
      val --> : 'a t * 'b t -> ('a -> 'b) t

      (* some built-in base types *)
      val string : string t
      val unit : unit t
      val bool : bool t
      val char : char t
      val int : int t
      val word : word t
      val real : real t
   end
```

While some details are shaped by the specific requirements of `show`, there are a number of (design) patterns that translate to other type-indexed values. The former kind of details are mostly shaped by the syntax of SML values that `show` is designed to produce. To this end, abstract types and phantom types are used to distinguish incomplete record, tuple, and datatype type-indices from each other and from complete type-indices. Also, names of record labels and datatype constructors need to be provided by the user.

Arbitrary user-defined datatypes

Perhaps the most important pattern is how the design supports arbitrary user-defined datatypes. A number of combinators together conspire to provide the functionality. First of all, to support new user-defined types, a combinator taking a conversion function to a previously supported type is provided:

```
val inj : ('a -> 'b) -> 'b t -> 'a t
```

An injection function is sufficient in this case, but in the general case, an embedding with injection and projection functions may be needed.

To support products (tuples and records) a product combinator is provided:

```
val * : ('a, 'k) p * ('b, 'k) p -> (('a, 'b) product, 'k) p
```

The second (phantom) type variable `'k` is there to distinguish between labelled and unlabelled products and the type `p` distinguishes incomplete products from complete type-indices of type `t`. Most type-indexed values do not need to make such distinctions.

To support sums (datatypes) a sum combinator is provided:

```
val + : 'a s * 'b s -> (('a, 'b) sum) s
```

Again, the purpose of the type `s` is to distinguish incomplete sums from complete type-indices of type `t`, which usually isn't necessary.

Finally, to support recursive datatypes, including sets of mutually recursive datatypes, a fixpoint tier is provided:

```
val Y : 'a t Fix.t
```

Together these combinators (with the more domain specific combinators `U`, `L`, `tuple`, `record`, `C0`, `C1`, and `data`) enable one to encode a type-index for any user-defined datatype.

Exceptions

The `exn` type in SML is a universal type into which all types can be embedded. SML also allows a program to generate new exception variants at run-time. Thus a mechanism is required to register handlers for particular variants:

```
val exn : exn t
val regExn : (exn -> ('a * 'a s) option) effect
```

The universal `exn` type-index then makes use of the registered handlers. The above particular form of handler, which converts an exception value to a value of some type and a type-index for that type (essentially an existential type) is designed to make it convenient to write handlers. To write a handler, one can conveniently reuse existing type-indices:

```
exception Int of int

local
   open Show
in
   val () = regExn (fn Int v => SOME (v, C1"Int" int)
                     | _     => NONE)
end
```

Note that a single handler may actually handle an arbitrary number of different exceptions.

Other types

Some built-in and standard types typically require special treatment due to their special nature. The most important of these are arrays and references, because cyclic data (ignoring closures) and observable sharing can only be constructed through them.

When arrow types are really supported, unlike in this case, they usually need special treatment due to the contravariance of arguments.

Lists and vectors require special treatment in the case of `show`, because of their special syntax. This isn't usually the case.

The set of base types to support also needs to be considered unless one exports an interface for constructing type-indices for entirely new base types.

Usage

Before going to the implementation, let's look at some examples. For the following examples, we'll assume a structure binding `Show :> SHOW`. If you want to try the examples immediately, just skip forward to the implementation.

To use `show`, one first needs a type-index, which is then given to `show`. To show a list of integers, one would use the type-index `list int`, which has the type `int list Show.t`:

```
val "[3, 1, 4]" =
    let open Show in show (list int) end
       [3, 1, 4]
```

Likewise, to show a list of lists of characters, one would use the type-index `list (list char)`, which has the type `char list list Show.t`:

```
val "[[#\"a\"], [#\"b\", #\"c\"], []]" =
    let open Show in show (list (list char)) end
       [[#"a"], [#"b", #"c"], []]
```

Handling standard types is not particularly interesting. It is more interesting to see how user-defined types can be handled. Although the `option` datatype is a standard type, it requires no special support, so we can treat it as a user-defined type. Options can be encoded easily using a sum:

```
fun option t =
    let
       open Show
    in
       inj (fn NONE => INL ()
             | SOME v => INR v)
           (data (C0"NONE" + C1"SOME" t))
    end

val "SOME 5" =
    let open Show in show (option int) end
       (SOME 5)
```

Readers new to type-indexed values might want to type annotate each subexpression of the above example as an exercise. (Use a compiler to check your annotations.)

Using a product, user specified records can be also be encoded easily:

```
val abc =
    let
       open Show
```

```
    in
       inj (fn {a, b, c} => a & b & c)
           (record (L"a" (option int) *
                    L"b" real *
                    L"c" bool))
    end

val "{a = SOME 1, b = 3.0, c = false}" =
    let open Show in show abc end
       {a = SOME 1, b = 3.0, c = false}
```

As you can see, both of the above use `inj` to inject user-defined types to the general purpose sum and product types.

Of particular interest is whether recursive datatypes and cyclic data can be handled. For example, how does one write a type-index for a recursive datatype such as a cyclic graph?

```
datatype 'a graph = VTX of 'a * 'a graph list ref
fun arcs (VTX (_, r)) = r
```

Using the `Show` combinators, we could first write a new type-index combinator for `graph`:

```
fun graph a =
    let
       open Fix Show
    in
       fix Y (fn graph_a =>
                 inj (fn VTX (x, y) => x & y)
                     (data (C1"VTX"
                               (tuple (U a *
                                       U (refc (list graph_a)))))))
    end
```

To show a graph with integer labels

```
val a_graph =
    let
       val a = VTX (1, ref [])
       val b = VTX (2, ref [])
       val c = VTX (3, ref [])
       val d = VTX (4, ref [])
       val e = VTX (5, ref [])
       val f = VTX (6, ref [])
    in
       arcs a := [b, d]
     ; arcs b := [c, e]
     ; arcs c := [a, f]
     ; arcs d := [f]
     ; arcs e := [d]
     ; arcs f := [e]
     ; a
    end
```

we could then simply write

```
val "VTX (1, ref [VTX (2, ref [VTX (3, ref [VTX (1, %0), \
    \VTX (6, ref [VTX (5, ref [VTX (4, ref [VTX (6, %3)])])] as %3)]), \
    \VTX (5, ref [VTX (4, ref [VTX (6, ref [VTX (5, %2)])])] as %2)]), \
```

```
    \VTX (4, ref [VTX (6, ref [VTX (5, ref [VTX (4, %1)])])] as %1)] as %0)" =
    let open Show in show (graph int) end
       a_graph
```

There is a subtle gotcha with cyclic data. Consider the following code:

```
exception ExnArray of exn array

val () =
    let
       open Show
    in
       regExn (fn ExnArray a =>
                  SOME (a, C1"ExnArray" (array exn))
                | _ => NONE)
    end

val a_cycle =
    let
       val a = Array.fromList [Empty]
    in
       Array.update (a, 0, ExnArray a) ; a
    end
```

Although the above looks innocent enough, the evaluation of

```
val "[|ExnArray %0|] as %0" =
    let open Show in show (array exn) end
       a_cycle
```

goes into an infinite loop. To avoid this problem, the type-index `array exn` must be evaluated only once, as in the following:

```
val array_exn = let open Show in array exn end

exception ExnArray of exn array

val () =
    let
       open Show
    in
       regExn (fn ExnArray a =>
                  SOME (a, C1"ExnArray" array_exn)
                | _ => NONE)
    end

val a_cycle =
    let
       val a = Array.fromList [Empty]
    in
       Array.update (a, 0, ExnArray a) ; a
    end

val "[|ExnArray %0|] as %0" =
    let open Show in show array_exn end
       a_cycle
```

Cyclic data (excluding closures) in Standard ML can only be constructed imperatively through arrays and references (combined with exceptions or recursive datatypes). Before recursing to a reference or an array, one needs to check whether that reference or array has already been seen before. When `ref` or `array` is called with a type-index, a new cyclicity checker is instantiated.

Implementation

```
structure SmlSyntax =
   struct
      local
         structure CV = CharVector and C = Char
      in
         val isSym = Char.contains "!%&$#+-/:<=>?@\\~`^|*"

         fun isSymId s = 0 < size s andalso CV.all isSym s

         fun isAlphaNumId s =
             0 < size s
             andalso C.isAlpha (CV.sub (s, 0))
             andalso CV.all (fn c => C.isAlphaNum c
                                orelse #"'" = c
                                orelse #"_" = c) s

         fun isNumLabel s =
             0 < size s
             andalso #"0" <> CV.sub (s, 0)
             andalso CV.all C.isDigit s

         fun isId s = isAlphaNumId s orelse isSymId s

         fun isLongId s = List.all isId (String.fields (#"." <\ op =) s)

         fun isLabel s = isId s orelse isNumLabel s
      end
   end

structure Show :> SHOW =
   struct
      datatype 'a t = IN of exn list * 'a -> bool * string
      type 'a s = 'a t
      type ('a, 'k) p = 'a t
      type u = unit
      type l = unit

      fun show (IN t) x = #2 (t ([], x))

      (* user-defined types *)
      fun inj inj (IN b) = IN (b o cross (id, inj))

      local
         fun surround pre suf (_, s) = (false, concat [pre, s, suf])
         fun parenthesize x = if #1 x then surround "(" ")" x else x
         fun construct tag =
             (fn (_, s) => (true, concat [tag, " ", s])) o parenthesize
         fun check p m s = if p s then () else raise Fail (m^s)
      in
        (* tuples and records *)
         fun (IN l) * (IN r) =
             IN (fn (rs, a & b) =>
                    (false, concat [#2 (l (rs, a)),
                                   ", ",
```

```
                                 #2 (r (rs, b))]))

val U = id
fun L l = (check SmlSyntax.isLabel "Invalid label: " l
         ; fn IN t => IN (surround (l^" = ") "" o t))

fun tuple (IN t) = IN (surround "(" ")" o t)
fun record (IN t) = IN (surround "{" "}" o t)

(* datatypes *)
fun (IN l) + (IN r) = IN (fn (rs, INL a) => l (rs, a)
                             | (rs, INR b) => r (rs, b))

fun C0 c = (check SmlSyntax.isId "Invalid constructor: " c
          ; IN (const (false, c)))
fun C1 c (IN t) = (check SmlSyntax.isId "Invalid constructor: " c
                 ; IN (construct c o t))

val data = id

fun Y ? = Fix.iso (IN, fn IN x => x) Fn.Y ?

(* exceptions *)
local
   val handlers = ref ([] : (exn -> unit t option) list)
in
   val exn = IN (fn (rs, e) =>
                   let
                      fun lp [] =
                          C0(concat ["<exn:",
                                     General.exnName e,
                                     ">"])
                        | lp (f::fs) =
                          case f e of
                             NONE => lp fs
                           | SOME t => t
                      val IN f = lp (!handlers)
                   in
                      f (rs, ())
                   end)
   fun regExn f =
       handlers := (Option.map
                       (fn (x, IN f) =>
                           IN (fn (rs, ()) =>
                                  f (rs, x))) o f)
                  :: !handlers
end

(* some built-in type constructors *)
local
   fun cyclic (IN t) =
       let
          exception E of ''a * bool ref
       in
          IN (fn (rs, v : ''a) =>
                let
                   val idx = Int.toString o length
                   fun lp (E (v', c)::rs) =
                       if v' <> v then lp rs
                       else (c := false ; (false, "%"^idx rs))
                     | lp (_::rs) = lp rs
```

```
                         | lp [] =
                           let
                              val c = ref true
                              val r = t (E (v, c)::rs, v)
                           in
                              if !c then
                                 r
                              else
                                 surround "" (" as %"^idx rs) r
                           end
                  in
                     lp rs
                  end)
         end

      fun aggregate pre suf toList (IN t) =
          IN (surround pre suf o
              (fn (rs, a) =>
                  (false,
                   String.concatWith
                      ", "
                      (map (#2 o curry t rs)
                           (toList a)))))

   in
      fun refc ? = (cyclic o inj ! o C1"ref") ?
      fun array ? = (cyclic o aggregate "[|" "|]" (Array.foldr op:: [])) ?
      fun list ? = aggregate "[" "]" id ?
      fun vector ? = aggregate "#[" "]" (Vector.foldr op:: []) ?
   end

   fun (IN _) --> (IN _) = IN (const (false, "<fn>"))

   (* some built-in base types *)
   local
      fun mk toS = (fn x => (false, x)) o toS o (fn (_, x) => x)
   in
      val string =
          IN (surround "\"" "\"" o mk (String.translate Char.toString))
      val unit = IN (mk (fn () => "()"))
      val bool = IN (mk Bool.toString)
      val char = IN (surround "#\"" "\"" o mk Char.toString)
      val int = IN (mk Int.toString)
      val word = IN (surround "0wx" "" o mk Word.toString)
      val real = IN (mk Real.toString)
   end
 end
end

(* Handlers for standard top-level exceptions *)
val () =
    let
       open Show
       fun E0 name = SOME ((), C0 name)
    in
       regExn (fn Bind => E0"Bind"
                | Chr => E0"Chr"
                | Div => E0"Div"
                | Domain => E0"Domain"
                | Empty => E0"Empty"
                | Match => E0"Match"
```

```
             | Option => E0"Option"
             | Overflow  => E0"Overflow"
             | Size => E0"Size"
             | Span => E0"Span"
             | Subscript => E0"Subscript"
             | _ => NONE)
 ; regExn (fn Fail s => SOME (s, C1"Fail" string)
             | _ => NONE)
end
```

Last edited on 2006-08-13 15:03:53 by VesaKarvonen.

TypeVariableScope

In Standard ML, every type variable is *scoped* (or bound) at a particular point in the program. A type variable can be either implicitly scoped or explicitly scoped. For example, `'a` is implicitly scoped in

```
val id: 'a -> 'a = fn x => x
```

and is implicitly scoped in

```
val id = fn x: 'a => x
```

On the other hand, `'a` is explicitly scoped in

```
val 'a id: 'a -> 'a = fn x => x
```

and is explicitly scoped in

```
val 'a id = fn x: 'a => x
```

A type variable can be scoped at a `val` or `fun` declaration. An SML type checker performs scope inference on each top-level declaration to determine the scope of each implicitly scoped type variable. After scope inference, every type variable is scoped at exactly one enclosing `val` or `fun` declaration. Scope inference shows that the first and second example above are equivalent to the third and fourth example, respectively.

Section 4.6 of the Definition specifies precisely the scope of an implicitly scoped type variable. A free occurrence of a type variable `'a` in a declaration `d` is said to be *unguarded* in `d` if `'a` is not part of a smaller declaration. A type variable `'a` is implicitly scoped at `d` if `'a` is unguarded in `d` and `'a` does not occur unguarded in any declaration containing `d`.

Scope inference examples

- In this example,

  ```
  val id: 'a -> 'a = fn x => x
  ```

  `'a` is unguarded in `val id` and does not occur unguarded in any containing declaration. Hence, `'a` is scoped at `val id` and the declaration is equivalent to the following.

  ```
  val 'a id: 'a -> 'a = fn x => x
  ```
- In this example,

  ```
  val f = fn x => let exception E of 'a in E x end
  ```

  `'a` is unguarded in `val f` and does not occur unguarded in any containing declaration. Hence, `'a` is scoped at `val f` and the declaration is equivalent to the following.

  ```
  val 'a f = fn x => let exception E of 'a in E x end
  ```
- In this example (taken from the Definition),

  ```
  val x: int -> int = let val id: 'a -> 'a = fn z => z in id id end
  ```

`'a` occurs unguarded in `val id`, but not in `val x`. Hence, `'a` is implicitly scoped at `val id`, and the declaration is equivalent to the following.

```
val x: int -> int = let val 'a id: 'a -> 'a = fn z => z in id id end
```

- In this example,

```
val f = (fn x: 'a => x) (fn y => y)
```

`'a` occurs unguarded in `val f` and does not occur unguarded in any containing declaration. Hence, `'a` is implicitly scoped at `val f`, and the declaration is equivalent to the following.

```
val 'a f = (fn x: 'a => x) (fn y => y)
```

This does not type check due to the ValueRestriction.

- In this example,

```
fun f x =
   let
      fun g (y: 'a) = if true then x else y
   in
      g x
   end
```

`'a` occurs unguarded in `fun g`, not in `fun f`. Hence, `'a` is implicitly scoped at `fun g`, and the declaration is equivalent to

```
fun f x =
   let
      fun 'a g (y: 'a) = if true then x else y
   in
      g x
   end
```

This fails to type check because `x` and `y` must have the same type, and hence `'a` can not be generalized at `fun g`. MLton reports

```
Error: scope.sml 3.7.
  Unable to generalize 'a.
    in: fun 'a g ((y): 'a) = (if true then x else y)
```

This problem could be fixed either by adding an explicit type constraint, as in `fun f (x: 'a)`, or by explicitly scoping `'a`, as in `fun 'a f x`.

Restrictions on type variable scope

It is not allowed to scope a type variable within a declaration in which it is already in scope (see the last restriction listed on page 9 of the Definition). For example, the following program is invalid.

```
fun 'a f (x: 'a) =
   let
      fun 'a g (y: 'a) = y
   in
      ()
   end
```

MLton reports

```
Error: z.sml 3.11.
  Type variable 'a scoped at an outer declaration.
```

This is an error even if the scoping is implicit. That is, the following program is invalid as well.

```
fun f (x: 'a) =
   let
      fun 'a g (y: 'a) = y
   in
      ()
   end
```

Last edited on 2005-12-02 03:01:09 by StephenWeeks.

Unicode

The current release of MLton does not support Unicode. We are working on adding support.

- `WideChar` structure.
- UTF-8 encoded source files.

There is no real support for Unicode in the Definition of Standard ML; there are only a few throw-away sentences along the lines of "ASCII must be a subset of the character set in programs".

Neither is there real support for Unicode in the Standard ML Basis Library. The general consensus (which includes the opinions of the editors of the Basis Library) is that the `WideChar` structure is insufficient for the purposes of Unicode. There is no `LargeChar` structure, which in itself is a deficiency, since a programmer can not program against the largest supported character size.

MLton has some preliminary support for 16 and 32 bit characters and strings. It is even possible to include arbitrary Unicode characters in 32-bit strings using a `\Uxxxxxxxx` escape sequence. (This longer escape sequence is a minor extension over the Definition which only allows `\uxxxx`.) This is by no means completely satisfactory in terms of support for Unicode, but it is what is currently available.

There are periodic flurries of questions and discussion about Unicode in MLton/SML. In December 2004, there was a discussion that led to some seemingly sound design decisions. The discussion started at:

http://mlton.org/pipermail/mlton/2004-December/026396.html

There is a good summary of points at:

http://mlton.org/pipermail/mlton/2004-December/026440.html

In November 2005, there was a followup discussion and the beginning of some coding.

http://mlton.org/pipermail/mlton/2005-November/028300.html

We are optimistic that support will appear in the next MLton release.

Also see

The fxp XML parser has some support for dealing with Unicode documents.

Last edited on 2005-12-02 04:26:33 by StephenWeeks.

UniversalType

A universal type is a type into which all other types can be embedded. Here's a Standard ML signature for a universal type.

```
signature UNIVERSAL_TYPE =
   sig
      type t

      val embed: unit -> ('a -> t) * (t -> 'a option)
   end
```

The idea is that `type t` is the universal type and that each call to `embed` returns a new pair of functions `(inject, project)`, where `inject` embeds a value into the universal type and `project` extracts the value from the universal type. A pair `(inject, project)` returned by `embed` works together in that `project u` will return `SOME v` if and only if `u` was created by `inject v`. If `u` was created by a different function `inject'`, then `project` returns `NONE`.

Here's an example embedding integers and reals into a universal type.

```
functor Test (U: UNIVERSAL_TYPE): sig end =
   struct
      val (intIn: int -> U.t, intOut) = U.embed ()
      val r: U.t ref = ref (intIn 13)
      val s1 =
         case intOut (!r) of
            NONE => "NONE"
          | SOME i => Int.toString i
      val (realIn: real -> U.t, realOut) = U.embed ()
      val () = r := realIn 13.0
      val s2 =
         case intOut (!r) of
            NONE => "NONE"
          | SOME i => Int.toString i
      val s3 =
         case realOut (!r) of
            NONE => "NONE"
          | SOME x => Real.toString x
      val () = print (concat [s1, " ", s2, " ", s3, "\n"])
   end
```

Applying `Test` to an appropriate implementation will print

```
13 NONE 13.0
```

Note that two different calls to embed on the same type return different embeddings.

Standard ML does not have explicit support for universal types; however, there are at least two ways to implement them.

Implementation Using Exceptions

While the intended use of SML exceptions is for exception handling, an accidental feature of their design is that the `exn` type is a universal type. The implementation relies on being able to declare exceptions locally to a function and on the fact that exceptions are generative.

```
structure U:> UNIVERSAL_TYPE =
   struct
      type t = exn

      fun 'a embed () =
         let
            exception E of 'a
            fun project (e: t): 'a option =
               case e of
                  E a => SOME a
                | _ => NONE
         in
            (E, project)
         end
   end
```

Implementation Using Functions and References

```
structure U:> UNIVERSAL_TYPE =
   struct
      datatype t = T of {clear: unit -> unit,
                         store: unit -> unit}

      fun 'a embed () =
         let
            val r: 'a option ref = ref NONE
            fun inject (a: 'a): t =
               T {clear = fn () => r := NONE,
                  store = fn () => r := SOME a}
            fun project (T {clear, store}): 'a option =
               let
                  val () = store ()
                  val res = !r
                  val () = clear ()
               in
                  res
               end
         in
            (inject, project)
         end
   end
```

Note that due to the use of a shared ref cell, the above implementation is not thread safe.

One could try to simplify the above implementation by eliminating the `clear` function, making `type t = unit -> unit`.

```
structure U:> UNIVERSAL_TYPE =
   struct
      type t = unit -> unit

      fun 'a embed () =
         let
            val r: 'a option ref = ref NONE
            fun inject (a: 'a): t = fn () => r := SOME a
            fun project (f: t): 'a option = (r := NONE; f (); !r)
```

```
      in
         (inject, project)
      end
end
```

While correct, this approach keeps the contents of the ref cell alive longer than necessary, which could cause a space leak. The problem is in `project`, where the call to `f` stores some value in some ref cell `r'`. Perhaps `r'` is the same ref cell as `r`, but perhaps not. If we do not clear `r'` before returning from `project`, then `r'` will keep the value alive, even though it is useless.

Also see

* PropertyList: Lisp-style property lists implemented with a universal type.

Last edited on 2005-05-29 03:04:34 by VesaKarvonen.

UnresolvedBugs

Here are the places where MLton deviates from the Definition of SML. In general, MLton complies with the Definition quite closely, typically much more closely than other SML compilers (see, e.g., our list of SML/NJ's deviations). In fact, the three deviations listed here are the only known deviations, and we have no plans to fix them. If you find a deviation not listed here, please report a Bug.

We don't plan to fix these bugs because one of them (parsing nested cases) has historically never been accepted by any SML compiler, while the other two clearly indicate problems in the Definition.

- MLton does not correctly parse case expressions nested within other matches. For example, the following fails.

  ```
  fun f 0 y =
        case x of
           1 => 2
         | _ => 3
    | f _ y = 4
  ```

  To do this in a program, simply parenthesize the case expression.

  Allowing such expressions, although compliant with the Definition, would be a mistake, since using parentheses is clearer and no SML compiler has ever allowed them. Furthermore, implementing this would require serious yacc grammar rewriting followed by postprocessing.
- MLton rejects rebinding of constructors with `val rec` declarations, as in

  ```
  val rec NONE = fn () => ()
  ```

  The Definition (bizarrely) requires this program to type check, but to raise `Bind`.

  We have no plans to change this behavior, as the Definition's behavior is clearly an error, a mismatch between the static semantics and the dynamic semantics.
- MLton does not hide the equality aspect of types declared in abstype declarations. So, MLton accepts programs like the following, while the Definition rejects them.

  ```
  abstype t = T with end
  val _ = fn (t1, t2 : t) => t1 = t2

  abstype t = T with val a = T end
  val _ = a = a
  ```

  One consequence of this choice is that MLton accepts the following program, in accordance with the Definition.

  ```
  abstype t = T with val eq = op = end
  val _ = fn (t1, t2 : t) => eq (t1, t2)
  ```

  Other implementations will typically reject this program, because they make an early choice for the type of `eq` to be `''a * ''a -> bool` instead of `t * t -> bool`. The choice is understandable, since the Definition accepts the following program.

  ```
  abstype t = T with val eq = op = end
  val _ = eq (1, 2)
  ```

Last edited on 2005-12-02 03:06:59 by StephenWeeks.

UnsafeStructure

This module is a subset of the `Unsafe` module provided by SML/NJ.

```
signature UNSAFE_MONO_ARRAY =
   sig
      type array
      type elem

      val create: int -> array
      val sub: array * int -> elem
      val update: array * int * elem -> unit
   end

signature UNSAFE_MONO_VECTOR =
   sig
      type elem
      type vector

      val sub: vector * int -> elem
   end

signature UNSAFE =
   sig
      structure Array:
         sig
            val create: int * 'a -> 'a array
            val sub: 'a array * int -> 'a
            val update: 'a array * int * 'a -> unit
         end
      structure CharArray: UNSAFE_MONO_ARRAY
      structure CharVector: UNSAFE_MONO_VECTOR
      structure Real64Array: UNSAFE_MONO_ARRAY
      structure Vector:
         sig
            val sub: 'a vector * int -> 'a
         end
      structure Word8Array: UNSAFE_MONO_ARRAY
      structure Word8Vector: UNSAFE_MONO_VECTOR
   end
```

Last edited on 2005-01-26 20:29:31 by MatthewFluet.

Useless

Useless is an optimization pass for the SSA IntermediateLanguage, invoked from SSASimplify.

Description

This pass:

- removes components of tuples that are constants (use unification)
- removes function arguments that are constants
- builds some kind of dependence graph where

- a value of ground type is useful if it is an arg to a primitive - a tuple is useful if it contains a useful component - a constructor is useful if it contains a useful component or is used in a `Case` transfer

If a useful tuple is coerced to another useful tuple, then all of their components must agree (exactly). It is trivial to convert a useful value to a useless one.

Implementation

useless.sig useless.fun

Details and Notes

It is also trivial to convert a useful tuple to one of its useful components -- but this seems hard.

Suppose that you have a `ref`/`array`/`vector` that is useful, but the components aren't -- then the components are converted to type `unit`, and any primitive args must be as well.

Unify all handler arguments so that `raise`/`handle` has a consistent calling convention.

Last edited on 2005-12-02 03:09:00 by StephenWeeks.

Users

Here is a list of companies, projects, and courses that use or have used MLton. If you use MLton and are not here, please add your project with a brief description and a link. Thanks.

Companies

- Hardcore Processing uses MLton as a crosscompiler from Linux to Windows for graphics and game software.
    - CEX3D Converter, a conversion program for 3D objects.
    - Interactive Showreel, which contains a crossplatform GUI-toolkit and a realtime renderer for a subset of RenderMan written in Standard ML.
    - various games
- PolySpace Technologies builds their product that detects runtime errors in embedded systems based on abstract interpretation.
- Sourcelight Technologies uses MLton internally for prototyping and for processing databases as part of their system that makes personalized movie recommendations.

Projects

- ADATE, Automatic Design of Algorithms Through Evolution, a system for automatic programming i.e., inductive inference of algorithms. ADATE can automatically generate non-trivial and novel algorithms written in Standard ML.
- CIL, a compiler for SML based on intersection and union types.
- ConCert, a project investigating certified code for grid computing.
- Cooperative Internet hosting tools
- Guugelhupf, a simple search engine.
- HaMLet a model implementation of Standard ML.
- Metis, a first-order prover used in the HOL4 theorem proving system.
- mlftpd, an ftp daemon written in SML. TomMurphy is also working on replacements for standard network services in SML. He also uses MLton to build his entries (2001, 2002, 2004, 2005) in the annual ICFP programming contest.
- MLOPE, an offline partial evaluator for Standard ML.
- RML, a system for developing, compiling and debugging and teaching structural operational semantics (SOS) and natural semantics specifications.
- SMLNJtrans, a program for generating SML/NJ transcripts in LaTeX.
- SSA PRE, an implementation of Partial Redundancy Elimination for MLton.
- STING, self-adjusting computation, a paradigm of computing where programs can automatically adjust to changes to their data.
- Tina (Time Petri net Analyzer)
- Twelf an implementation of the LF logical framework.

Courses

- Harvard CS-152, undergraduate programming languages.
- HÃ¸gskolen i Ã stfold IAI30202, programming languages.

Last edited on 2006-09-01 17:29:17 by MatthewFluet.

Utilities

This page is a collection of basic utilities used in the examples on various pages. See

- InfixingOperators, and
- ProductType

for longer discussions on some of these utilities.

```
(* Operator precedence table *)
infix   8  * / div mod        (* +1 from Basis Library *)
infix   7  + - ^              (* +1 from Basis Library *)
infixr  6  :: @               (* +1 from Basis Library *)
infix   5  = <> > >= < <=     (* +1 from Basis Library *)
infix   4  <\ \>
infixr  4  </ />
infix   3  o
infix   2  >|
infixr  2  |<
infix   1  :=                 (* -2 from Basis Library *)
infix   0  before &

(* Some basic combinators *)
fun const x _ = x
fun cross (f, g) (x, y) = (f x, g y)
fun curry f x y = f (x, y)
fun fail e _ = raise e
fun id x = x

(* Product type *)
datatype ('a, 'b) product = & of 'a * 'b

(* Sum type *)
datatype ('a, 'b) sum = INL of 'a | INR of 'b

(* Some type shorthands *)
type 'a uop = 'a -> 'a
type 'a fix = 'a uop -> 'a
type 'a thunk = unit -> 'a
type 'a effect = 'a -> unit
type ('a, 'b) emb = ('a -> 'b) * ('b -> 'a)

(* Infixing, sectioning, and application operators *)
fun x <\ f = fn y => f (x, y)
fun f \> y = f y
fun f /> y = fn x => f (x, y)
fun x </ f = f x

(* Piping operators *)
val op>| = op</
val op|< = op\>
```

Last edited on 2006-08-13 14:40:14 by VesaKarvonen.

ValueRestriction

The value restriction is a rule that governs when type inference is allowed to polymorphically generalize a value declaration. In short, the value restriction says that generalization can only occur if the right-hand side of an expression is syntactically a value. For example, in

```
val f = fn x => x
val _ = (f "foo"; f 13)
```

the expression `fn x => x` is syntactically a value, so `f` has polymorphic type `'a -> 'a` and both calls to `f` type check. On the other hand, in

```
val f = let in fn x => x end
val _ = (f "foo"; f 13)
```

the expression `let in fn x => end end` is not syntactically a value and so `f` can either have type `int -> int` or `string -> string`, but not `'a -> 'a`. Hence, the program does not type check.

The Definition of SML spells out precisely which expressions are syntactic values (it refers to such expressions as *non-expansive*). An expression is a value if it is of one of the following forms.

- a constant (`13`, `"foo"`, `13.0`, ...)
- a variable (`x`, `y`, ...)
- a function (`fn x => e`)
- the application of a constructor other than `ref` to a value (`Foo v`)
- a type constrained value (`v: t`)
- a tuple in which each field is a value `(v1, v2, ...)`
- a record in which each field is a value `{l1 = v1, l2 = v2, ...}`
- a list in which each element is a value `[v1, v2, ...]`

Why the value restriction exists

The value restriction prevents a ref cell (or an array) from holding values of different types, which would allow a value of one type to be cast to another and hence would break type safety. If the restriction were not in place, the following program would type check.

```
val r: 'a option ref = ref NONE
val r1: string option ref = r
val r2: int option ref = r
val () = r1 := SOME "foo"
val v: int = valOf (!r2)
```

The first line violates the value restriction because `ref NONE` is not a value. All other lines are type correct. By its last line, the program has cast the string `"foo"` to an integer. This breaks type safety, because now we can add a string to an integer with an expression like `v + 13`. We could even be more devious, by adding the following two lines, which allow us to threat the string `"foo"` as a function.

```
val r3: (int -> int) option ref = r
val v: int -> int = valOf (!r3)
```

Eliminating the explicit `ref` does nothing to fix the problem. For example, we could replace the declaration of `r` with the following.

```
val f: unit -> 'a option ref = fn () => ref NONE
val r: 'a option ref = f ()
```

The declaration of `f` is well typed, while the declaration of `r` violates the value restriction because `f ()` is not a value.

Unnecessarily rejected programs

Unfortunately, the value restriction rejects some programs that could be accepted.

```
val id: 'a -> 'a = fn x => x
val f: 'a -> 'a = id id
```

The type constraint on `f` requires `f` to be polymorphic, which is disallowed because `id id` is not a value. MLton reports the following type error.

```
Error: z.sml 2.19.
  Can't bind type variable: 'a.
    in: val 'a (f): ('a -> 'a) = id id
```

MLton indicates the inability to make `f` polymorphic by saying that it can't bind the type variable `'a` at the declaration. MLton doesn't explicitly mention the value restriction, but that is the reason. If we leave the type constraint off of `f`

```
val id: 'a -> 'a = fn x => x
val f = id id
```

then the program succeeds; however, MLton gives us the following warning.

```
Warning: z.sml 2.1.
  Unable to locally determine type of variable: f.
    type: ??? -> ???
    in: val f = id id
```

This warning indicates that MLton couldn't polymorphically generalize `f`, nor was there enough context using `f` to determine its type. This in itself is not a type error, but it it is a hint that something is wrong with our program. Using `f` provides enough context to eliminate the warning.

```
val id: 'a -> 'a = fn x => x
val f = id id
val _ = f 13
```

But attempting to use `f` as a polymorphic function will fail.

```
val id: 'a -> 'a = fn x => x
val f = id id
val _ = f 13
val _ = f "foo"
```

Alternatives to the value restriction

There would be nothing wrong with treating `f` as polymorphic in

```
val id: 'a -> 'a = fn x => x
val f = id id
```

One might think that the value restriction could be relaxed, and that only types involving `ref` should be disallowed. Unfortunately, the following example shows that even the type `'a -> 'a` can cause problems. If this program were allowed, then we could cast an integer to a string (or any other type).

```
val f: 'a -> 'a =
   let
      val r: 'a option ref = ref NONE
   in
      fn x =>
      let
         val y = !r
         val () = r := SOME x
      in
         case y of
            NONE => x
          | SOME y => y
      end
   end
val _ = f 13
val _ = f "foo"
```

The previous version of Standard ML took a different approach (MilnerEtAl90, Tofte90, ImperativeTypeVariable) than the value restriction. It encoded information in the type system about when ref cells would be created, and used this to prevent a ref cell from holding multiple types. Although it allowed more programs to be type checked, this approach had significant drawbacks. First, it was significantly more complex, both for implementors and for programmers. Second, it had an unfortunate interaction with the modularity, because information about ref usage was exposed in module signatures. This either prevented the use of references for implementing a signature, or required information that one would like to keep hidden to propagate across modules.

In the early nineties, Andrew Wright studied about 250,000 lines of existing SML code and discovered that it did not make significant use of the extended typing ability, and proposed the value restriction as a simpler alternative (Wright95). This was adopted in the revised Definition of SML.

Working with the value restriction

One technique that works with the value restriction is EtaExpansion. We can use eta expansion to make our `id id` example type check follows.

```
val id: 'a -> 'a = fn x => x
val f: 'a -> 'a = fn z => (id id) z
```

This solution means that the computation (in this case `id id`) will be performed each time `f` is applied, instead of just once when `f` is declared. In this case, that is not a problem, but it could be if the declaration of `f` performs substantial computation or creates a shared data structure.

Another technique that sometimes works is to move a monomorphic computation prior to a (would-be) polymorphic declaration so that the expression is a value. Consider the following program, which fails due to the value restriction.

```
datatype 'a t = A of string | B of 'a
val x: 'a t = A (if true then "yes" else "no")
```

It is easy to rewrite this program as

```
datatype 'a t = A of string | B of 'a
local
   val s = if true then "yes" else "no"
in
   val x: 'a t = A s
end
```

The following example (taken from Wright95) creates a ref cell to count the number of times a function is called.

```
val count: ('a -> 'a) -> ('a -> 'a) * (unit -> int) =
   fn f =>
   let
      val r = ref 0
   in
      (fn x => (r := 1 + !r; f x), fn () => !r)
   end
val id: 'a -> 'a = fn x => x
val (countId: 'a -> 'a, numCalls) = count id
```

The example does not type check, due to the value restriction. However, it is easy to rewrite the program, staging the ref cell creation before the polymorphic code.

```
datatype t = T of int ref
val count1: unit -> t = fn () => T (ref 0)
val count2: t * ('a -> 'a) -> (unit -> int) * ('a -> 'a) =
   fn (T r, f) => (fn () => !r, fn x => (r := 1 + !r; f x))
val id: 'a -> 'a = fn x => x
val t = count1 ()
val countId: 'a -> 'a = fn z => #2 (count2 (t, id)) z
val numCalls = #1 (count2 (t, id))
```

Of course, one can hide the constructor `T` inside a `local` or behind a signature.

Also see

- ImperativeTypeVariable

Last edited on 2006-03-28 00:58:56 by StephenWeeks.

VariableArityPolymorphism

Standard ML programmers often face the problem of how to provide a variable-arity polymorphic function. For example, suppose one is defining a combinator library, e.g. for parsing or pickling. The signature for such a library might look something like the following.

```
signature COMBINATOR =
   sig
      type 'a t

      val int: int t
      val real: real t
      val string: string t
      val unit: unit t
      val tuple2: 'a1 t * 'a2 t -> ('a1 * 'a2) t
      val tuple3: 'a1 t * 'a2 t * 'a3 t -> ('a1 * 'a2 * 'a3) t
      val tuple4: 'a1 t * 'a2 t * 'a3 t * 'a4 t
                  -> ('a1 * 'a2 * 'a3 * 'a4) t
      ...
   end
```

The question is how to define a variable-arity tuple combinator. Traditionally, the only way to take a variable number of arguments in SML is to put the arguments in a list (or vector) and pass that. So, one might define a tuple combinator with the following signature.

```
val tupleN: 'a list -> 'a list t
```

The problem with this approach is that as soon as one places values in a list, they must all have the same type. So, programmers often take an alternative approach, and define a family of `tuple<N>` functions, as we see in the `COMBINATOR` signature above.

The family-of-functions approach is ugly for many reasons. First, it clutters the signature with a number of functions when there should really only be one. Second, it is *closed*, in that there are a fixed number of tuple combinators in the interface, and should a client need a combinator for a large tuple, he is out of luck. Third, this approach often requires a lot of duplicate code in the implementation of the combinators.

Fortunately, using Fold01N and products, one can provide an interface and implementation that solves all these problems. Here is a simple pickling module that converts values to strings.

```
structure Pickler =
   struct
      type 'a t = 'a -> string

      val unit = fn () => ""

      val int = Int.toString

      val real = Real.toString

      val string = id

      type 'a accum = 'a * string list -> string list

      val tuple =
         fn z =>
```

```
         Fold01N.fold
         {finish = fn ps => fn x => concat (rev (ps (x, []))),
          start = fn p => fn (x, l) => p x :: l,
          zero = unit}
         z

      val ` =
         fn z =>
         Fold01N.step1
         {combine = (fn (p, p') => fn (x & x', l) => p' x' :: "," :: p (x, l))}
         z
   end
```

If one has `n` picklers of types

```
val p1: a1 Pickler.t
val p2: a2 Pickler.t
...
val pn: an Pickler.t
```

then one can construct a pickler for n-ary products as follows.

```
tuple `p1 `p2 ... `pn $ : (a1 & a2 & ... & an) Pickler.t
```

For example, with `Pickler` in scope, one can prove the following equations.

```
"" = tuple $ ()
"1" = tuple `int $ 1
"1,2.0" = tuple `int `real $ (1 & 2.0)
"1,2.0,three" = tuple `int `real `string $ (1 & 2.0 & "three")
```

Here is the signature for `Pickler`. It shows why the `accum` type is useful.

```
signature PICKLER =
   sig
      type 'a t

      val int: int t
      val real: real t
      val string: string t
      val unit: unit t

      type 'a accum
      val ` : ('a accum, 'b t, ('a, 'b) prod accum,
               'z1, 'z2, 'z3, 'z4, 'z5, 'z6, 'z7) Fold01N.step1
      val tuple: ('a t, 'a accum, 'b accum, 'b t, unit t,
                  'z1, 'z2, 'z3, 'z4, 'z5) Fold01N.t
   end

structure Pickler: PICKLER = Pickler
```

Last edited on 2006-03-21 22:06:02 by StephenWeeks.

Variant

A *variant* is an arm of a datatype declaration. For example, the datatype

```
datatype t = A | B of int | C of real
```

has three variants: `A`, `B`, and `C`.

Last edited on 2005-12-02 03:13:02 by StephenWeeks.

VesaKarvonen

Vesa Karvonen is a student at the University of Helsinki. His interests lie in the design and implementation of programming languages.

Things he'd like to see for SML and hopes to be able to contribute towards:

- A practical tool for documenting libraries. Preferably one that is based on extracting the documentation from source code comments.
- A good IDE. Possibly an enhanced SML mode (`esml-mode`) for Emacs. Google for SLIME video to get an idea of what he'd like to see. Some specific notes:
    - show type at point
    - robust, consistent indentation
    - show documentation
    - jump to definition
- Documented and cataloged libraries. Perhaps something like Boost, but for SML libraries.

Last edited on 2005-08-12 13:52:37 by VesaKarvonen.

WantedPages

Pages that don't exist and the pages that link to them. Please help fill these in. Also see OrphanedPages.

1. CCodegen: Chunkify
2. CVS: Sources, Subversion
3. Codegen: Machine
4. Defunctionalization: ClosureConvert
5. FirstOrder: IntermediateLanguage, SSA, SSA2
6. FlatLattice: CommonArg
7. HigherOrder: IntermediateLanguage
8. LambdaLift: SXMLSimplify
9. LookupConstants: Defunctorize
10. MLDoc: Libraries
11. MLLex: Documentation, Features, FrontEnd, Installation, Libraries
12. MLNLFFIGEN: Features
13. MLRISC: PropertyList
14. MLYacc: Documentation, Features, FrontEnd, Installation, Libraries, MLBasisAvailableLibraries
15. McCorkle: Credits
16. PackWord: RayRacine
17. SimplyTyped: IntermediateLanguage, SSA, SSA2
18. TypeError: Bugs20051202
19. TypeInference: FirstClassPolymorphism
20. Uncurry: SXMLSimplify
21. Untyped: Machine
22. VilleLaurikari: TrustedGroup
23. ZZA: CompilerPassTemplate
24. ZZB: CompilerPassTemplate
25. ZZZ: CompilerPassTemplate
26. ZZZNext: TalkTemplate
27. ZZZOtherPass: CompilerPassTemplate
28. ZZZPrev: TalkTemplate
29. ZZZSimplify: CompilerPassTemplate

Last edited on 2004-11-09 02:12:23 by StephenWeeks.

WebSite

This web site is a Wiki and is implemented using MoinMoin. If you're new to Wikis or to MoinMoin, they have a lot of help pages. We have customized the look and feel, so some of their descriptions may not apply.

Next Steps

- AccessControl. Who can edit what.
- CreatingPages.
- EditingPages.
- SystemInfo. What version of MoinMoin we use, plus more.
- WikiMacros. Special macros for this site.
- WikiTool. Edit pages with your favorite text editor.

Site Maintenance

- OrphanedPages. Pages that no other page links to. Please help by linking to these.
- WantedPages. Pages that don't exist and the pages that link to them. Please help fill these in.
- OldPages. Pages with the oldest modification times.
- PageSize. Pages sorted in decreasing order of size.
- RecentChanges. Pages that have been changed recently.

Navigation

The box in the upper-right corner is to Google search the entire web site. Also in the upper right is a link to an Index of all pages, sorted by page title.

You can also do a search of just the wiki.

Wiki full-text search
Display context of search results
Case-sensitive searching

Wiki title search

Last edited on 2004-12-03 00:40:23 by StephenWeeks.

WesleyTerpstra

Wesley W. Terpstra is a PhD student at the Technische Universität Darmstadt (Germany).

Research interests

- Distributed systems (P2P)
- Number theory (Error-correcting codes)

My interest in SML is centered on the fact the the language is able to directly express ideas from number theory which are important for my work. Modules and Functors seem to be a very natural basis for implementing many algebraic structures. MLton provides an ideal platform for actual implementation as it is fast and has unboxed words.

Things I would like from MLton in the future:

- Some better optimization of mathematical expressions
- IPv6 and multicast support
- A complete GUI toolkit like mGTK
- More supported platforms so that applications written under MLton have a wider audience

---

Last edited on 2004-12-19 03:55:34 by WesleyTerpstra.

WholeProgramOptimization

Whole-program optimization is a compilation technique in which optimizations operate over the entire program. This allows the compiler many optimization opportunities that are not available when analyzing modules separately (as with separate compilation).

Most of MLton's optimizations are whole-program optimizations. Because MLton compiles the whole program at once, it can perform optimization across module boundaries. As a consequence, MLton often reduces or eliminates the run-time penalty that arises with separate compilation of SML features such as functors, modules, polymorphism, and higher-order functions. MLton takes advantage of having the entire program to perform transformations such as: defunctorization, monomorphisation, higher-order control-flow analysis, inlining, unboxing, argument flattening, redundant-argument removal, constant folding, and representation selection. Whole-program compilation is an integral part of the design of MLton and is not likely to change.

Last edited on 2004-12-06 06:01:10 by StephenWeeks.

WikiMacros

Here are the wiki macros available in addition to the usual MoinMoin ones.

- `[[Cite(anchor, text)]]` displays text as a link to the corresponding reference on the References page.
  Examples: a paper
- `[[DownloadSVN(pathToFile)]]` displays a download link to the ViewCVS page for `pathToFile`.
  Examples: Makefile , main.fun
- `[[IncludeSVN(pathToFile, type)]]` textually includes the latest contents of `pathToFile`, formatted with Enscript as `type` (as in the `!#syntax` processor). If `type` is omitted, use the extension of `pathToFile`.
  Example:

```
(* Copyright (C) 1999-2005 Henry Cejtin, Matthew Fluet, Suresh
 *    Jagannathan, and Stephen Weeks.
 * Copyright (C) 1997-2000 NEC Research Institute.
 *
 * MLton is released under a BSD-style license.
 * See the file MLton-LICENSE for details.
 *)

structure Main = Main ()

val _ =
   let
      open Trace.Immediate
   in
      debug := Out Out.error
      ; flagged ()
      ; on []
   end
```

- `[[ViewSVN(pathToFile)]]` displays a link to the ViewCVS page for `pathToFile`.
  Examples: Makefile , main.fun
- `[[ViewSVNSDir(pathToDir)]]` displays a link to the ViewCVS page for `pathToDir`.
  Examples: main

Last edited on 2005-08-10 12:43:15 by MatthewFluet.

WikiName

A WikiName is a word that uses capitalized words. WikiNames automatically become hyperlinks to the WikiName's page.

Last edited on 2005-12-02 03:20:19 by StephenWeeks.

WikiTool

We have written a simple command-line tool that makes it possible to edit wiki pages using your favorite text editor instead of within a browser text box. The tool provides a CVS/SVN-like command-line interface that can be used to update local copies of files from the web and to commit local modifications to the web.

The tool is written in SML (of course) and is [http://mlton.org/cgi-bin/viewcvs.cgi/mlton/wiki/ available in the MLton CVS]. To compile it, you need to have the latest SVN of the MLton library sources, and point the MLB path variable `MLTON_SRC_LIB` at the `lib/mlton` dir in the sources.

Here's a quick tutorial on how to use the tool

1. Create a new directory for your local copy of the wiki files.
2. In that directory, login.

   ```
   wiki login http://mlton.org StephenWeeks <my password>
   ```
3. Checkout (the raw wiki markup) files with commands like:

   ```
   wiki checkout Home
   wiki checkout Index Documentation
   ```
4. Edit the files using your favorite text editor.
5. Commit your changes with a command like

   ```
   wiki commit UserGuide
   ```
6. Logout.

   ```
   wiki logout
   ```

That's it for the simple use. There are also other commands like cvs.

- Download the new version of a file from the web if there is one.

  ```
  wiki update UserGuide
  ```
- Schedule a new file to be added (must be later committed, just like CVS) .

  ```
  wiki add NewFile
  ```
- Rename a page

  ```
  wiki rename OldFile NewFile
  ```
- Remove a page

  ```
  wiki remove DeletedFile
  ```
- Attach files to a page

  ```
  wiki attach <file> <attachment>
  ```
- Detach files to a page

  ```
  wiki detach <file> <attachment>
  ```

`rename` and `remove` shouldn't work for most people on `mlton.org` because of the way our AccessControl is set up.

This code is a two-day hack and is not widely used. But we've found it useful. Please send bug reports to MLton@mlton.org.

Last edited on 2005-12-02 03:21:22 by StephenWeeks.

XML

XML is an IntermediateLanguage, translated from CoreML by Defunctorize, optimized by XMLSimplify, and translated by Monomorphise to SXML.

Description

XML is polymorphic, higher-order, with flat patterns. Every XML expression is annotated with its type. Polymorphic generalization is made explicit through type variables annotating `val` and `fun` declarations. Polymorphic instantiation is made explicit by specifying type arguments at variable references. XML patterns can not be nested and can not contain wildcards, constraints, flexible records, or layering.

Implementation

xml.sig xml.fun
xml-tree.sig xml-tree.fun

Type Checking

XML also has a type checker, used for debugging. At present, the type checker is also the best specification of the type system of XML. If you need more details, the type checker (type-check.sig , type-check.fun), is pretty short.

Since the type checker does not affect the output of the compiler (unless it reports an error), it can be turned off. The type checker recursively descends the program, checking that the type annotating each node is the same as the type synthesized from the types of the expressions subnodes.

Details and Notes

XML uses the same atoms as Core ML, hence all identifiers (constructors, variables, etc.) are unique and can have properties attached to them. Finally, XML has a simplifier (XMLShrink), which implements a reduction system.

Types

XML types are either type variables or applications of n-ary type constructors. There are many utility functions for constructing and destructing types involving built-in type constructors.

A type scheme binds list of type variables in a type. The only interesting operation on type schemes is the application of a type scheme to a list of types, which performs a simultaneous substitution of the type arguments for the bound type variables of the scheme. For the purposes of type checking, it is necessary to know the type scheme of variables, constructors, and primitives. This is done by associating the scheme with the identifier using its property list. This approach is used instead of the more traditional environment approach for reasons of speed.

XmlTree

Before defining `XML`, the signature for language XML, we need to define an auxiliary signature `XML_TREE`, that contains the datatype declarations for the expression trees of XML. This is done solely for the purpose of modularity -- it allows the simplifier and type checker to be defined by separate functors (which take a structure matching `XML_TREE`). Then, `Xml` is defined as the signature for a module containing the expression trees, the simplifier, and the type checker.

Both constructors and variables can have type schemes, hence both constructor and variable references specify the instance of the scheme at the point of references. An instance is specified with a vector of types, which corresponds to the type variables in the scheme.

XML patterns are flat (i.e. not nested). A pattern is a constructor with an optional argument variable. Patterns only occur in `case` expressions. To evaluate a case expression, compare the test value sequentially against each pattern. For the first pattern that matches, destruct the value if necessary to bind the pattern variables and evaluate the corresponding expression. If no pattern matches, evaluate the default. All patterns of a case statement are of the same variant of `Pat.t`, although this is not enforced by ML's type system. The type checker, however, does enforce this. Because tuple patterns are irrefutable, there will only ever be one tuple pattern in a case expression and there will be no default.

XML contains value, exception, and mutually recursive function declarations. There are no free type variables in XML. All type variables are explicitly bound at either a value or function declaration. At some point in the future, exception declarations may go away, and exceptions may be represented with a single datatype containing a `unit ref` component to implement genericity.

XML expressions are like those of CoreML, with the following exceptions. There are no records expressions. After type inference, all records (some of which may have originally been tuples in the source) are converted to tuples, because once flexible record patterns have been resolved, tuple labels are superfluous. Tuple components are ordered based on the field ordering relation. XML eta expands primitives and constructors so that there are always fully applied. Hence, the only kind of value of arrow type is a lambda. This property is useful for flow analysis and later in code generation.

An XML program is a list of toplevel datatype declarations and a body expression. Because datatype declarations are not generative, the defunctorizer can safely move them to toplevel.

Last edited on 2005-12-02 04:26:42 by StephenWeeks.

XMLShrink

XMLShrink is an optimization pass for the XML IntermediateLanguage, invoked from XMLSimplify.

Description

This pass performs optimizations based on a reduction system.

Implementation

shrink.sig shrink.fun

Details and Notes

The simplifier is based on Shrinking Lambda Expressions in Linear Time.

The source program may contain functions that are only called once, or not even called at all. Match compilation introduces many such functions. In order to reduce the program size, speed up later phases, and improve the flow analysis, a source to source simplifier is run on XML after type inference and match compilation.

The simplifier implements the reductions shown below. The reductions eliminate unnecessary declarations (see the side constraint in the figure), applications where the function is immediate, and case statements where the test is immediate. Declarations can be eliminated only when the expression is nonexpansive (see Section 4.7 of the Definition), which is a syntactic condition that ensures that the expression has no effects (assignments, raises, or nontermination). The reductions on case statements do not show the other irrelevant cases that may exist. The reductions were chosen so that they were strongly normalizing and so that they never increased tree size.

-
  ```
  let x = e1 in e2
  ```
  reduces to
  ```
  e2 [x -> e1]
  ```
  if `e1` is a constant or variable or if `e1` is nonexpansive and `x` occurs zero or one time in `e2`
-
  ```
  (fn x => e1) e2
  ```
  reduces to
  ```
  let x = e2 in e1
  ```
-
  ```
  e1 handle e2
  ```
  reduces to
  ```
  e1
  ```

if `e1` is nonexpansive

- ```
  case let d in e end of p1 => e1 ...
  ```

  reduces to

  ```
  let d in case e of p1 => e1 ... end
  ```
- ```
  case C e1 of C x => e2
  ```

  reduces to

  ```
  let x = e1 in e2
  ```

Last edited on 2005-12-02 03:22:57 by StephenWeeks.

XMLSimplify

The optimization passes for the XML IntermediateLanguage are collected and controlled by the `XmlSimplify` functor (xml-simplify.sig , xml-simplify.fun).

The following optimization passes are implemented:

- XMLSimplifyTypes
- XMLShrink

The optimization passes can be controlled from the command-line by the options

- `diag-pass <pass>` -- keep diagnostic info for pass
- `drop-pass <pass>` -- omit optimization pass
- `keep-pass <pass>` -- keep the results of pass
- `xml-passes <passes>` -- xml optimization passes

Last edited on 2005-08-19 15:22:55 by MatthewFluet.

XMLSimplifyTypes

XMLSimplifyTypes is an optimization pass for the XML IntermediateLanguage, invoked from XMLSimplify.

Description

This pass simplifies types in an XML program, eliminating all unused type arguments.

Implementation

simplify-types.sig simplify-types.fun

Details and Notes

It first computes a simple fixpoint on all the `datatype` declarations to determine which `datatype tycon` args are actually used. Then it does a single pass over the program to determine which polymorphic declaration type variables are used, and rewrites types to eliminate unused type arguments.

This pass should eliminate any spurious duplication that the Monomorphise pass might perform due to phantom types.

Last edited on 2005-12-02 03:24:10 by StephenWeeks.

ZZZOrphanedPages

The contents of these pages have been moved to other pages.

These templates are used by other pages.

- CompilerPassTemplate
- TalkTemplate

Last edited on 2005-12-02 13:11:48 by MatthewFluet.

Zone

Zone is an optimization pass for the SSA2 IntermediateLanguage, invoked from SSA2Simplify.

Description

This pass breaks large SSA2 functions into zones, which are connected subgraphs of the dominator tree. For each zone, at the node that dominates the zone (the "zone root"), it places a tuple collecting all of the live variables at that node. It replaces any variables used in that zone with offsets from the tuple. The goal is to decrease the liveness information in large SSA functions.

Implementation

zone.sig zone.fun

Details and Notes

Compute strongly-connected components to avoid put tuple constructions in loops.

There are two (expert) flags that govern the use of this pass

```
-max-function-size <n>
-zone-cut-depth <n>
```

Zone splitting only works when the number of basic blocks in a function is > `n`. The `n` used to cut the dominator tree is set by `-zone-cut-depth`.

There is currently no attempt to be safe-for-space. That is, the tuples are not restricted to containing only "small" values.

In the `HOL` program, the particular problem is the main function, which has 161,783 blocks and 257,519 variables -- the product of those two numbers being about 41 billion. Now, we're not likely going to need that much space since we use a sparse representation. But even 1/100th would really hurt. And of course this rules out bit vectors.

Last edited on 2005-12-02 03:24:42 by StephenWeeks.

eXene

eXene is a multi-threaded X Window System toolkit written in ConcurrentML.

There is a group at K-State working toward eXene 2.0.

Last edited on 2005-12-01 04:04:43 by StephenWeeks.

fxp

fxp is an XML parser written in Standard ML.

It has a patch to compile with MLton.

Last edited on 2005-09-09 19:15:51 by StephenWeeks.

mGTK

mGTK is a wrapper for GTK+, a GUI toolkit.

We recommend using mGTK 0.93, which is not listed on their home page, but is available at the file release page. To test it, after unpacking, do `cd examples; make mlton`, after which you should be able to run the many examples (`signup-mlton`, `listview-mlton`, ...).

Also see

- Glade

Last edited on 2005-12-02 03:33:24 by StephenWeeks.