MLRISC
A framework for retargetable and optimizing compiler back ends

Allen Leung

New York University
719 Broadway, Rm. 714
New York, NY 10003.
leunga@cs.nyu.edu

Lal George

Bell Laboratories
600–700 Mountain Ave.
Murray Hill, NJ 07974–0636.
george@research.bell-labs.com

May 17, 2018

Abstract

Writing native code generators for modern processors is a significant investment. Unfortunately it is difficult to reuse this investment for other architectures, and even more difficult to reuse for other source language compilers. MLRISC is a customizable optimizing back-end written in Standard ML[1] and has been successfully retargeted to multiple architectures. MLRISC deals elegantly with the special requirements imposed by the execution model of different high-level, typed languages, by allowing many components of the system to be customized to fit the source language semantics and runtime system requirements.

[1] url: http://cm.bell-labs.com/cm/cs/what/smlnj/sml.html

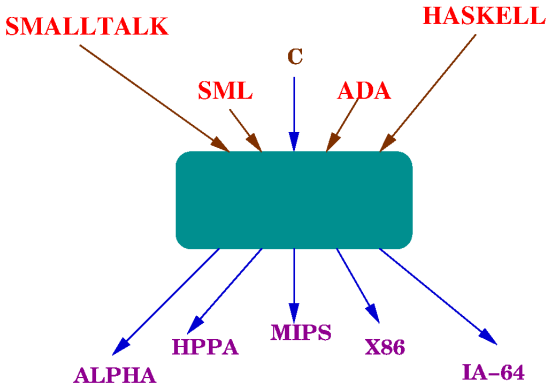
Contents

Part I
MLRISC

1 MLRISC

A framework for retargetable and optimizing compiler back ends

Lal George[2] Allen Leung[3]
Bell Labs New York University



MLRISC logo

Contributors[4]

Writing native code generators for modern processors is a significant investment. Unfortunately it is difficult to reuse this investment for other architectures, and even more difficult to reuse for other source language compilers. MLRISC is a customizable optimizing back-end written in Standard ML[5] and has been successfully retargeted to multiple architectures. MLRISC deals elegantly with the special requirements imposed by the execution model of different high-level, typed languages, by allowing many components of the system to be customized to fit the source language semantics and runtime system requirements.

The Overview pages on the left provide an introduction the MLRISC system, mostly from the client's perspective, while the System pages give a more detailed look at the innards, and are of interest to MLRISC hackers. As usual, development of the system has outpaced the documentation process substantally; thus the latter part of the document is incomplete but it may still be useful.

These pages are also available in tech report[6] form.

[4]url: `contributors.html`
[5]url: `http://cm.bell-labs.com/cm/cs/what/smlnj/sml.html`
[6]url: `../latex/mlrisc.ps`

2 Contributors

2.0.1 Past

- Florent Guillame (INRIA)
- George C. Necula (CMU)
- Ken Cline (CMU)
- Andrew Bernard (CMU)
- Dino Oliva (NEC)

2.0.2 Present

- Allen Leung (NYU)
- Fermin Reig (University of Glasgow)

3 Requirements

The most up-to-date MLRISC system requires Standard ML of New Jersey[7] version 110.0.3 or later.

[7] url: `http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`

4 How to Obtain MLRISC

There are a few ways to obtain the MLRISC system.

1. An old version of MLRISC is available from this link[8]. This version is stable but very out-dated, and does not contain the most up-to-date features.
2. New experimental versions are available from the SML/NJ software page[9] as part of the SML/NJ compiler releases. These versions are relative stable, but do not include the entire MLRISC source tree.
3. Allen[10] keeps an up-to-date version of MLRISC at NYU for private use. This version includes everything but is under constant changes, so beware! To access the CVS repository, set your `CVSROOT` environment variable to

   ```
   :pserver:mlrisc@react-ilp.cs.nyu.edu:/home/leunga/mlrisc
   ```

   and checkout the repository using

   ```
   cvs co MLRISC++
   ```

   The password to use is `mlrisc`.
4. Generally speaking, you can get the latest version of MLRISC by asking Lal[11].

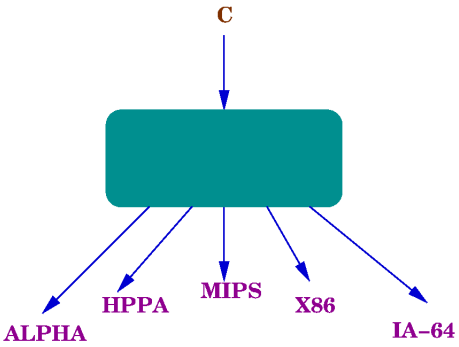
MLRISC is *free, open source* software, and is released under the SML/NJ license[12].

[8] url: `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/MLRISC/quick-tour/index.html`
[9] url: `http://cm.bell-labs.com/cm/cs/what/smlnj/software.html`
[10] url: `mailto:leunga@cs.nyu.edu`
[11] url: `mailto:george@research.bell-labs.com`
[12] url: `http://cm.bell-labs.com/cm/cs/what/smlnj/license.html`

Part II
Overview

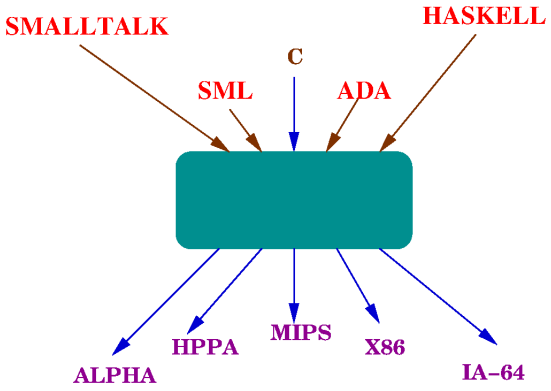
5 Problem Statement

Writing a native code generator for any language is a significant investment, especially for todays modern processors with require extensive compiler support to achieve high performance. The algorithms that must be used to generate high quality code are complex, sometimes quite delicate, and require substantial infrastructure.



Retargeting compiler

A specific architecture has a relatively short life time in relation to the time taken to build the code generator, and one quickly needs the ability to retarget to new versions of the architecture, or to different target architectures. This is by no means an open problem. There are many compilers today that target multiple architectures, however the quality of code varies. For example, *lcc* by Chris Fraser and David Hansen does no back end optimizations; *gcc* from the Free Software Foundation does extensive peephole and simple data flow optimizations, and falls short on advanced superscalar optimizations; and finally the *IMPACT* compiler done by the Impact group at the University of Illinois specializes in more advanced superscalar and predicated architectures.



UNCOL?

Assuming the retargeting issue is solved, one would like to use all the developed infrastructure for multiple source languages. This problem is far from solved; even though *gcc* has been used for multiple languages like Ada, Pascal, and Modula III, each of these have similiar execution models or were forced to adopt C conventions. *gcc* cannot be used directly for languages such as Lisp, Smalltalk, Haskell, or ML

that have radically different execution models and special requirements to support advanced language features.

6 Contributions

The optimizations provided by MLRISC are at a similar level to those performed by the Impact compiler; several target back ends exist (Dec Alpha, HPPA, Sparc, x86, and PPC); but more importantly, the framework has been demonstrated in real use[13] for languages with radically different execution models. These include:

Compiler	Association
SML/NJ	Bell Labs and Princeton
TIL	CMU
Tiger	Princeton
C–	OGI
SML/Regions	DIKU
Moby	Bell Labs

The strength of MLRISC lies in the ability to easily create high quality code generator for each of these systems. For example:

Tiger: Has an execution model very similar to C with stack allocated activation frames, and also maintains static and dynamic chains to support lexical scoping.

TIL: Is similar to C in its use of activation frames, however it uses a *typed intermediate language* that supports *almost tag-free* garbage collection. This has severe implications on the interaction of spilling and garbage collection. The set of live variables and their locations, be it registers or frame slots, is recorded in a trace table for a specific program point. When spilling occurs, it is necessary to adjust some of these trace tables to reflect the new locations of live variables.

SML/NJ: Has no runtime stack, but stores all execution context in a garbage collected heap. This arrangement imposes special requirements for spilling registers. SML/NJ also does *dynamic linking* — that is to say, no use is made of a conventional linker, but machine code is generated directly and linked into the interactive environment, dynamically.

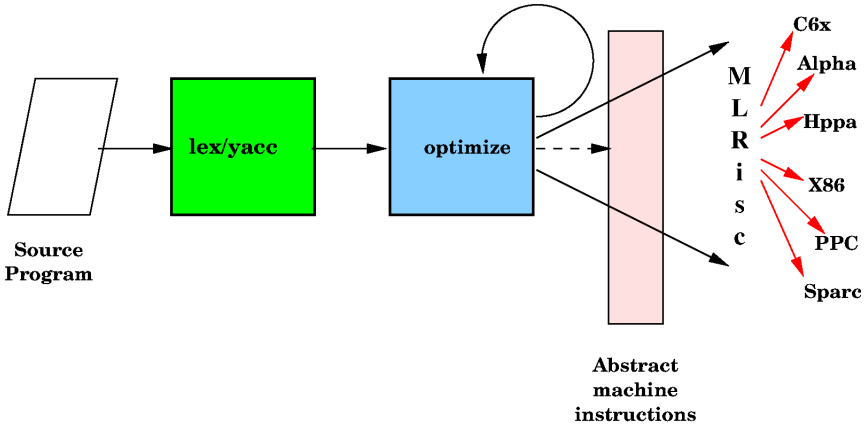
C–: Is a C-like portable assembly language used as an intermediate language for high level typed language, and provides direct compilation support for exceptions and precise garbage collection. In addition, it allows interoperability with C function calls.

It is not uncommon for any of these systems to store special global values in dedicated registers, and use their own parameter passing and callee-save conventions. In any language that supports garbage collection, there are also the issues of generating gc type maps, and gc-safety in aggressive optimizations. MLRISC deals with all these important issues by allowing customization of many aspects of the system.

[13] url: `systems.html`

7 MLRISC Based Compiler

A traditional compiler will typically consist of a lex/yacc based front end, an optimization phase that is repeatedly invoked over some intermediate representation, and finally a back end code generation phase. The intermediate representation is usually at a level of detail appropriate to the optimization being performed, and may be far removed from the native instructions of the target architecture. The back end proceeds by translating the intermediate representation into instructions and registers for an abstract machine that is much closer to the target architecture. Retargetting is then achieved by mapping the registers and instructions of the abstract machine to registers and instructions of the target architecture.



MLRISC based compiler

An MLRISC based compiler, on the other hand, translates the intermediate representation into MLRISC instructions and it is the MLRISC instructions that get mapped onto instructions of the target architecture. Another possibility is to translate the front end abstract machine instructions instead of the intermediate representation. Once MLRISC instructions have been generated, nearly all aspects of high quality code generation come for free. A long story would be cut short if MLRISC were just another abstract machine.

The key idea behind MLRISC is that there is no single MLRISC instruction set or intermediate program representation, but the MLRISC intermediate representation is specialized to the needs of the front end source language being compiled. The specialization does not stop there, but the:

- target instruction set,
- flowgraph, and
- entire optimization suite

are specialized to the needs of the front end. The ability to consistently specialize each of these to create a back end for a specific language, summarizes the characteristics of MLRISC that distinguishes it from other retargetable backends.

It is important to emphasize that little optimizations performed on the MLRISC intermediate representation. Most optimizations are done on a flowgraph of target machine instructions, to enable optimizations that take advantage of the characteristics of each architectural. The MLRISC intermediate representation is just used as a stepping stone to get to the flowgraph.

8 MLRISC Intermediate Representation

The MLRISC intermediate language is called *MLTREE* At the lowest level, the core of MLTREE is a *Register Transfer Language (RTL)* but represented in tree form. The tree form makes it convenient to use tree pattern matching tools like BURG (where appropriate) to do target instruction selection. Thus a tree such as:

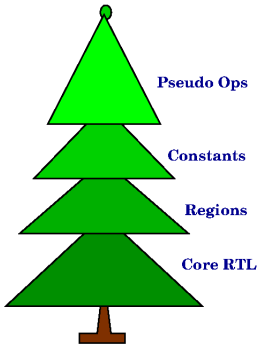
```
MV(32, t,
   ADDT(32, MULT(32, REG(32, b), REG(32, b)),
            MULT(32, MULT(REG(32, a), LI(4)), REG(32, c))))
```

computes `t := b*b + 4*a*c` to 32-bit precision. The nodes `ADDT` and `MULT` are the trapping form of addition and multiplication, and `LI` is used for integer constants. An infinite number of registers are assumed by the model, however depending on the target machine the first `0..K` registers map onto the first `K` registers on the target machine. Everything else is assumed to be a pseudo-register. The `REG` node is used to indicate a general purpose register.

The core MLTREE language makes no assumptions about instructions or calling convections of the target architecture. Trees can be created and combined in almost any form, with certain meaningless trees such as `LOAD(32, FLOAD(64, LI 0))` being forbidden by the MLTREE type structure.

Such pure trees are nice but inadequate in real compilers. One needs to be able to propagate front end specific information, such as frame sizes and frame offsets where the actual values are only available after register allocation and spilling. One could add support for frames in MLRISC, however this becomes a slippery slope because some compilers (e.g. SML/NJ) do not have a conventional notion of frames — indeed there is no runtime stack in the execution of SML/NJ. A frame organization for one person may not meet the needs for another, and so on. In MLRISC, the special requirements of different compilers is communicated into the MLTREE language, and subsequently into the optimizations phases, by specializing the MLTREE data structure with client specific information. There are currently *five* dimensions over which one could specialize the MLTREE language.

Constants Constants are an abstraction for integer literals whose value is known after certain phases of code generation. Frame sizes and offsets are an example.



MLRISC intermediate representation

Regions While the data dependencies between arithmetic operations is implicit in the instruction, the data dependencies between memory operations is not. Regions are an abstract view of memory that make this dependence explicit and is specially useful for instruction reordering.

Pseudo-ops Pseudo-ops are intended to correspond to pseudo-op directives provided by native assemblers to lay out data, jump tables, and perform alignment.

Annotations Annotations[14] are used for injecting semantics and other program information from the front-end into the backend. For example, a probability annotation can be attached to a branch instruction. Similarly, line number annotations can be attached to basic blocks to aid debugging. In many language implementations function local variables are spilled to activation frames on the stack. Spill slots contribute to the size of a function's frame. When an instruction produces a spill, we may need to update the frame associated to that instruction (increase the size of its spilling area). The frame for the current function can be injected in an annotation, which can be later examined by the spill callback during register allocation.

Annotations are implemented as an universal type and can be arbitrarily extended. Individual annotations can be associated with compiler objects of varying granularity, from compilation units, to regions, basic blocks, flow edges, and down to the instructions.

User Defined Extensions In the most extreme case, the basic constructors defined in the MLTREE language may be inadequate for the task at hand. MLTREE allows the client to arbitrarily extend the set of statements and expressions to more closely match the source language and the target architecture(s).

For example, when using MLRISC for the backend of a DSP compiler it may be useful to extend the set of MLRISC operators to include fix point and saturated arithmetic. Similarly, when developing a language for loop parallelization, it may be useful to extend the MLTREE language with higher-level loop constructs.

8.1 Examples

In the SML/NJ compiler, an encoding of a list of registers is passed to the garbage collector as the roots of live variables. This encoding cannot be computed until register allocation has been performed, therefore the integer literal encoding is represented as an abstract constant[15].

Again, in the SML/NJ compiler, most stores are for initializing records in the allocation space, therefore representing every slot in the allocation space as a unique region allows one to commute most store instructions. Similarly, most loads are from *immutable* records, and a simple analysis marks these are being accesses to *read-only* memory. Read-only memory is characterized as having multiple *uses* but no *definitions*.

In the TIL compiler, a *trace table* is generated for every call site that records the set of live variables, their location (register or stack offset), and the type associated with the variable. This table is integrated into the program using the abstract pseudo-op mechanism. An interesting aspect of these tables is that they may need adjustment based on the results of register spilling.

The more convention use of the psuedo-op abstraction is to propagate function prologue and epilogue information.

The constants abstraction are created by a tree node called `CONST`. In the SML/NJ compiler, the tree that communicates garbage collection information looks like:

[14] url: `annotations.html`
[15] url: `constants.html`

```
MV(32, maskReg, CONST{r110,r200,r300,r400 ...})
```

where `maskReg` is a dedicated register. On the DEC Alpha, this would get translated to:

```
LDA maskReg, {encode(r110,r200,r300,r400, ...)}
```

which indicates that the alpha instruction set (and optimization suite) know about these types of values. Further, after register allocation, the `LDA` instruction may not be sufficient as the encoding may result in a value that is too large as an operand to `LDA`. Two instructions may ultimately be required to load the encoding into the `maskReg` register. This expansion is done during span-dependency resolution[16].

All these examples are intended to indicate that one intermediate representation and optimization suite does not fit all, but that the intermediate representation and optimization suite needs to be specialized to the needs of the client.

[16] url: `span-dep.html`

9 MLRisc Generation

Every compiler will eventually compile down to an abstract machine that it believes will execute source programs efficiently. The abstract machine will typically consists of abstract machine registers and instructions, one or more stacks, and parameter passing conventions. The hope is that all this will map down efficiently onto the target machine. Indeed, the abstract machine should be reasonably close to architectures that are envisioned as possible targets. Several step need to be followed in the generation of MLRisc.

1. The first step in generating target machine code is to define the MLRisc intermediate representation after it has been appropriately specialized. The interfaces that describe the dimensions of specialization are quite simple. Depending on the compiler, these may be target dependent; for example, in the SML/NJ compiler, the encoding of registers used to indicate the roots of garbage collection depend on how the runtime system decodes the information.

2. The only real connection between the MLRisc intermediate representation and the target machine is that the first $0..K-1$ MLRisc registers map onto the first K physical registers on the target machine. Thus some mapping of dedicated abstract machine registers to physical target registers is required. It is not always necessary to map abstract machine registers to physical machine registers. For example, on architectures like the x86 with few registers, some abstract machine registers may be mapped to fixed memory locations. Thus an abstract machine register like the `maskReg` may have something like:

```
LOAD(32, LABEL maskRegLab)
```

   spliced instead.

3. The unit of compilation is called a cluster[17] which is the smallest unit for inter-procedural optimizations. A cluster will typically consist of several entry points that may call each other, as well as call local functions in the module. For maximum flexibility, the parameter passing convention for local functions should be specialized by the register allocator[18].

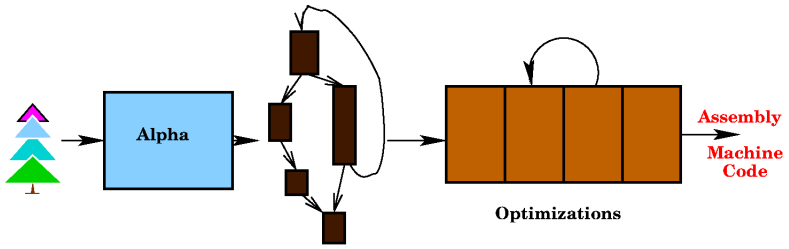
   Once the MLRisc trees for a cluster have been built, they must be converted into target assembly or machine code. This is done by building up a function (*codegen*) that glues together optimizations modules that have been specialized. For example, the target instruction set must be specialized to hold the MLRisc constants; the flowgraph must be specialized to carry these instructions as well as the MLRisc pseudo-ops; the optimization modules must know about several front end constraints such as how to spill registers.

If the module that translates the abstract machine instructions into MLRisc instructions has been appropriately parameterized, then it can be reused for multiple target architectures. For high level languages it is better to generate MLRisc instructions from the high level intermediate form used by the front end of the compiler.

[17] url: `cluster.html`
[18] url: `mlrisc-ra.html`

10 Back End Optimizations

Once MLRisc trees have been generated, they are passed into a module that generates a flowgraph of target machine instructions. Again, this module and all subsequent optimization phases have been specialized to the front end. Nearly all instruction selection modules provided by MLRISC use a simple tree

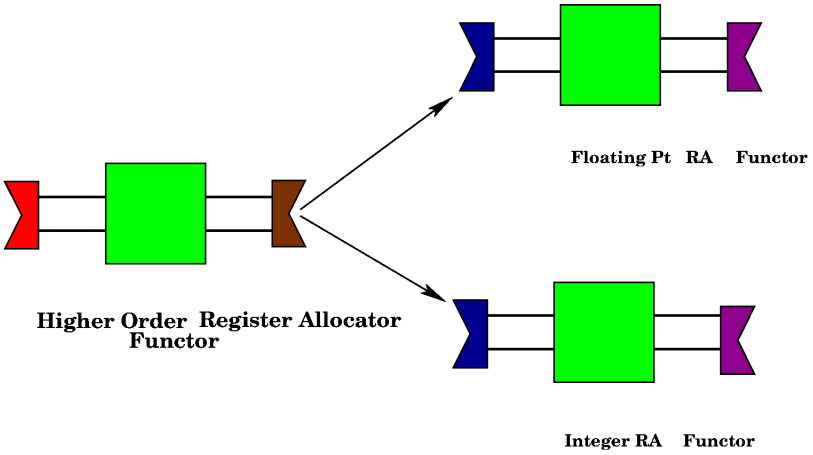


Back end optimizations

pattern matching algorithm rather than the more heavy weight BURG tools — including the x86 It is important to emphasis that all optimizations are performed on the flowgraph of target machine instructions and *not* MLRisc immediate IR. There is complete flexibility in the order, and nature of the optimizations performed.

11 Register Allocation

All the optimization modules are written in a generic fashion but parameterized over architecture and client information. The Standard ML module system is a central mechanism to the design and organization of MLRISC. Parameterized modules in Standard ML are provided by *functors*, that takes the specification of input modules and produces a module that matches some output specification. In particular, SML/NJ modules are *higher order*, which means that a functor can yield functors as a result. I will use register allocation as an example.



Back end optimizations

The register allocator is written has a higher order functor which when applied to suitable arguments produces an integer or floating point register allocator. The figure is simplifed because the output functor is not restricted to integer and floating point allocators but could also be other types of allocators, for example, condition code. The integer and floating point register allocators are functors that only take *client specific* parameters as input, whereas the higher-order takes architectural parameters as input. The client specific parameters include:

```
nFreeRegs : int
dedicated : int list
spill : ..
reload : ..
```

where:

`nFreeRegs` is the number of free registers or essentially the number of colors available for coloring the interference graph.

`dedicated` is the list of dedicated registers. It is useful to exclude these from the graph-color process to reduce the size of the data structures created.

`spill/reload` are functions that describe how to spill and reload registers that need to be spilled or reloaded in an instruction. These two functions are perhaps the most complicated pieces of information that need to be supplied by a client of MLRISC.

The architecture specific parameters supplied to the higher-order functor include:

```
firstPseudoReg : int
maxPseudoR : unit -> int
defUse : instruction -> (int list * int list)
```

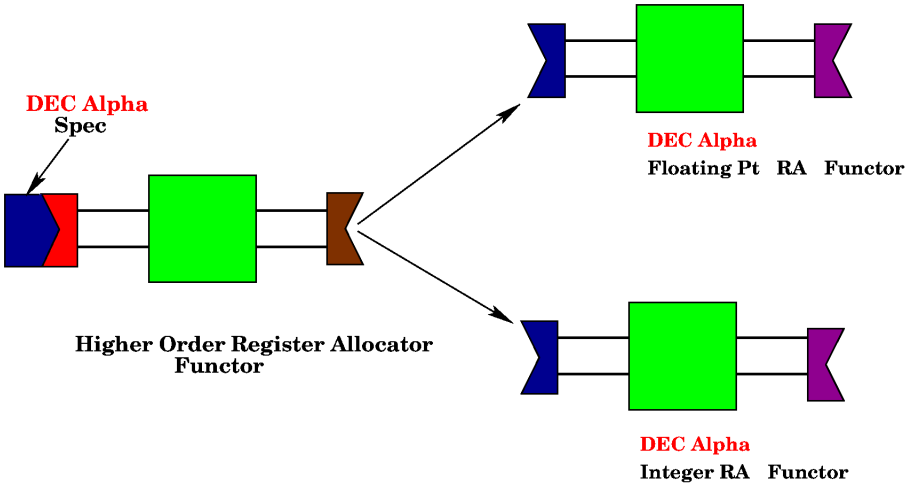
where:

`firstPseudoR` is an integer representing the first pseudo register. Any register below this value is a physical register.

`maxPseudoR` is a function that returns an integer indicating the number of the highest pseudo-register that has been used in the program. This number is useful in estimating the intial size of various tables.

`defUse` is a function that returns the registers defined and used by an instruction.

These parameters are largely self explanatory, however, there are addition architectural parameters that relate to the internal representation of instructions that would be ugly to explain. For example there is the need for a module that does liveness analysis over the register class that is being allocated. This type of complexity can be shielded from a user. For the DEC Alpha the situation is as shown in the figure:



Back end optimizations

The client only sees the functors on the right, to which only client specific information need be provided. There is the illusion of a dedicated DEC Alpha integer and floating point register allocator. There are several advantages to this:

- The architectural parameters that are implementation specific do not need to be explained to a user, and are supplied by someone that intimately understands the port to the target architecture.
- The number of parameters that a client supplies is reduced.
- The parameters that the client supplies is restricted to things that concern the front end.

12 Machine Description

12.1 Overview

MDGen is a simple tool for generating various modules in the MLRISC customizable code generator directly from machine descriptions. These descriptions contain architectural information such as:

1. How the the register file(s) are organized.
2. How instructions are encoded in machine code: MLRISC uses this information to generate machine instructions directly into a byte stream. Directly machine code generation is used in the SML/NJ compiler.
3. How instructions are pretty printed in assembly: this is used for debugging and also for assembly output for other non-SML/NJ backends.
4. How instructions are internally represented in MLRISC.
5. Other information needed for performing optimizations, which include:
   (a) The register transfer list (RTL) that defines the operational semantics of the instruction.
   (b) Delay slot mechanisms.
   (c) Information for performing span dependency resolution.
   (d) Pipeline and reservation table characteristics.

Currently, item 5 is not ready for prime time.

12.1.1 Why MDGen?

MLRISC manipulates all instruction sets via a set of abstract interfaces, which allows the programmer to arbitrarily choose an instruction representation that is most convenient for a particular architecture. However, various functions that manipulate this representation must be provided by the instruction set's programmer. As the number and complexities of each optimizations grow, and as the number of architectures increases, the functions for manipulating the instructions become more numerous and complex. In order to keep the effort of developing and maintaining an instruction set manageable, the MDGen tool is developed to (partially) automate this task.

12.1.2 Syntax

MDGen's machine descriptions are written in a syntax that is very much like that of Standard ML[19]. Most core SML constructs are recognized. In addition, new declaration forms specific to MDGen are used to specify architectural information.

[19] url: `http://cm.bell-labs.com/cm/cs/what/smlnj/sml.html`

Reserved Words All SML keywords are reserved words in MDGen. In addition, the following keywords are also reserved:

```
always architecture assembly at backwards big bits branching called
candidate cell cells cellset debug delayslot dependent endian field
fields formats forwards instruction internal little locations lowercase
name never nodelayslot nullified opcode ordering padded pipeline predicated
register rtl signed span storage superscalar unsigned uppercase
verbatim version vliw when
```

Two kinds are quotations marks are also reserved:

```
[[ ]]
‘‘ ’’
```

The first `[[ ]]` is for describing semantics. The second `‘‘ ’’` is for describing assembly syntax.

Syntactic Sugar MDGen recognizes the following syntactic sugar.

Record abbreviations Record expressions such as `x=x,y=y,z=z` can be simplified to just `x,y,z`.

Binary literals Literals in binary can be written with the prefix `0b` (for integer types) or `0wb` (for word types). For example, `0wb101111` is the same as `0wx2f` and `0w79`.

Bit slices A bit slice, which extracts a range of bits from a word, can be written using an `at` expression. For example, `w at [16..18]` means the same thing as `Word32.andb(Word32.>>(w, 0w16),0w7)`, i.e. it extracts bit 16 to 18 from `w`. The least significant bit the zeroth bit.

In general, we can write:

```
w at [range1, range2, ..., rangen]
```

to extract a sequence of slices from w and concatenate them together. For example, the expression

```
0wxabcd at [0..3, 4..7, 8..11, 12..15]
```

swap the 4 nybbles from the 16-bit word, and evaluates to `0wxdcba`.

Signature Signature declarations of the form

```
val x y z : int -> int
```

can be used as a shorthand for the more verbose:

```
val x : int -> int
val y : int -> int
val z : int -> int
```

12.1.3 Elaboration Semantics

Unfortunately, there is no complete formal semantics of how an MDGen specification elaborates. But generally speaking, a machine description is a just a structure (in the SML sense). Different components of this structure describe different aspects of the architecture.

Syntactic Overloading In general, the syntactic overloading are used heavily in MDGen. There are three types of definitions:

- Definitions that defines properties of the instruction set.
- Definitions of functions and terms that are in the RTL meta-language. The syntax of MDGen's RTL language is borrowed heavily from Lambda-RTL, which in turns is borrowed heavily from SML.
- Definitions of functions and types that are to be included in the output generated by the MDGen tool. These are usually auxiliary helper functions and definitions.

In general, entities of type 2, when appearing in other context, are properly meta-quoted in the semantics quotations `[[ ]]`.

12.1.4 Basic Structure of A Machine Description

The machine description for an architecture are defined via an `architecture` declaration, which has the following general form.

```
architecture name =
struct
    architecture type declaration
    endianess declaration
    storage class declarations
    locations declarations
    assembly case declarations
    delayslot declaration
    instruction machine encoding format declarations
    nested structure declarations
    instruction definition
end
```

12.2 Describing the Architecture

12.2.1 Architecture type

Architecture type declaration specifies whether the architecture is a superscalar or a VLIW/EPIC machine. Currently, this information is ignored.

architecture type declaration `::= superscalar | vliw`

12.2.2 Storage class

Storage class declarations specify various information about the registers in the architecture. For example, the Alpha has 32 general purpose registers and 32 floating point registers. In addition, MLRISC requires that each architecture specifies a (pseudo) register type[20] for holding condition codes (`CC`). To specify these information in MDGen, we can say:

[20] Called cellkind in MLRISC.

```
storage
  GP "r" = 32 cells of 64 bits in cellset called "register"
             assembly as (fn (30,_) => "$sp"
                           | (r,_)  => "$"^Int.toString r
                         )
| FP "f" = 32 cells of 64 bits in cellset called "floating point register"
             assembly as (fn (f,_) => "$f"^Int.toString f)
| CC "cc" = cells of 64 bits in cellset GP called "condition code register"
             assembly as "cc"
```

- There are 32 64-bit general purpose registers, 32 64-bit floating point registers, while `CC` is not a real register type.
- Cellsets are used by MLRISC for annotating liveness information in the program. The clause `in cellset` states that register type `GP` and `FP` are allotted their own components in the cellset, while the register type `CC` are put in the same cellset component as `GP`.
- The clause `assembly as` specifies how each register is to be pretty printed. On the Alpha, general purpose register are pretty printed with prefix `$`, while floating point registers are pretty printed with the prefix `$f`. A special case is made for register 30, which is the stack pointer, and is pretty printing as `$sp`. Pseudo condition code registers are pretty printed with the prefix `cc`.

12.2.3 Locations

Special locations in the register files can be declared using the `locations` declarations. On the Alpha, GPR 30 is the stack pointer, GPR 28 and floating point register 30 are used as the assembly temporaries. This special constants can be defined as follows:

```
locations
    stackptrR = $GP[30]
and asmTmpR   = $GP[28]
and fasmTmp   = $FP[30]
```

12.3 Specifying the Machine Encoding

12.3.1 Endianess

The endianess declaration specifies whether the machine is little endian or big endian so that the correct machine instruction encoding functions can be generated. The general syntax of this is:

endianess declaration `::= little endian | big endian`

The Alpha is little endian, so we just say:

```
little endian
```

12.3.2 Defining New Instruction Formats

How instructions are encoded are specified using `instruction format` declarations. An instruction format declaration has the following syntax:

instruction machine encoding format declarations ::=
```
  instruction formats n bits
    format1
  | format2
  | format3
  | ...
  | formatn-1
  | formatn
```

Each encoding format can be a primitive format, or a derived format.

Primitive formats A primitive format is simply specified by giving it a name and specifying the position, names and types of its fields. This is usually the same way it is described in a architectural reference manual.

Here is how we specify some of the (32 bit) primitive instruction formats used in the Alpha.

```
instruction formats 32 bits
  Memory{opc:6, ra:5, rb:GP 5, disp: signed 16}
| Jump{opc:6=0wx1a,ra:GP 5,rb:GP 5,h:2,disp:int signed 14}
| Memory_fun{opc:6, ra:GP 5, rb:GP 5, func:16}
| Branch{opc:branch 6, ra:GP 5, disp:signed 21}
| Fbranch{opc:fbranch 6, ra:FP 5, disp:signed 21}
| Operate0{opc:6,ra:GP 5,rb:GP 5,sbz:13..15=0,_:1=0,func:5..11,rc:GP 5}
| Operate1{opc:6,ra:GP 5,lit:signed 13..20,_:1=1,func:5..11,rc:GP 5}
```

For example, the format `Memory`

```
  Memory{opc:6, ra:5, rb:GP 5, disp: signed 16}
```

has a 6-bit opcode field, a 5-bit `ra` field, a 5-bit `rb` field which always hold a general purpose register, and a 16-bit sign-extended displacement field. The field to the left is positioned at the most significant bits, while the field to the right is positioned at the least. The widths of these fields must add up to 32 bits.

Similarly, the format `Jump`

```
Jumpopc:6=0wx1a,ra:GP 5,rb:GP 5,h:2,disp:int signed 14
```

contains a 6-bit opcode field which always hold the constant `0x1a`, two 5-bit fields `ra` and `rb` which are of type `GP`, and a 14-bit sign-extended field of type integer.

Each field in a primitive format has one of 5 forms:

name : *position*
name : *position* = *value*
name : *type position*
name : *type position* = *value*
_ : *position* = *value*

where *position* is either a width, or a bits range $n..m$, with an optional `signed` prefix. The last form, with a wild card for the field name, can be used to specify an anonymous field that always has a fixed value.

By default, a field has type `Word32.word`. If a type T is specified, then the function `emit_`T is implicitly called to convert the type into the appropriate encoding. The function `emit_`T are generated automatically by MDGen if it is a cellkind defined by the `storage` class declaration, or if it is a primitive type such as integer or boolean. There are also other ways to automatically generate this function (more on this later.)

For example, the format `Operate1`

```
Operate1{opc:6,ra:GP 5,lit:signed 13..20,_:1=1,func:5..11,rc:GP 5}
```

states that bits 26 to 31 are allocated to field `opc`, bits 21 to 25 are allocated to field `ra`, which is of type `GP`, bits 13 to 20 are allocated to field `lit`, bit 12 is a single bit of value 1, etc.

MDGen generates a function for each primitive format declaration of the same name that can be used for emitting the instruction. In the case of the Alpha, the following functions are generated:

```
val Memory : {opc:Word32.word, ra:Word32.word,
              rb:int, disp:Word32.word} -> unit
val Jump   : {ra:int, rb:int, disp:Word32.word} -> unit
val Operate1 : {opc:Word32.word, ra:int, lit:Word32.word,
                func:Word32.word, rc:int} -> unit
```

Derived formats Derived formats are simply instruction formats that are defined in terms of other formats. On the alpha, we have a `Operate` format that simplifies to either `Operate0` or `Operate1`, depending on whether the second argument is a literal or a register.

```
Operate{opc,ra,rb,func,rc} =
  (case rb of
    I.REGop rb => Operate0{opc,ra,rb,func,rc}
  | I.IMMop i  => Operate1{opc,ra,lit=itow i,func,rc}
  | I.HILABop le => Operate1{opc,ra,lit=Highle=le,func,rc}
  | I.LOLABop le => Operate1{opc,ra,lit=Lowle=le,func,rc}
  | I.LABop le => Operate1{opc,ra,lit=itow(LabelExp.valueOf le),func,rc}
  )
```

12.3.3 Generating Encoding Functions

In MLRISC, we represent an instruction as a set of ML datatypes. Some of these datatypes represent specific fields or opcodes of the instructions. MDGen lets us to associate a machine encoding to each datatype constructor directly in the specification, and automatically generates an encoding function for these datatypes.

There are two different ways of specifying an encoding. The first way is just to write the machine encoding directly next the constructor. Here's an example directly from the Alpha description:

```
structure Instruction =
struct
   datatype branch! =  (* table C-2 *)
      BR   0x30
                | BSR 0x34
                              | BLBC 0x38
    | BEQ  0x39 | BLT 0x3a | BLE  0x3b
```

```
     | BLBS 0x3c | BNE 0x3d | BGE  0x3e
     | BGT  0x3f

   datatype fbranch! = (* table C-2 *)
                FBEQ 0x31 | FBLT 0x32
     | FBLE 0x33             | FBNE 0x35
     | FBGE 0x36 | FBGT 0x37

   ...
end
```

The datatypes `branch` and `fbranch` represent specific branch opcodes for integer branches `BRANCH`, or floating point branches `FBRANCH`. On the Alpha, instruction `BR` is encoded with an opcode of `0x30`, instruction `BSR` is encoded as `0x34` etc. MDGen automatically generates two functions

```
 val emit_branch : branch -> Word32.word
 val emit_fbranch : branch -> Word32.word
```

that perform this encoding.

In the specification for the instruction set, we state that the `BRANCH` instruction should be encoded using format `Branch`, while the `FBRANCH` instruction should be encoded using format `Fbranch`.

```
structure MC =
struct
   (* Auxiliary function for computing the displacement of a label *)
   fun disp ... = ...
   ...
end

...

instruction
  ...

| BRANCH of branch * $GP * Label.label
  Branch{opc=branch,ra=GP,disp=disp label}

| FBRANCH of fbranch * $FP * Label.label
  Fbranch{opc=fbranch,ra=FP,disp=disp label}

| ...
```

Since the primitive instructions formats `Branch` and `FBranch` are defined with branch and fbranch as the type in the opcode field

```
| Branch{opc:branch 6, ra:GP 5, disp:signed 21}
| Fbranch{opc:fbranch 6, ra:FP 5, disp:signed 21}
```

the functions `emit_branch` and `emit_fbranch` are implicitly called.

Another way to specify an encoding is to specify a range, as in the following example:

```
datatype fload[0x20..0x23]! = LDF | LDG | LDS | LDT

datatype fstore[0x24..0x27]! = STF | STG | STS | STT
```

This states that `LDF` should be assigned the encoding `0x20`, `LDG` the encoding `0x21` etc. This form is useful for specifying a consecutive range.

12.3.4 Encoding Variable Length Instructions

Most architectures nowadays have fixed length encodings for instructions. There are some notatable exceptions, however. The Intel x86 architecture uses a legacy variable length encoding. Modern RISC machines developed for embedded systems may utilize space-reduction compression schemes in their instruction sets. Finally, VLIW machines usually have some form of NOP compression scheme for compacting issue packets.

12.4 Specifying the Assembly Formats

12.4.1 Assembly Case Declaration

The assembly case declaration specifies whether the assembly should be emitted in lower case, upper case, or verbatim. If either lower case or upper case is specified, all literal strings are converted to the appropriate case. The general syntax of this declaration is:

assembly case declaration `::=`

```
   lowercase assembly
 | uppercase assembly
 | verbatim  assembly
```

12.4.2 Assembly Annotations

Assembly output are specified in the assembly meta quotations `‘‘ ’’`, or string quotations `" "`. For example, here is a fragment from the Alpha description:

```
instruction
  ...
| LOAD of {ldOp:load, r: $GP, b: $GP, d:operand, mem:Region.region}
  ‘‘<ldOp> ^<r>, <d>()<mem>’’

| STORE of {stOp:store, r: $GP, b: $GP, d:operand, mem:Region.region}
  ‘‘<stOp> ^<r>, <d>()<mem>’’

| BRANCH of branch * $GP * Label.label
  ‘‘<branch> ^<GP>, <label>’’

| FBRANCH of fbranch * $FP * Label.label
  ‘‘<fbranch> ^<FP>, <label>’’

| CMOVE of {oper:cmove, ra: $GP, rb:operand, rc: $GP}
  ‘‘<oper> ^<ra>, <rb>, <rc>’’
```

```
| FOPERATE of {oper:foperate, fa: $FP, fb: $FP, fc: $FP}
  ``<oper>^<fa>, <fb>, <fc>''

| ...
```

All characters within the quotations `` `` '' `` have the same interpretation as in the string quotation `" "`, except when they are delimited by the *backquotes* `< >`. Here's how the backquote is interpreted:

- If it is <x> and x is a variable name of type t, and if an assembly function of type t is defined, then it will be invoked to convert x to the appropriate text.
- If it is <x> and x is a variable name of type t, and if an assembly function of type t is NOT defined, then the function `emit_`x will be called to pretty print x.
- If it is <e> where e is a general expression, then it will be used directly.

12.4.3 Generating Assembly Functions

Similar to machine encodings, we can attach assembly annotations to datatype definitions and let MDGen generate the assembly functions for us. Annotations take two forms, explicit or implicit. Explicit annotations are enclosed within assembly quotations `` `` '' ``.

For example, on the Alpha the datatype `operand` is used to represent an integer operand. This datatype is defined as follows:

```
datatype operand =
    REGop of $GP                    ``<GP>''
  | IMMop of int                    ``<int>''
  | HILABop of LabelExp.labexp      ``hi(<labexp>)''
  | LOLABop of LabelExp.labexp      ``lo(<labexp>)''
  | LABop of LabelExp.labexp        ``<labexp>''
  | CONSTop of Constant.const       ``<const>''
```

Basicaly this states that `REGop r` should be pretty printed as `$r`, `IMMop i` as `i`, `HILABexp le` as `hi(le)`, etc.

Implicit assembly annotations are specified by simply attaching an exclamation mark at the end of the datatype name. This states that the assembly output is the same as the name of the datatype constructor[21]. For example, the datatype `operate` is a listing of all integer opcodes used in MLRISC.

```
datatype operate! = (* table C-5 *)
    ADDL  (0wx10,0wx00)                       | ADDQ (0wx10,0wx20)
                        | CMPBGE(0wx10,0wx0f) | CMPEQ (0wx10,0wx2d)
  | CMPLE (0wx10,0wx6d) | CMPLT (0wx10,0wx4d) | CMPULE (0wx10,0wx3d)
  | CMPULT(0wx10,0wx1d) | SUBL  (0wx10,0wx09)
  | SUBQ  (0wx10,0wx29)
  | S4ADDL(0wx10,0wx02) | S4ADDQ (0wx10,0wx22) | S4SUBL (0wx10,0wx0b)
  | S4SUBQ(0wx10,0wx2b) | S8ADDL (0wx10,0wx12) | S8ADDQ (0wx10,0wx32)
  | S8SUBL(0wx10,0wx1b) | S8SUBQ (0wx10,0wx3b)
```

[21] But appropriately modified by the assembly case declaration.

```
  | AND   (0wx11,0wx00) | BIC    (0wx11,0wx08) | BIS (0wx11,0wx20)
                                                 | EQV (0wx11,0wx48)
  | ORNOT (0wx11,0wx28) | XOR    (0wx11,0wx40)

  | EXTBL (0wx12,0wx06) | EXTLH  (0wx12,0wx6a) | EXTLL(0wx12,0wx26)
  | EXTQH (0wx12,0wx7a) | EXTQL  (0wx12,0wx36) | EXTWH(0wx12,0wx5a)
  | EXTWL (0wx12,0wx16) | INSBL  (0wx12,0wx0b) | INSLH(0wx12,0wx67)
  | INSLL (0wx12,0wx2b) | INSQH  (0wx12,0wx77) | INSQL(0wx12,0wx3b)
  | INSWH (0wx12,0wx57) | INSWL  (0wx12,0wx1b) | MSKBL(0wx12,0wx02)
  | MSKLH (0wx12,0wx62) | MSKLL  (0wx12,0wx22) | MSKQH(0wx12,0wx72)
  | MSKQL (0wx12,0wx32) | MSKWH  (0wx12,0wx52) | MSKWL(0wx12,0wx12)
  | SLL   (0wx12,0wx39) | SRA    (0wx12,0wx3c) | SRL  (0wx12,0wx34)
  | ZAP   (0wx12,0wx30) | ZAPNOT (0wx12,0wx31)
  | MULL  (0wx13,0wx00)                         | MULQ (0wx13,0wx20)
                        | UMULH  (0wx13,0wx30)
  | SGNXL "addl" (0wx10,0wx00) (* same as ADDL *)
```

This definitions states that `ADDL` should be pretty printed as `addl`, `ADDQ` as `addq`, etc. However, the opcode `SGNXL` is pretty printed as `addl` since it has been explicitly overridden.

12.5 Defining the Instruction Set

How the instruction set is represented is declared using the `instruction` declaration. For example, here's how the Alpha instruction set is defined:

```
instruction
   DEFFREG of $FP
 | LDA of {r: $GP, b: $GP, d:operand}
 | LDAH of {r: $GP, b: $GP, d:operand}
 | LOAD of {ldOp:load, r: $GP, b: $GP, d:operand, mem:Region.region}
 | STORE of {stOp:store, r: $GP, b: $GP, d:operand, mem:Region.region}
 | FLOAD of {ldOp:fload, r: $FP, b: $GP, d:operand, mem:Region.region}
 | FSTORE of {stOp:fstore, r: $FP, b: $GP, d:operand, mem:Region.region}
 | JMPL of {r: $GP, b: $GP, d:int} * Label.label list
 | JSR of {r: $GP, b: $GP, d:int} * C.cellset * C.cellset * Region.region
 | RET of {r: $GP, b: $GP, d:int}
 | BRANCH of branch * $GP * Label.label
 | FBRANCH of fbranch * $FP * Label.label
 | ...
```

The `instruction` declaration defines a datatype and specifies that this datatype is used to represent the instruction set. Generally speaking, the instruction set's designer has complete freedom in how the datatype is structured, but there are a few simple rules that she should follow:

- If a field represents a register, it should be typed with the appropriate storage types `$GP`, `$FP`, etc. instead of `int`. MDGen will treat its value in the correct manner; for example, during assembly emission a field declared type `int` is printed as an integer, while a field declared type `$GP` is displayed as a general purpose register.
- MDGen recognizes the following special types: `label`, `labexp`, `region`, and `cellset`.

12.6 Specifying Instruction Semantics

MLRISC performs all optimizations at the granulariy of individual instructions, specialized to the architecture at hand. Many optimizations are possible only if the "semantics" of the instructions set to are properly specified. MDGen contains a *register transfer language* (RTL) sub-language that let us to describe instruction semantics in a modular and succinct manner.

The semantics of this RTL sub-language has been borrowed heavily from Norman Ramsey's and Jack Davidson's Lambda RTL. There are a few main differences, however:

- The syntax of our RTL language is closer to that of ML than Lambda RTL.
- Our RTL language, like that of MDGen, is tied closely to MLRISC.

12.7 How to Run the Tool

12.8 Machine Description

Here are some machine descriptions in varing degree of completion.

- `Sparc`[22]
- `Hppa`[23]
- `Alpha`[24]
- `PowerPC`[25]
- `X86`[26]

12.9 Syntax Highlighting Macros

- For vim 5.3[27]

[22] **file:** `sparc/sparc.md`
[23] **file:** `hppa/hppa.md`
[24] **file:** `alpha/alpha.md`
[25] **file:** `ppc/ppc.md`
[26] **file:** `X86/X86.md`
[27] url: `md.vim`

13 Garbage Collection Safety

13.1 Motivation

High level languages such as SML make use of garbage collectors to reclaim unused storage at runtime. For generality, I assume that a precise, compacting garbage collector is used. In general, low-level optimizations that reorder instructions pass *gc safepoints*, when applied naively, are not safe. In general, two general classes of safety issues can be identified:

derived values A derived value x is a value that are dependent on the addresses of one of more heap allocated objects $a1, a2, a3, \ldots$ and/or the recent branch history. When these allocated objects $a1, a2, a3, \ldots$ are moved by the garbage collector, x has to be adjusted accordingly.

For example, inductive variable elimination may transformed an array indexing into a running pointer to the middle of an array object. Such running pointer is a derived value and is dependent on the starting address of the array.

The main difficulty in handling a derived value x during garbage collection is that sometimes it is impossible or counter-productive to recompute from $a1, a2, a3, \ldots$. For example, when the recent branch history is unknown, or when the precise relationship between x and $a1, a2, a3, \ldots$ cannot be inferred from context. We call these *unrecoverable* derived values.

incomplete allocation If heap allocation is performed inlined, then code motion may render some allocation incomplete at a gc safepoint. In general, incomplete allocation has to be completed, or rolled backed and then reexecuted after garbage collection, when the source language semantics allow it.

Typically, two gc safepoints cannot be separated by an unbounded number of allocations, which implies that in general, optimizations that move instructions between basic blocks are unsafe when naively applied, which greatly limits the class of optimizations in such an environment to trivial basic block level optimizations. framework is a necessity.

13.2 Safety Framework

MLRISC contains a gc-safety framework for performing aggressive machine level optimizations, including SSA-based scalar optimizations, global instruction scheduling, and global register allocation. Unlike previous work in this area, phases that perform optimizations and phases that maintain and update garbage collection information are completely separate, and the optimizer is constructed in a fully modular manner. In particular, gc-types and safety constraints are *parameterizable* by the source language semantics, the object representation, and the target architectures.

This framework has the following overall structure:

Garbage collection invariants annotation The front-end client is responsible for annotating each value in the program with a *gc type*, which is used to specify the abstract object representation, and the constraints on code motion that may be applied to such a value. The front-end uses an architecture independent RTL[28] language for representing the program, thus this annotation phase is portable between target architectures.

GC constraints propagation After instruction selection, gc constraint are propagated throughout the machine level program representation. Again, for portability, gc typing rules are specified in terms

[28] url: `mltree.html`

of the RTL [29] of the machine instructions. In this phase, unsafe code motions which exposes unrecoverable derived values to gc safepoints are automatically identified. (Pseudo) control dependence and anti-control dependence constraints are then added the program representation to prohibit all gc-unsafe code motions.

Machine level optimizations After constraints propagation, traditional machine level optimizations such as SSA optimizations and/or global scheduling are applied, without regard to gc information. This is safe because all gc safety constraints have been translated into the appropriate code motion constraints.

GC type propagation and gc code generation GC type inference is performed when all optimizations have been performed. GC safepoints are then identified and the root sets are determined. In addition, compensation code are inserted at gc points for repairing recoverable derived values.

13.3 Concurrency Safety

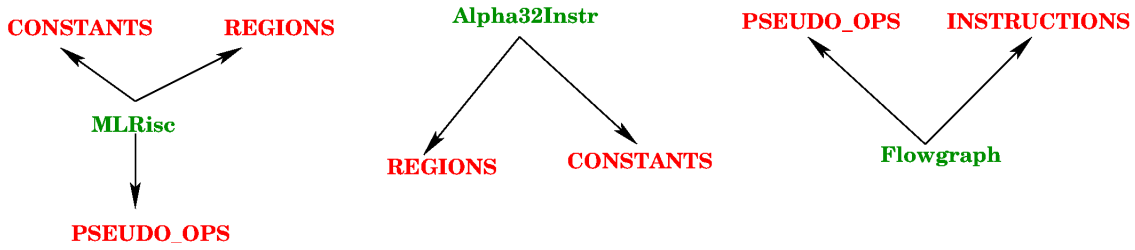
In the presence of *concurrency*, i.e. multiple threads of control that communicate via a shared heap, the above framework will have to slightly extended. As in before, we assume that context switching can only occur at well-defined *safepoints*. The crucial aspect is that values that are live at safepoints must be classified as *local* or *global*. Local values are only observable from the local thread, while global values are potentially observable and mutable from other threads. The invariants to maintain are as follows:

- Only local and recoverable derived values may be live at a safepoint,
- Only local and recoverable allocation may be incomplete at a safepoint

[29] url: `mltree.html`

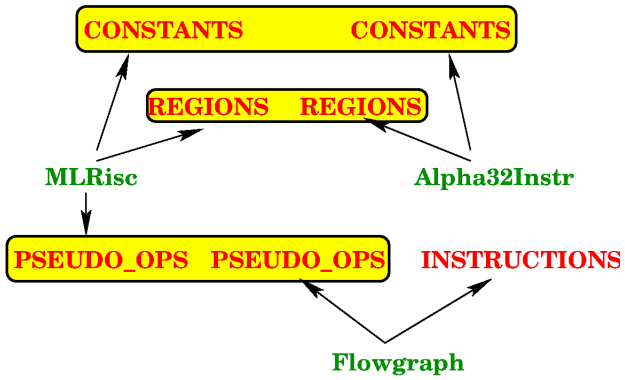
14 System Integration

In a heavily parameterized system like this, one very quickly ends up with a large number of modules and dependencies making it very easy to mix things up in the wrong way. For example, MLRisc is pa-



module dependencies

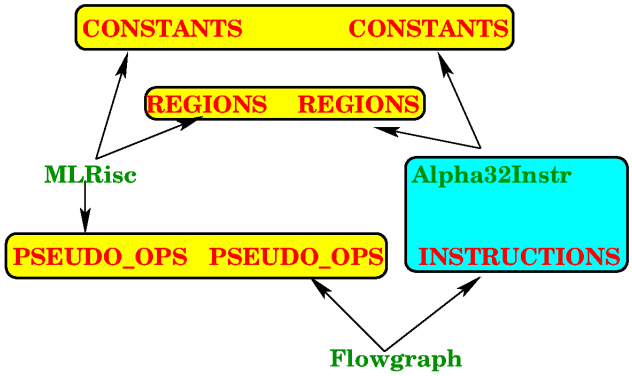
rameterised over pseudo-ops, constants, and regions. An instruction set must be parameterized over constants so that instructions that carry immediate operands can also carry these abstract constants. Instructions must also be parameterized over regions so that memory operations can be appropriately annotated. Finally, the flowgraph module must be parameterized over instructions it carries in basic blocks and pseudo-ops that describe data layout and alignment constraints.



sharing constraints

In integrating a system that involves these modules, it must be the case that they were created with the same base modules. That is to say the pseudo-ops in flowgraphs must be the same abstraction that was used to define the MLRisc intermediate representation. Alternatively, we want sharing constraints that assert that identity of modules used to specialize other modules. In Standard ML, this is a complete non-issue. A single line that says exactly that is all that is needed to maintain consistency, and the module system does the rest to ensure that the final system is built correctly.

In certain cases one wants to write a specific module for a particular architecture. For instance it may be desirable to collapse trap barriers on the DEC Alpha where it is legal to do so. The INSTRUCTIONS interface is abstract with no built-in knowledge of trap barriers as not all architectures have them. Further the DEC Alpha has fairly unique trap barrier semantics, that one may want to write an optimization module specific and dedicated to the Alpha instruction set and architecture, and forget about writing anything generic. In this case, the Standard ML module system allows one to say that a specific abstraction actually



Back end optimizations

is or matches a more detailed interface. That is to say the INSTRUCTION interface is really the DEC Alpha instruction set.

15 Optimizations

MLRISC assumes that all high level optimizations (target independent) have already been performed. This includes things like inlining, array dependence analysis, and array bounds check elimination. The target dependent optimizations that remain include register allocation, scheduling and traditional optimizations to support scheduling.

15.1 Register allocation

MLRISC includes a state-of-the-art graph-coloring based register allocator that has an aggressive algorithm for copy-propagation. The latter guarantees to eliminate copy instructions without introducing spills.

Spills in the register allocator are under the control of the client via call-backs to the front end. Where to spill registers and the associated information that must be maintained is client specific and varies with the compiler.

15.2 Scheduling for Superscalar Architectures

Several algorithms for acyclic global scheduling are provided. These include:

- Superblock,
- a variant of Bernstein/Rodeh, and
- Percolation based scheduling.

These algorithms tend to be quite complex and require a large number of support data structures and analysis. These include data structures such as:

- dominator/post dominator trees,
- loop nesting tree,
- control dependency graphs, and
- data dependency graphs.

Support analysis and optimization include:

- constant propagation,
- global value numbering,
- global code motion, and
- loop invariant hoisting.

15.3 VLIW Compilation

MLRISC also contains a framework for the compilation of predicated VLIW architectures. Currently, the following algorithms have been implemented.

- hyperblock formation
- hyperblock scheduling
- modulo scheduling

16 Graphical Interface

All the major data structures and intermediate program states can be viewed graphically using daVinci[30] and vcg[31] The following screen dumps are intended to represent the range of possibilities. Graphical tools like these are an indispensible debugging aid. Each of the dumps below were taken when generating code for the mandelbrot on the HPPA architecture. It will be necessary to make netscape fill the size of the screen to view these easily. Even though some of these graphs look quite complex, daVinci has several *navigational* modes that allow walking to successors, or predecessors, or navigating through a scaled down map of the graph. The navigational view is shown as another window, and the view into the graph that is being displayed is usually outlined in blue.

Control Flowgraph after Optimization:[32] Each basic block is shown with its dynamic profile and code before and after a specific optimization. This view saves having to pour through pages of assembly code listings – a tedious and frustrating activity.

SSA form:[33] The generated flow graph is converted to SSA form which makes many code improvement optimizations easy and efficient.

Data Dependency Graph[34] A graphical view of the data dependency graph and the various kinds of dependencies decorating the edges, provides a useful clue to why instructions got rearranged the way they did. The navigational view helps to control the complexity in the display.

[30] url: `http://www.Informatik.Uni-Bremen.DE/ davinci/`
[31] url: `http://www.cs.uni-sb.de/RW/users/sander/html/gsvcg1.html`

17 Line Counts

	SML/NJ	MLRISC
Generic	3,023	6,814
Hppa	725	2,285
Alpha	614	2,316
TOTAL	4,362	11,415

The table shows the number of lines involved in a basic MLRISC code generator for SML/NJ that only does graph coloring register allocation. The SML/NJ column shows the number of lines specific to SML/N and the MLRISC column shows the number of lines specific to MLRISC. The Generic shows the number of lines that are target independent for both SML/NJ and MLRISC. The Hppa and Alpha shows the number of lines that are target dependent for both the HP Hppa and DEC Alpha targets.

The bulk of the `3,023` generic to SML/NJ is involved in the generation of MLRisc trees. Once this is done the incremental cost of adding a target is between `600` to `700` lines.

The MLRISC column shows that the bulk of MLRISC is quite generic and a client is saved from writing `11,415` lines of code.

	SML/NJ	MLRISC
Generic	121 + 3,023	15,686 + 6,814
Hppa	32 + 725	920 + 2,285
Alpha	614	2,316
TOTAL	153 + 4,362	16,606 + 11,415

If one were to include the preliminary numbers for global acyclic scheduling in the above table, we find that the incremental cost required by the client is quite small – approximately `153` lines of which `121` are generic. However, the scheduling infra structure is quite large, a lot of it being quite generic.

18 Systems Using MLRISC

Currently these are the systems that are known to be using MLRISC.

- SML/NJ[35], a Standard ML compiler.
- C–[36], a portable assembly language.
- The Church Project[37]: compilation with flow types.
- The LGIC Project[38]: a compiler for the CHILL language, targeting PowerPC.
- The Moby Language[39]

Please send additions to Allen Leung[40]

[35] url: `http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`
[36] url: `http://www.dcs.gla.ac.uk/ reig/c--/index.html`
[37] url: `http://www.cs.bu.edu/groups/church/`
[38] url: `http://compiler.kaist.ac.kr/projects/lgic`
[39] url: `http://www.cs.bell-labs.com/who/jhr/moby/index.html`
[40] url: `mailto:leunga@cs.nyu.edu`

19 Future Work

19.1 Short Term

Detailed user manual: A detailed user manual describing the interfaces, algorithms, and examples on how to put together code generators.

Support for GC: There is a strong interaction with support for GC and global code motion. MLRISC aims at providing a generic framework for code generators, and finding the right level of information to support GC and global code motion is an issue. I think we have several solutions to address this that need more evaluation.

Other architectures: There is the need to port to other architectures like the MIPS, and the IA-64.

19.2 Long Term

Predicated VLIW compilation: Currently, the framework for predicated VLIW architectures compilation is incomplete, and contain only one back end (C6)

Other compilers: I would really like to see some major compiler effort bootstrapped with an MLRISC backend.

Verification It is extremely difficult to debug errors in modules that perform aggressive code reorganizations. Ideas from formal methods such as typed assembly language (TAL) or Proof Carrying Code (PCC) are worth investigating.

Part III
System

20 Architecture of MLRISC

20.1 Core Components

The core components of MLRISC allow the client to quickly construct an backend for various architectures. These components include:

- The MLTREE[41] language, which is a RTL-like intermediate language that is used by the client to communicate to the MLRISC system. A client is responsible for writing the module that generates MLTREE from the source program representation.
- Instruction selection modules[42], which generates target machine instructions from MLTREE.
- The Register Allocator[43], which performs register allocation.
- Assemblers[44], which emits assembly code.

For systems that require direct machine code generation, the following modules are included:

- Span dependency resolution[45] modules, which compute addresses from symbolic addresses, fill delay slots, and expand instructions that are *span dependent*
- Machine code emitters[46], which emit executable machine code into a binary stream.

20.2 Optimization Modules

In addition, MLRISC has been enhanced to support various types of machine level optimizations. These include:

- Core optimizations, which includes various types of control flow transformation, and architectural specific peephole optimizations.
- SSA based scalar optimizations
- ILP optimizations for superscalars
- ILP optimizations for VLIW/EPIC architectures
- GC safety analysis

20.3 Basic Concepts

Basic concepts in MLRISC are:

- Instructions[47] – the instruction set of the target architecture.

[41] url: `mltree.html`
[42] url: `instrsel.html`
[43] url: `ra.html`
[44] url: `asm.html`
[45] url: `span-dep.html`
[46] url: `mc.html`
[47] url: `instructions.html`

- Cells[48] – which describes registers, memory and other mutable resources in the machine.
- Regions[49] – a client defined abstract type used to represent aliasing information available from the front-end.
- Constants[50] – a client defined place holder used to represent constants whose values are unknown in the front-end.
- Pseudo Ops[51] – a client defined
- Annotations[52] – this is a generic mechinism for propagating information in the MLRISC sstem. The client may attach arbitrary annotation of various granularity to MLRISC's program representation, which can then be propagated to later phases. These can be information related to profiling frequency, dependence, comments, and/or types. The same mechanism is also used to propagate analysis information one optimization phase to another.
- Instruction Streams[53] – an abstraction for describing a stream of instructions. Instruction streams are used to connect modules such as instruction selection, assembler, machine code emitter, and control flow graph builder.
- Regmap[54] – a mapping between registers names. MLRISC register allocators represent the result of register allocation as a regmap.
- Labels[55] – a type representing symbolic labels.
- Label Expressions[56] – a type representing constant expressions involving symbolic labels.

20.4 How Things Are Fit Together

MLRISC uses two different program representations, clusters and MLRISC IR.

- Cluster[57] is light-weight representation that is used when only the most basic optimizations are required.
- MLRISC IR[58] is more heavy-weight representation that is built from the MLRISC graph library[59] and the MLRISC compiler graph library[60]. MLRISC IR allows more complex transformations and analysis of the program graph.

Conversion modules between the two representations are provided.

In general MLRISC optimization phases are transformations applied on one of these representations. Optimizations may be chained together to form a compiler backend. For example, a minimal backend consists of

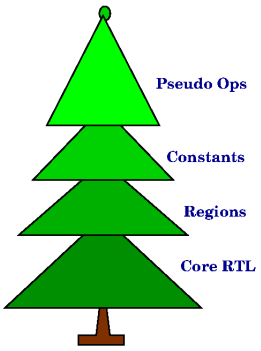
[48] url: `cells.html`
[49] url: `regions.html`
[50] url: `constants.html`
[51] url: `pseudo-ops.html`
[52] url: `annotations.html`
[53] url: `streams.html`
[54] url: `regmap.html`
[55] url: `labels.html`
[56] url: `labelexp.html`
[57] url: `cluster.html`
[58] url: `mlrisc-ir.html`
[59] url: `graphs.html`
[60] url: `compiler-graphs.html`

- the instruction selection module, which translates MLTree[61] into target instructions,
- the flowgraph builder, which conversts a stream of target instructions into a cluster,
- the register allocator, which performs register allocation, and
- the assembly code emitter, which generates assembly output

[61] url: `mltree.html`

21 The MLTREE Language

MLTree is the register transfer language used in the MLRISC system. It serves two important purposes:



MLTree

1. As an intermediate representation for a compiler front-end to talk to the MLRISC system,
2. As specifications for instruction semantics

The latter is needed for optimizations which require precise knowledge of such; for example, algebraic simplification and constant folding.

MLTree is a low-level *typed* language: all operations are typed by its width or precision. Operations on floating point, integer, and condition code are also segregated, to prevent accidental misuse. MLTree is also *tree-oriented* so that it is possible to write efficient MLTree transformation routines that uses SML pattern matching.

Here are a few examples of MLTree statements.

```
MV(32,t,
   ADDT(32,
     MULT(32,REG(32,b),REG(32,b)),
     MULT(32,
       MULT(32,LI(4),REG(32,a)),REG(32,c))))
```

computes `t := b*b + 4*a*c`, all in 32-bit precision and overflow trap enabled; while

```
MV(32,t,
   ADD(32,
     CVTI2I(32,SIGN_EXTEND,8,
       LOAD(8,
         ADD(32,REG(32,a),REG(32,i))))))
```

loads the byte in address `a+i` and sign extend it to a 32-bit value.

The statement

```
IF([],CMP(64,GE,REG(64,a),LI 0),
      MV(64, t, REG(64, a)),
      MV(64, t, NEG(64, REG(64, a)))
  )
```

in more traditional form means:

```
if a >= 0 then
   t := a
else
   t := -a
```

This example can be also expressed in a few different ways:

1. With the conditional move construct described in Section 21.2.2:

```
MV(64, t,
   COND(CMP(64, GE, REG(64, a)),
        REG(64, a),
        NEG(64, REG(64, a))))
```

2. With explicit branching using the conditional branch construct `BCC`:

```
MV(64, t, REG(64, a));
BCC([], CMP(64, GE, REG(64, a)), L1);
MV(64, t, NEG(64, REG(64, a)));
DEFINE L1;
```

21.1 The Definitions

MLTree is defined in the signature `MLTREE`[62] and the functor `MLTreeF`[63]

The functor `MLTreeF` is parameterized in terms of the label expression type, the client supplied region datatype, the instruction stream type, and the client defined MLTree extensions.

```
functor MLTreeF
  (structure LabelExp : LABELEXP[64]
   structure Region : REGION[65]
   structure Stream : INSTRUCTION_STREAM[66]
   structure Extension : MLTREE_EXTENSION[67]
  ) : MLTREE
```

[62] **file:** `mltree/mltree.sig`
[63] **file:** `mltree/mltree.sml`
[64] url: `labelexp.html`
[65] url: `regions.html`
[66] url: `streams.html`
[67] **file:** `mltree/mltree-extension.sig`

21.1.1 Basic Types

The basic types in MLTree are statements (`stm`) integer expressions (`rexp`), floating point expression (`fexp`), and conditional expressions (`ccexp`). Statements are evaluated for their effects, while expressions are evaluated for their value. (Some expressions could also have trapping effects. The semantics of traps are unspecified.) These types are parameterized by an extension type, which we can use to extend the set of MLTree operators. How this is used is described in Section 22.

References to registers are represented internally as integers, and are denoted as the type `reg`. In addition, we use the types `src` and `dst` as abbreviations for source and destination registers.

```
type reg = int
type src = reg
type dst = reg
```

All operators on MLTree are *typed* by the number of bits that they work on. For example, 32-bit addition between `a` and `b` is written as `ADD(32,a,b)`, while 64-bit addition between the same is written as `ADD(64,a,b)`. Floating point operations are denoted in the same manner. For example, IEEE single-precision floating point add is written as `FADD(32,a,b)`, while the same in double-precision is written as `FADD(64,a,b)`

Note that these types are low level. Higher level distinctions such as signed and unsigned integer value, are not distinguished by the type. Instead, operators are usually partitioned into signed and unsigned versions, and it is legal (and often useful!) to mix signed and unsigned operators in an expression.

Currently, we don't provide a direct way to specify non-IEEE floating point together with IEEE floating point arithmetic. If this distinction is needed then it can be encoded using the extension mechanism described in Section 22.

We use the types `ty` and `fty` to stand for the number of bits in integer and floating point operations.

```
type ty  = int
type fty = int
```

21.1.2 The Basis

The signature `MLTREE_BASIS`[68] defines the basic helper types used in the MLTREE signature.

```
signature MLTREE_BASIS =
sig

  datatype cond = LT | LTU | LE | LEU | EQ | NE | GE | GEU | GT | GTU

  datatype fcond =
     ? | !<=> | == | ?= | !<> | !?>= | < | ?< | !>= | !?> |
     <= | ?<= | !> | !?<= | > | ?> | !<= | !?< | >= | ?>= |
     !< | !?= | <> | != | !? | <=> | ?<>

  datatype ext = SIGN_EXTEND | ZERO_EXTEND

  datatype rounding_mode = TO_NEAREST | TO_NEGINF | TO_POSINF | TO_ZERO
```

[68] **file:** `mltree/mltree-basis.sig`

```
  type ty = int
  type fty = int

end
```

The most important of these are the types `cond` and `fcond`, which represent the set of integer and floating point comparisions. These types can be combined with the comparison constructors `CMP` and `FCMP` to form integer and floating point comparisions.

Operator	Comparison
`LT`	Signed less than
`LTU`	Unsigned less than
`LE`	Signed less than or equal
`LEU`	Unsigned less than or equal
`EQ`	Equal
`NE`	Not equal
`GE`	Signed greater than or equal
`GEU`	Unsigned greater than or equal
`GT`	Signed greater than
`GTU`	Unsigned greater than

Floating point comparisons can be "decoded" as follows. In IEEE floating point, there are four different basic comparisons tests that we can performed given two numbers a and y:

$a < b$ Is a less than b?

$a = b$ Is a equal to b?

$a > b$ Is a greater than to b?

$a?b$ Are a and b unordered (incomparable)?

Comparisons can be joined together. For example, given two double-precision floating point expressions a and b, the expression `FCMP(64,<=>,a,b)` asks whether a is less than, equal to or greater than b, i.e. whether a and b are comparable. The special symbol `!` negates the meaning the of comparison. For example, `FCMP(64,!>=,a,b)` means testing whether a is less than or incomparable with b.

21.2 Integer Expressions

A reference to the ith integer register with an n-bit value is written as `REG(`n`,`i`)`. The operators `LI`, `LI32`, and `LABEL`, `CONST` are used to represent constant expressions of various forms. The sizes of these constants are inferred from context.

```
  REG   : ty * reg -> rexp
  LI    : int -> rexp
  LI32  : Word32.word -> rexp
  LABEL : LabelExp.labexp -> rexp
  CONST : Constant.const -> rexp
```

The following figure lists all the basic integer operators and their intuitive meanings. All operators except `NOTB, NEG, NEGT` are binary and have the type

```
  ty * rexp * rexp -> rexp
```

The operators NOTB, NEG, NEGT have the type

```
ty * rexp -> rexp
```

ADD	Twos complement addition
NEG	negation
SUB	Twos complement subtraction
MULS	Signed multiplication
DIVS	Signed division, round to zero (nontrapping)
QUOTS	Signed division, round to negative infinity (nontrapping)
REMS	Signed remainder (???)
MULU	Unsigned multiplication
DIVU	Unsigned division
REMU	Unsigned remainder
NEGT	signed negation, trap on overflow
ADDT	Signed addition, trap on overflow
SUBT	Signed subtraction, trap on overflow
MULT	Signed multiplication, trap on overflow
DIVT	Signed division, round to zero, trap on overflow or division by zero
QUOTT	Signed division, round to negative infinity, trap on overflow or division by zero
REMT	Signed remainder, trap on division by zero
ANDB	bitwise and
ORB	bitwise or
XORB	bitwise exclusive or
NOTB	ones complement
SRA	arithmetic right shift
SRL	logical right shift
SLL	logical left shift

21.2.1 Sign and Zero Extension

Sign extension and zero extension are written using the operator CVTI2I. CVTI2I(m,SIGN_EXTEND,n,e) sign extends the n-bit value e to an m-bit value, i.e. the $n-1$th bit is of e is treated as the sign bit. Similarly, CVTI2I(m,ZERO_EXTEND,n,e) zero extends an n-bit value to an m-bit value. If $m \leq n$, then CVTI2I(m,SIGN_EXTEND,n,e) = CVTI2I(m,ZERO_EXTEND,n,e).

```
datatype ext = SIGN_EXTEND | ZERO_EXTEND
CVTI2I : ty * ext * ty * rexp -> rexp
```

21.2.2 Conditional Move

Most new superscalar architectures incorporate conditional move instructions in their ISAs. Modern VLIW architectures also directly support full predication. Since branching (especially with data dependent branches) can introduce extra latencies in highly pipelined architectures, condtional moves should be used in place of short branch sequences. MLTree provide a conditional move instruction COND, to make it possible to directly express conditional moves without using branches.

```
COND : ty * ccexp * rexp * rexp -> rexp
```

Semantically, `COND(`*ty*`,`*cc*`,`a`,`b`)` means to evaluate *cc*, and if *cc* evaluates to true then the value of the entire expression is a; otherwise the value is b. Note that a and b are allowed to be *eagerly* evaluated. In fact, we are allowed to evaluate to *both* branches, one branch, or neither [69].

Various idioms of the `COND` form are useful for expressing common constructs in many programming languages. For example, MLTree does not provide a primitive construct for converting an integer value `x` to a boolean value (0 or 1). But using `COND`, this is expressible as `COND(32,CMP(32,NE,x,LI 0),LI 1,LI 0)`. SML/NJ represents the boolean values true and false as machine integers 3 and 1 respectively. To convert a boolean condition e into an ML boolean value, we can use

```
COND(32,e,LI 3,LI 1)
```

Common C idioms can be easily mapped into the `COND` form. For example,

- `if (e1) x = y` translates into `MV(32,x,COND(32,e1,REG(32,y),REG(32,x)))`
- ```
  x = e1;
  if (e2) x = y
  ```

  translates into `MV(32,x,COND(32,e2,REG(32,y),e1))`
- `x = e1 == e2` translates into `MV(32,x,COND(32,CMP(32,EQ,e1,e2),LI 1,LI 0)`
- `x = ! e` translates into `MV(32,x,COND(32,CMP(32,NE,e,LI 0),LI 1,LI 0)`
- `x = e ? y : z` translates into `MV(32,x,COND(32,e,REG(32,y),REG(32,z)))`, and
- `x = y < z ? y : z` translates into

```
MV(32,x,
    COND(32,
       CMP(32,LT,REG(32,y),REG(32,z)),
          REG(32,y),REG(32,z)))
```

In general, the `COND` form should be used in place of MLTree's branching constructs whenever possible, since the former is usually highly optimized in various MLRISC backends.

21.2.3 Integer Loads

Integer loads are written using the constructor `LOAD`.

```
LOAD  : ty * rexp * Region.region -> rexp
```

The client is required to specify a region[70] that serves as aliasing information for the load.

[69] When possible.
[70] url: `regions.html`
```

21.2.4 Miscellaneous Integer Operators

An expression of the LET(s,e) evaluates the statement s for its effect, and then return the value of expression e.

```
LET  : stm * rexp -> rexp
```

Since the order of evaluation is MLTree operators are *unspecified* the use of this operator should be severely restricted to only *side-effect*-free forms.

21.3 Floating Point Expressions

Floating registers are referenced using the term FREG. The ith floating point register with type n is written as FREG(n,i).

```
FREG   : fty * src -> fexp
```

Built-in floating point operations include addition (FADD), subtraction (FSUB), multiplication (FMUL), division (FDIV), absolute value (FABS), negation (FNEG) and square root (FSQRT).

```
FADD  : fty * fexp * fexp -> fexp
FSUB  : fty * fexp * fexp  -> fexp
FMUL  : fty * fexp * fexp -> fexp
FDIV  : fty * fexp * fexp -> fexp
FABS  : fty * fexp -> fexp
FNEG  : fty * fexp -> fexp
FSQRT : fty * fexp -> fexp
```

A special operator is provided for manipulating signs. To combine the sign of a with the magnitude of b, we can write FCOPYSIGN(a,b)[71].

```
FCOPYSIGN : fty * fexp * fexp -> fexp
```

To convert an n-bit signed integer e into an m-bit floating point value, we can write CVTI2F(m,n,e)[72].

```
CVTI2F : fty * ty * rexp -> fexp
```

Similarly, to convert an n-bit floating point value e to an m-bit floating point value, we can write CVTF2F(m,n,e)[73].

```
CVTF2F : fty * fty * -> fexp


datatype rounding_mode = TO_NEAREST | TO_NEGINF | TO_POSINF | TO_ZERO
CVTF2I : ty * rounding_mode * fty * fexp -> rexp


FLOAD : fty * rexp * Region.region -> fexp
```

[71] What should happen if a or b is nan?
[72] What happen to unsigned integers?
[73] What is the rounding semantics?

21.4 Condition Expressions

Unlike languages like C, MLTree makes the distinction between condition expressions and integer expressions. This distinction is necessary for two purposes:

- It clarifies the proper meaning intended in a program, and
- It makes to possible for a MLRISC backend to map condition expressions efficiently onto various machine architectures with different condition code models. For example, architectures like the Intel x86, Sparc V8, and PowerPC contains dedicated condition code registers, which are read from and written to by branching and comparison instructions. On the other hand, architectures such as the Texas Instrument C6, PA RISC, Sparc V9, and Alpha does not include dedicated condition code registers. Conditional code registers in these architectures can be simulated by integer registers.

A conditional code register bit can be referenced using the constructors `CC` and `FCC`. Note that the *condition* must be specified together with the condition code register.

```
CC   : Basis.cond * src -> ccexp
FCC  : Basis.fcond * src -> ccexp
```

For example, to test the `Z` bit of the `%psr` register on the Sparc architecture, we can used `CC(EQ,SparcCells.psr)`.

The comparison operators `CMP` and `FCMP` performs integer and floating point tests. Both of these are *typed* by the precision in which the test must be performed under.

```
CMP  : ty * Basis.cond * rexp * rexp -> ccexp
FCMP : fty * Basis.fcond * fexp * fexp -> ccexp
```

Condition code expressions may be combined with the following logical connectives, which have the obvious meanings.

```
TRUE  : ccexp
FALSE : ccexp
NOT   : ccexp -> ccexp
AND   : ccexp * ccexp -> ccexp
OR    : ccexp * ccexp -> ccexp
XOR   : ccexp * ccexp -> ccexp
```

21.5 Statements

Statement forms in MLTree includes assignments, parallel copies, jumps and condition branches, calls and returns, stores, sequencing, and annotation.

21.5.1 Assignments

Assignments are segregated among the integer, floating point and conditional code types. In addition, all assignments are *typed* by the precision of destination register.

```
MV   : ty * dst * rexp -> stm
FMV  : fty * dst * fexp -> stm
CCMV : dst * ccexp -> stm
```

21.5.2 Parallel Copies

Special forms are provided for parallel copies for integer and floating point registers. It is important to emphasize that the semantics is that all assignments are performed in parallel.

```
COPY  : ty * dst list * src list -> stm
FCOPY : fty * dst list * src list -> stm
```

21.5.3 Jumps and Conditional Branches

Jumps and conditional branches in MLTree take two additional set of annotations. The first represents the *control flow* and is denoted by the type `controlflow`. The second represent *control-dependence* and *anti-control-dependence* and is denoted by the type `ctrl`.

```
type controlflow = Label.label list
type ctrl = reg list
```

Control flow annotation is simply a list of labels, which represents the set of possible targets of the associated jump. Control dependence annotations attached to a branch or jump instruction represents the new definition of *pseudo control dependence predicates*. These predicates have no associated dynamic semantics; rather they are used to constraint the set of potential code motion in an optimizer (more on this later).

The primitive jumps and conditional branch forms are represented by the constructors `JMP`, `BCC`.

```
JMP : ctrl * rexp * controlflow  -> stm
BCC : ctrl * ccexp * Label.label -> stm
```

In addition to `JMP` and `BCC`, there is a *structured* if/then/else statement.

```
IF  : ctrl * ccexp * stm * stm -> stm
```

Semantically, `IF`(c, x, y, z) is identical to

```
BCC(c, x, L1)
z
JMP([], L2)
DEFINE L1
y
DEFINE L2
```

where `L1` and `L2` are new labels, as expected.

Here's an example of how control dependence predicates are used. Consider the following MLTree statement:

```
IF([p], CMP(32, NE, REG(32, a), LI 0),
     MV(32, b, PRED(LOAD(32, m, ...)), p),
     MV(32, b, LOAD(32, n, ...)))
```

In the first alternative of the `IF`, the `LOAD` expression is constrainted by the control dependence predicate `p` defined in the `IF`, using the predicate constructor `PRED`. These states that the load is *control dependent* on the test of the branch, and thus it may not be legally hoisted above the branch without potentially

violating the semantics of the program. For example, semantics violation may happen if the value of `m` and `a` is corrolated, and whenever `a` = 0, the address in `m` is not a legal address.

Note that on architectures with speculative loads, the control dependence information can be used to guide the transformation of control dependent loads into speculative loads.

Now in constrast, the `LOAD` in the second alternative is not control dependent on the control dependent predicate `p`, and thus it is safe and legal to hoist the load above the test, as in

```
MV(32, b, LOAD(32, n, ...));
IF([p], CMP(32, NE, REG(32, a), LI 0),
     MV(32, b, PRED(LOAD(32, m, ...)), p),
     SEQ []
  )
```

Of course, such transformation is only performed if the optimizer phases think that it can benefit performance. Thus the control dependence information does *not* directly specify any transformations, but it is rather used to indicate when aggressive code motions are legal and safe.

21.5.4 Calls and Returns

Calls and returns in MLTree are specified using the constructors `CALL` and `RET`, which have the following types.

```
CALL : rexp * controlflow * mlrisc * mlrisc *
       ctrl * ctrl * Region.region -> stm
RET  : ctrl * controlflow -> stm
```

The `CALL` form is particularly complex, and require some explanation. Basically the seven parameters are, in order:

address of the called routine.

control flow annotation for this call. This information specifies the potential targets of this call instruction. Currently this information is ignored but will be useful for interprocedural optimizations in the future.

definition and use These lists specify the list of potential definition and uses during the execution of the call. Definitions and uses are represented as the type `mlrisc` list. The contructors for this type is:

```
CCR : ccexp -> mlrisc
GPR : rexp -> mlrisc
FPR : fexp -> mlrisc
```

definition of control and anti-control dependence These two lists specifies definitions of control and anti-control dependence.

region annotation for the call, which summarizes the set of potential memory references during execution of the call.

The matching return statement constructor `RET` has two arguments. These are:

anti-control dependence This parameter represents the set of anti-control dependence predicates defined by the return statement.

control flow This parameter specifies the set of matching procedure entry points of this return. For example, suppose we have a procedure with entry points `f` and `f'`. Then the MLTree statements

```
f:   ...
     JMP L1
f':  ...
L1:  ...
     RET ([], [f, f'])
```

can be used to specify that the return is either from the entries `f` or `f'`.

21.5.5 Stores

Stores to integer and floating points are specified using the constructors `STORE` and `FSTORE`.

```
STORE  : ty * rexp * rexp * Region.region -> stm
FSTORE : fty * rexp * fexp * Region.region -> stm
```

The general form is

`STORE`($width$, $address$, $data$, $region$)

Stores for condition codes are not provided.

21.5.6 Miscelleneous Statements

Other useful statement forms of MLTree are for sequencing (`SEQ`), defining a local label (`DEFINE`).

```
SEQ    : stm list -> stm
DEFINE : Label.label -> stm
```

The constructor `DEFINE L` has the same meaning as executing the method `defineLabel L` in the stream interface[74].

21.6 Annotations

Annotations[75] are used as the generic mechanism for exchanging information between different phases of the MLRISC system, and between a compiler front end and the MLRISC back end. The following constructors can be used to annotate a MLTree term with an annotation:

```
MARK : rexp * Annotations.annotation -> rexp
FMARK : fexp * Annotations.annotation -> fexp
CCMARK : ccexp * Annotations.annotation -> ccexp
ANNOTATION : stm * Annotations.annotation -> stm
```

[74] url: `stream.html`
[75] url: `annotations.html`

22 MLTree Extensions

Pattern matching over the MLTREE intermediate representation may not be sufficient to provide access to all the registers or operations provided on a specific architecture. MLTREE extensions is a method of extending the MLTREE intermediate language so that it is a better match for the target architecture.

22.1 Why Extensions

Pattern matching over the MLTREE intermediate representation may not be sufficient to provide access to all the registers or operations provided on a specific architecture. MLTREE extensions is a method of extending the MLTREE intermediate language so that it is a better match for the target architecture.

For example there may be special registers to support the increment-and-test operation on loop indices, or support for complex mathematical functions such as square root, or access to hardware specific registers such as the current register window pointer on the SPARC architecture. It is not usually possible to write expression trees that would directly generate these instructions. Some complex operations can be generated by performing a peephole optimization over simpler instructions, however this is not always the most convenient or simple thing to do.

22.2 Cyclic Dependency

The easiest way to provide extensions is to parameterize the MLTREE interface with types that extend the various kinds of trees. Thus if the type `sext` represented statement extensions, we might define MLTREE statement trees as :

```
datatype stm
  = ...
  | SEXT of sext * mlrisc list * stm list

and mlrisc = GPR of rexp | FPR of fexp | CCR of ccexp
```

where the constructor `SEXT` applies the extension to a list of arguments. This approach is unsatisfactory in several ways, for example, if one wanted to extend MLTREEs with for-loops, then the following could be generated:

```
SEXT(FORLOOP, [GPR from, GPR to, GPR step], body)
```

First, the loop arguments have to be wrapped up in `GPR` and there is little self documentation on the order of elements that are arguments to the for-loop. It would be better to be able to write something like:

```
SEXT(FORLOOP{from=f, to=t, step=s, body=b})
```

Where `f`, `t`, and `s` are `rexp` trees representing the loop index start, end, and step size; `b` is a stm list representing the body of the loop. Unfortunately, there is a cyclic dependency as MLTREEs are defined in terms of `sext`, and `sext` is defined in terms of MLTREEs. The usual way to deal with cyclic dependencies is to use polymorphic type variables.

22.3 MLTREE EXTENSION

The statement extension type `sext`, is now a type constructor with arity four, i.e. `('s, 'r, 'f, 'c) sx` where `sx` is used instead of `sext`, and *'s*, *'r*, *'f*, and *'c* represents MLTREE statement expressions, register expressions, floating point expressions, and condition code expressions. Thus the for-loop extension could be declared using something like:

```
datatype ('s,'r,'f,'c) sx
  = FORLOOP of {from: 'r, to: 'r, step: 'r, body: 's}
```

and the MLTREE interface is defined as:

```
signature MLTREE = sig
  type ('s, 'r, 'f, 'c) sx

  datatype stm =
    = ...
    | SEXT of sext

 withtype sext = (stm, rexp, fexp, cexp) sx
end
```

where `sext` is the user defined statement extension but the type variables have been instantiated to the final form the the MLTREE `stm`, `rexp`, `fexp`, and `cexp` components.

22.4 Compilation

There are dedicated modules that perform pattern matching over MLTREEs and emit native instructions, and similar modules must be written for extensions. However, the same kinds of choices used in regular MLTREE patterns must be repeated for extensions. For example, one may define an extension for the Intel IA32 of the form:

```
datatype ('s,'r,'f,'c) sx = PUSHL of 'r | POPL of 'r | ...
```

that translate directly to the Intel push and pop instructions; the operands in each case are either memory locations or registers, but immediates are allowed in the case of `PUSHL`. Considerable effort has been invested into pattern matching the extensive set of addressing modes for the Intel architecture, and one would like to reuse this when compiling extensions. The pattern matching functions are exposed by a set of functions exported from the instruction selection module, and provided in the MLTREE interface. They are:

```
struture I : INSTRUCTIONS
datatype reducer =
  REDUCER of {
    reduceRexp    : rexp -> reg,
    reduceFexp    : fexp -> reg,
    reduceCCexp   : ccexp -> reg,
    reduceStm     : stm * an list -> unit,
    operand       : rexp -> I.operand,
    reduceOperand : I.operand -> reg,
```

```
    addressOf     : rexp -> I.addressing_mode,
    emit          : I.instr * an list -> unit,
    instrStream   : (I.instr, I.regmap, I.cellset) stream,
    mltreeStream  : (stm, I.regmap, mlrisc list) stream
  }
```

where `I` is the native instruction set.

`reduceRexp` : reduces an MLTREE `rexp` to a register, and similarly for `reduceFexp` and `reduceCCexp`.

`reduceStm` : reduces an MLTREE `stm` to a set of instructions that implement the set of statements.

`operand` : reduced an MLTREE `rexp` into an instruction operand — usually an immediate or memory address.

`operand` : moves a native operand into a register.

`addressOf` : reduces an MLTREE `rexp` into a memory address.

`emit` : emits an instruction together with an annotation.

`instrStream` : is the native instruction output stream, and

`mltreeStream` : is the MLTREE output stream.

Each extension must provide a function `compileSext` that compiles a statement extension into native instructions. In the `MLTREE_EXTENSION_COMP` interface we have:

```
val compileSext: reducer -> stm: MLTREE.sexp, an:MLTREE.an list -> unit
```

The use of extensions must follow a special structure.

1. A module defining the extension type using a type constructor of arity four. Let us call this structure `ExtTy` and must match the `MLTREE_EXTENSION` interface.
2. The extension module must be used to specialize MLTREEs.
3. A module that describes how to compile the extension must be created, and must match the `MLTREE_EXTENSION_COMP` interace. This module will typically be functorized over the MLTREE interface. Let us call the result of applying the functor, `ExtComp`.
4. The extension compiler must be passed as a parameter to the instruction selection module that will invoke it whenever an extension is seen.

22.5 Multiple Extensions

Multiple extensions are handled in a similar fashion, except that the extension type used to specialize MLTREEs is a tagged union of the individual extensions. The functor to compile the extension dispatches to the compilation modules for the individual extensions.

22.6 Example

Suppose you are in the process of writing a compiler for a digital signal processing(*DSP*) programming language using the MLRISC framework. This wonderful language that you are developing allows the programmer to specify high level looping and iteration, and aggregation constructs that are common in DSP applications. Furthermore, since saturated and fixed point arithmetic are common constructs in DSP applications, the language and consequently the compiler should directly support these operators. For simplicity, we would like to have a unified intermediate representation that can be used to directly represent high level constructs in our language, and low level constructs that are already present in MLTree. Since, MLTree does not directly support these constructs, it seems that it is not possible to use MLRISC for such a compiler infrastructure without substantial rewrite of the core components.

Let us suppose that for illustration that we would like to implement high level looping and aggregation constructs such as

```
for i := lower bound ... upper bound
    body
x := sum{i := lower bound ... upper bound} expression
```

together with saturated arithmetic mentioned above.

Here is a first attempt:

```
structure DSPMLTreeExtension
struct
   structure Basis = MLTreeBasis
   datatype ('s,'r,'f,'c) sx =
      FOR of Basis.var * 'r * 'r * 's
   and ('s,'r,'f,'c) rx =
      SUM of Basis.var * 'r * 'r * 'r
    | SADD of 'r * 'r
    | SSUB of 'r * 'r
    | SMUL of 'r * 'r
    | SDIV of 'r * 'r
   type ('s,'r,'f,'c) fx = unit
   type ('s,'r,'f,'c) ccx = unit
end
structure DSPMLTree : MLTreeF
    (structure Extension = DSPMLTreeExtension
     ...
    )
```

In the above signature, we have defined two new datatypes `sx` and `rx` that are used for representing the DSP statement and integer expression extensions. Integer expression extensions include the high level sum construct, and the low levels saturated arithmetic operators. The recursive type definition is necessary to "inject" these new constructors into the basic MLTree definition.

The following is an example of how these new constructors that we have defined can be used. Suppose the source program in our DSP language is:

```
for i := a ... b
{  s := sadd(s, table[i]);
}
```

where sadd is the saturated add operator. For simplicity, let us also assume that all operations and addresses are in 32-bits. Then the translation of the above into our extended DSP-MLTree could be:

```
EXT(FOR(i, REG(32, a), REG(32, b),
        MV(32, s, REXT(32, SADD(REG(32, s),
             LOAD(32,
                 ADD(32, REG(32, table),
                     SLL(32, REG(32, i), LI 2)),
                  region))))
       ))
```

One potential short coming of our DSP extension to MLTree is that the extension does not allow any further extensions. This restriction may be entirely satisfactory if DSP-MLTree is only used in your compiler applications and no where else. However, if DSP-MLTree is intended to be an extension library for MLRISC, then we must build in the flexibility for extension. This can be done in the same way as in the base MLTree definition, like this:

```
functor ExtensibleDSPMLTreeExtension
  (Extension : MLTREE_EXTENSION[76]) =
struct
   structure Basis = MLTreeBasis
   structure Extension = Extension
   datatype ('s,'r,'f,'c) sx =
      FOR of Basis.var * 'r * 'r * 's
    | EXT of ('s,'r,'f,'c) Extension.sx
   and ('s,'r,'f,'c) rx =
      SUM of Basis.var * 'r * 'r * 'r
    | SADD of 'r * 'r
    | SSUB of 'r * 'r
    | SMUL of 'r * 'r
    | SDIV of 'r * 'r
    | REXT of ('s,'r,'f,'c) Extension.rx
   withtype
        ('s,'r,'f,'c) fx   = ('s,'r,'f,'c) Extension.fx
   and  ('s,'r,'f,'c) ccx  = ('s,'r,'f,'c) Extension.ccx
end
```

As in MLTREE, we provide two new extension constructors EXT and REXT in the definition of DSP_MLTREE, which can be used to further enhance the extended MLTREE language.

[76] **file:** mltree/mltree-extension.sig

23 MLTree Utilities

The MLRISC system contains numerous utilities for working with MLTree datatypes. Some of the following utilizes are also useful for clients use:

MLTreeUtils implements basic hashing, equality and pretty printing functions,

MLTreeFold implements a fold function over the MLTree datatypes,

MLTreeRewrite implements a generic rewriting engine,

MLTreeSimplify implements a simplifier that performs algebraic simplification and constant folding.

23.0.1 Hashing, Equality, Pretty Printing

The functor `MLTreeUtils`[77] provides the basic utilities for hashing an MLTree term, comparing two MLTree terms for equality and pretty printing. The hashing and comparision functions are useful for building hash tables using MLTree datatype as keys. The signature of the functor is:

```
signature MLTREE_UTILS[78] =
sig
   structure T : MLTREE

   (*
    * Hashing
    *)
   val hashStm   : T.stm -> word
   val hashRexp  : T.rexp -> word
   val hashFexp  : T.fexp -> word
   val hashCCexp : T.ccexp -> word

   (*
    * Equality
    *)
   val eqStm     : T.stm * T.stm -> bool
   val eqRexp    : T.rexp * T.rexp -> bool
   val eqFexp    : T.fexp * T.fexp -> bool
   val eqCCexp   : T.ccexp * T.ccexp -> bool
   val eqMlriscs : T.mlrisc list * T.mlrisc list -> bool

   (*
    * Pretty printing
    *)
   val show : (string list * string list) -> T.printer

   val stmToString   : T.stm -> string
   val rexpToString  : T.rexp -> string
```

[77] **file:** `mltree/mltree-utils.sml`
[78] **file:** `mltree/mltree-utils.sig`

```
   val fexpToString  : T.fexp -> string
   val ccexpToString : T.ccexp -> string

end
functor MLTreeUtils[79]
  (structure T : MLTREE
   (* Hashing extensions *)
   val hashSext  : T.hasher -> T.sext -> word
   val hashRext  : T.hasher -> T.rext -> word
   val hashFext  : T.hasher -> T.fext -> word
   val hashCCext : T.hasher -> T.ccext -> word

   (* Equality extensions *)
   val eqSext  : T.equality -> T.sext * T.sext -> bool
   val eqRext  : T.equality -> T.rext * T.rext -> bool
   val eqFext  : T.equality -> T.fext * T.fext -> bool
   val eqCCext : T.equality -> T.ccext * T.ccext -> bool

   (* Pretty printing extensions *)
   val showSext  : T.printer -> T.sext -> string
   val showRext  : T.printer -> T.ty * T.rext -> string
   val showFext  : T.printer -> T.fty * T.fext -> string
   val showCCext : T.printer -> T.ty * T.ccext -> string
  ) : MLTREE_UTILS =
```

The types `hasher`, `equality`, and `printer` represent functions for hashing, equality and pretty printing. These are defined as:

```
   type hasher =
      {stm    : T.stm -> word,
       rexp   : T.rexp -> word,
       fexp   : T.fexp -> word,
       ccexp  : T.ccexp -> word
      }

   type equality =
      { stm    : T.stm * T.stm -> bool,
        rexp   : T.rexp * T.rexp -> bool,
        fexp   : T.fexp * T.fexp -> bool,
        ccexp  : T.ccexp * T.ccexp -> bool
      }
   type printer =
      { stm    : T.stm -> string,
        rexp   : T.rexp -> string,
        fexp   : T.fexp -> string,
        ccexp  : T.ccexp -> string,
        dstReg : T.ty * T.var -> string,
```

[79] **file:** `mltree/mltree-utils.sml`

```
    srcReg : T.ty * T.var -> string
  }
```

For example, to instantiate a `Utils` module for our `DSPMLTree`, we can write:

```
structure U = MLTreeUtils
  (structure T = DSPMLTree
   fun hashSext {stm, rexp, fexp, ccexp} (FOR(i, a, b, s)) =
        Word.fromIntX i + rexp a + rexp b + stm s
   and hashRext {stm, rexp, fexp, ccexp} e =
       (case e of
          SUM(i,a,b,c) => Word.fromIntX i + rexp a + rexp b + rexp c
        | SADD(a,b) => rexp a + rexp b
        | SSUB(a,b) => 0w12 + rexp a + rexp b
        | SMUL(a,b) => 0w123 + rexp a + rexp b
        | SDIV(a,b) => 0w1245 + rexp a + rexp b
       )
   fun hashFext _ _ = 0w0
   fun hashCCext _ _ = 0w0
   fun eqSext {stm, rexp, fexp, ccexp}
     (FOR(i, a, b, s), FOR(i', a', b', s')) =
        i=i' andalso rexp(a,a') andalso rexp(b,b') andalso stm(s,s')
   fun eqRext {stm, rexp, fexp, ccexp} (e,e') =
    (case (e,e') of
       (SUM(i,a,b,c),SUM(i',a',b',c')) =>
         i=i' andalso rexp(a,a') andalso rexp(b,b') andalso stm(c,c')
     | (SADD(a,b),SADD(a',b')) => rexp(a,a') andalso rexp(b,b')
     | (SSUB(a,b),SSUB(a',b')) => rexp(a,a') andalso rexp(b,b')
     | (SMUL(a,b),SMUL(a',b')) => rexp(a,a') andalso rexp(b,b')
     | (SDIV(a,b),SDIV(a',b')) => rexp(a,a') andalso rexp(b,b')
     | _ => false
    )
   fun eqFext _ _ = true
   fun eqCCext _ _ = true

   fun showSext {stm, rexp, fexp, ccexp, dstReg, srcReg}
         (FOR(i, a, b, s)) =
       "for("^dstReg i^":="^rexp a^".."^rexp b^")"^stm s
   fun ty t = "."^Int.toString t
   fun showRext {stm, rexp, fexp, ccexp, dstReg, srcReg} e =
        (case (t,e) of
          SUM(i,a,b,c) =>
           "sum"^ty t^"("^dstReg i^":="^rexp a^".."^rexp b^")"^rexp c
        | SADD(a,b) => "sadd"^ty t^"("rexp a^","^rexp b^")"
        | SSUB(a,b) => "ssub"^ty t^"("rexp a^","^rexp b^")"
        | SMUL(a,b) => "smul"^ty t^"("rexp a^","^rexp b^")"
        | SDIV(a,b) => "sdiv"^ty t^"("rexp a^","^rexp b^")"
        )
   fun showFext _ _ = ""
```

```
      fun showCCext _ _ = ""
     )
```

23.0.2 MLTree Fold

The functor `MLTreeFold`[80] provides the basic functionality for implementing various forms of aggregation function over the MLTree datatypes. Its signature is

```
signature MLTREE_FOLD[81] =
sig
   structure T : MLTREE

   val fold : 'b folder -> 'b folder
end
functor MLTreeFold[82]
  (structure T : MLTREE
   (* Extension mechnism *)
   val sext  : 'b T.folder -> T.sext * 'b -> 'b
   val rext  : 'b T.folder -> T.ty * T.rext * 'b -> 'b
   val fext  : 'b T.folder -> T.fty * T.fext * 'b -> 'b
   val ccext : 'b T.folder -> T.ty * T.ccext * 'b -> 'b
  ) : MLTREE_FOLD =
```

The type `folder` is defined as:

```
   type 'b folder =
       { stm   : T.stm * 'b -> 'b,
         rexp  : T.rexp * 'b -> 'b,
         fexp  : T.fexp * 'b -> 'b,
         ccexp : T.ccexp * 'b -> 'b
       }
```

23.0.3 MLTree Rewriting

The functor `MLTreeRewrite`[83] implements a generic term rewriting engine which is useful for performing various transformations on MLTree terms. Its signature is

```
signature MLTREE_REWRITE[84] =
sig
   structure T : MLTREE

  val rewrite :
       (* User supplied transformations *)
       { rexp  : (T.rexp -> T.rexp) -> (T.rexp -> T.rexp),
```

[80] **file:** `mltree/mltree-fold.sml`
[81] **file:** `mltree/mltree-fold.sig`
[82] **file:** `mltree/mltree-fold.sml`
[83] **file:** `mltree/mltree-rewrite.sml`
[84] **file:** `mltree/mltree-rewrite.sig`

```
        fexp  : (T.fexp -> T.fexp) -> (T.fexp -> T.fexp),
        ccexp : (T.ccexp -> T.ccexp) -> (T.ccexp -> T.ccexp),
        stm   : (T.stm -> T.stm) -> (T.stm -> T.stm)
      } -> T.rewriters
end
functor MLTreeRewrite[85]
  (structure T : MLTREE
   (* Extension *)
   val sext : T.rewriter -> T.sext -> T.sext
   val rext : T.rewriter -> T.rext -> T.rext
   val fext : T.rewriter -> T.fext -> T.fext
   val ccext : T.rewriter -> T.ccext -> T.ccext
  ) : MLTREE_REWRITE =
```

The type `rewriter` is defined in signature `MLTREE`[86] as:

```
   type rewriter =
       { stm   : T.stm -> T.stm,
         rexp  : T.rexp -> T.rexp,
         fexp  : T.fexp -> T.fexp,
         ccexp : T.ccexp -> T.ccexp
       }
```

23.0.4 MLTree Simplifier

The functor `MLTreeSimplify`[87] implements algebraic simplification and constant folding for MLTree. Its signature is:

```
signature MLTREE_SIMPLIFIER[88] =
sig

   structure T : MLTREE

   val simplify  :
        addressWidth : int  -> T.simplifier

end
functor MLTreeSimplifier[89]
  (structure T : MLTREE
   (* Extension *)
   val sext : T.rewriter -> T.sext -> T.sext
   val rext : T.rewriter -> T.rext -> T.rext
   val fext : T.rewriter -> T.fext -> T.fext
   val ccext : T.rewriter -> T.ccext -> T.ccext
  ) : MLTREE_SIMPLIFIER =
```

[85] **file:** `mltre/mltree-rewrite.sml`
[86] **file:** `mltree/mltree.sig`
[87] **file:** `mltree/mltree-simplify.sml`
[88] **file:** `mltree/mltree-simplify.sig`
[89] **file:** `mltree/mltree-simplify.sml`

Where type *simplifier* is defined in signature MLTREE[90] as:

```
type simplifier =
    { stm   : T.stm -> T.stm,
      rexp  : T.rexp -> T.rexp,
      fexp  : T.fexp -> T.fexp,
      ccexp : T.ccexp -> T.ccexp
    }
```

[90] **file:** mltree/mltree.sig

24 Instruction Selection

Instruction selection modules are reponsible for translating MLTree[91] statements into a flowgraph consisting of target machine instructions. MLRISC decomposes this complex task into *three* components:

Instruction selection modules which are responsible for mapping a sequence of MLTree statements into a sequence target machine code,

Flowgraph builders which are responsible for constructing the graph representation of the program from a sequence of target machine instructions, and

Client Extender which are responsible for compiling MLTree extensions (see also Section 22).

By detaching these components, extra flexiblity is obtained. For example, the MLRISC system uses two different internal representations. The first, cluster[92], is a *light-weight* representation which is suitable for simple compilers without extensive optimizations; the second, MLRISC IR[93], is a *heavy duty* representation which allows very complex transformations to be performed. Since the flowgraph builders are detached from the instruction selection modules, the same instruction selection modules can be used for both representations.

For consistency, the three components communicate to each other via the same stream[94] interface.

24.1 Interface Definition

All instruction selection modules satisfy the following signature:

```
signature MLTREECOMP[95] =
sig
   structure T : MLTREE[96]
   structure I : INSTRUCTIONS[97]
   structure C : CELLS[98]
      sharing T.LabelExp = I.LabelExp[99]
      sharing I.C = C

   type instrStream = (I.instruction,C.regmap,C.cellset) T.stream
   type mltreeStream = (T.stm,C.regmap,T.mlrisc list) T.stream

   val selectInstructions : instrStream -> mltreeStream
end
```

Intuitively, this signature states that the instruction selection module returns a function that can transform a stream of MLTree statements (`mltreeStream`) into a stream of instructions of the target machine (`instrStream`).

[91] url: `mltree.html`
[92] url: `cluster.html`
[93] url: `mlrisc-ir.html`
[94] url: `stream.html`
[95] **file:** `mltree/mltreecomp.sig`
[96] url: `mltree.html`
[97] url: `instructions.html`
[98] url: `cells.html`
[99] url: `labelexp.html`

24.1.1 Compiling Client Extensions

Compilation of client extensions to MLTREE is controlled by the following signature:

```
signature MLTREE_EXTENSION_COMP[100] =
sig
   structure T : MLTREE[101]
   structure I : INSTRUCTIONS[102]
   structure C : CELLS[103]
      sharing T.LabelExp = I.LabelExp[104]
      sharing I.C = C

   type reducer =
     (I.instruction,C.regmap,C.cellset,I.operand,I.addressing_mode) T.reducer

   val compileSext : reducer -> {stm:T.sext, an:T.an list} -> unit
   val compileRext : reducer -> {e:T.ty * T.rext, rd:C.cell, an:T.an list} -> unit
   val compileFext : reducer -> {e:T.ty * T.fext, fd:C.cell, an:T.an list} -> unit
   val compileCCext : reducer -> {e:T.ty * T.ccext, ccd:C.cell, an:T.an list} -> unit
end
```

Methods `compileSext`, `compileRext`, etc. are callbacks that are responsible for compiling MLTREE extensions. The arguments to these callbacks have the following meaning:

reducer The first argument is always the `reducer`, which contains internal methods for translating ML-Tree constructs into machine code. These methods are supplied by the instruction selection modules.

an This is a list of annotations that should be attached to the generated code.

ty, fty These are the types of the extension construct.

stm, e This are the extension statement and expression.

rd, fd, cd These are the target registers of the expression extension, i.e. the callback should generate the appropriate code for the expression and writes the result to this target.

For example, when an instruction selection encounters a `FOR`(i, a, b, s) statement extension defined in Section 22, the callback

```
compileStm reducer { stm=FOR(i,a,b,s), an=an }
```

is be involved.

The `reducer` type is defined in the signature `MLTREE`[105] and has the following type:

[100] **file:** `mltree/mltreecomp.sig`
[101] url: `mltree.html`
[102] url: `instructions.html`
[103] url: `cells.html`
[104] url: `labelexp.html`
[105] **file:** `mltree/mltree.sig`

```
datatype reducer =
  REDUCER of
  { reduceRexp    : rexp -> reg,
    reduceFexp    : fexp -> reg,
    reduceCCexp   : ccexp -> reg,
    reduceStm     : stm * an list -> unit,
    operand       : rexp -> I.operand,
    reduceOperand : I.operand -> reg,
    addressOf     : rexp -> I.addressing_mode,
    emit          : I.instruction * an list -> unit,
    instrStream   : (I.instruction,C.regmap,C.cellset) stream,
    mltreeStream  : (stm,C.regmap,mlrisc list) stream
  }
```

The components of the reducer are

reduceRexp, reduceFexp, reduceCCexp These functions take an expression of type integer, floating point and condition code, translate them into machine code and return the register that holds the result.

reduceStm This function takes an MLTree statement and translates it into machine code. it also takes a second argument, which is the list of annotations that should be attached to the statement.

operand This function translates an `rexp` into an `operand` of the machine architecture.

reduceOperand This function takes an operand of the machine architecture and reduces it into an integer register.

addressOf This function takes an `rexp`, treats it as an address expression and translates it into the appropriate `addresssing_mode` of the target architecture.

emit This function emits an instruction with attached annotations to the instruction stream

instrStream, mltreeStream These are the instruction stream and mltree streams that are currently bound to the extender.

24.1.2 Extension Example

Here is an example of how the extender mechanism can be used, using the `DSP_MLTREE` extensions defined in Section 22. We need supply two new functions, `compileDSPStm` for compiling the `FOR` construct, and `compileDSPRexp` for compiling the `SUM`, and saturated arithmetic instructions.

The first function, `compileDSPStm`, is generic and simply translates the `FOR` loop into the appropriate branches. Basically, we will translate `FOR`$(i, start, stop, body)$ into the following loop in pseudo code:

```
       limit = stop
       i   = start
       goto test
loop:  body
       i = i + 1
test:  if i <= limit goto loop
```

This transformation can be implemented as follows:

```
functor DSPMLTreeExtensionComp
   (structure I : DSP_INSTRUCTION_SET
    structure T : DSP_MLTREE
      sharing I.LabelExp = T.LabelExp
   ) =
struct
  structure I = I
  structure T = T
  structure C = I.C

  type reducer =
    (I.instruction,C.regmap,C.cellset,I.operand,I.addressing_mode) T.reducer

  fun mark(s, []) = s
    | mark(s, a::an) = mark(ANNOTATION(s, a), an)
  fun compileSext (REDUCER{reduceStm, ...})
     {stm=FOR(i, start, stop, body), an} =
  let val limit = C.newReg()
      val loop  = Label.newLabel ""
      val test  = Label.newLabel ""
  in  reduceStm(
        SEQ[MV(32, i, start),
            MV(32, limit, stop),
            JMP([], [LABEL(LabelExp.LABEL test)], []),
            LABEL loop,
            body,
            MV(32, i, ADD(32, REG(32, i), LI 1),
            LABEL test,
            mark(BCC([],
                   CMP(32, LE, REG(32, i), REG(32, limit)),
                     loop),
                 an),
           ]
     )
  end

  ...
```

In this transformation, we have chosen to proprogate the annotation `an` into the branch constructor.

Assuming the target architecture that we are translated into contains saturated arithmetic instructions `SADD`, `SSUB`, `SMUL` and `SDIV`, the DSP extensions `SUM` and saturated arithmetic expressions can be handled as follows.

```
  fun compileRext (REDUCER{reduceStm, reduceRexp, emit, ...})
      {ty, e, rd, an} =
  (case (ty,e) of
     (_,T.SUM(i, a, b, exp)) =>
       reduceStm(SEQ[MV(ty, rd, LI 0),
                     FOR(i, a, b,
```

```
                    mark(MV(ty, rd, ADD(ty, REG(ty, rd), exp)), an))
                ]
            )
  | (32,T.SADD(x,y)) => emit(I.SADD{r1=reduceRexp x,r2=reduceRexp y,rd=rd},an)
  | (32,T.SSUB(x,y)) => emit(I.SSUB{r1=reduceRexp x,r2=reduceRexp y,rd=rd},an)
  | (32,T.SMUL(x,y)) => emit(I.SMUL{r1=reduceRexp x,r2=reduceRexp y,rd=rd},an)
  | (32,T.SDIV(x,y)) => emit(I.SDIV{r1=reduceRexp x,r2=reduceRexp y,rd=rd},an)
  | ...
  )

  fun compileFext _ _ = ()
  fun compileCCext _ _ = ()

end
```

Note that in this example, we have simply chosen to reduce a `SUM` expression into the high level `FOR` construct. Clearly, other translation schemes are possible.

24.2 Instruction Selection Modules

Here are the actual code for the various back ends:

1. `Sparc`[106]
2. `PA-RISC`[107]
3. `Alpha`[108]
4. `Power PC`[109]
5. `X86`[110]
6. C6xx

[106] **file:** `sparc/mltree/sparc.sml`
[107] **file:** `hppa/mltree/hppa.sml`
[108] **file:** `alpha/mltree/alpha.sml`
[109] **file:** `ppc/mltree/ppc.sml`
[110] **file:** `x86/mltree/x86.sml`

25 Assemblers

25.0.1 Overview

Assemblers in MLRISC satisfy the signature INSTRUCTION_EMITTER[111], which is defined as:

```
signature INSTRUCTION_EMITTER =
sig
   structure I : INSTRUCTIONS[112]
   structure C : CELLS[113]
   structure S : INSTRUCTION_STREAM[114]
   structure P : PSEUDO_OPS[115]
      sharing I.C = C
      sharing S.P = P

   val makeStream : Annotations.annotations ->
                    ((int -> int) -> I.instruction -> unit,
                     unit,'b,'c,'d,'e) S.stream
end
```

The function makeStream returns an instruction stream. By default the output is bound to the stream AsmStream.asmOutStream defined in the structure AsmStream[116] at creation time.

The structure AsmStream satisfy the following signature.

```
signature ASM_STREAM = sig
  val asmOutStream : TextIO.outstream ref
  val withStream : TextIO.outstream -> ('a -> 'b) -> 'a -> 'b
end
```

25.0.2 Redirecting the Output

It is possible to redirect the output of an instruction stream. For example, the following statement

```
val asm = makeStream []
```

binds the output of asm to AsmStream.asmOutStream, which by default is just TextIO.stdOut. On the other hand, the statement

```
val asm = AsmStream.withStream mystream makeStream []
```

binds the output of asm to mystream.

[111] **file:** emit/instruction-emitter.sig
[112] url: instructions.html
[113] url: cells.html
[114] url: streams.html
[115] url: pseudo-ops.html
[116] **file:** emit/asmStream.sml

25.0.3 More Details

Assemblers are automatically generated by the MDGen[117] tool. Some specific generated assemblers are listed below:

1. `Sparc`[118]
2. `Hppa`[119]
3. `Alpha`[120]
4. `Power PC`[121]
5. `X86`[122]

[117] url: `mlrisc-md.html`
[118] **file:** `sparc/emit/sparcAsm.sml`
[119] **file:** `hppa/emit/hppaAsm.sml`
[120] **file:** `alpha/emit/alphaAsm.sml`
[121] **file:** `ppc/emit/ppcAsm.sml`
[122] **file:** `x86/emit/x86Asm.sml`

26 Machine Code Emitters

26.0.1 Overview

MLRISC lets the client to directly emit machine code and bypass the traditional assembly mechanism.

Machine code emitters in MLRISC satisfy the signature INSTRUCTION_EMITTER[123], which is defined as:

```
signature INSTRUCTION_EMITTER =
sig

   structure I : INSTRUCTIONS[124]
   structure C : CELLS[125]
   structure S : INSTRUCTION_STREAM[126]
   structure P : PSEUDO_OPS[127]
      sharing I.C = C
      sharing S.P = P

   val makeStream : Annotations.annotations ->
                    ((int -> int) -> I.instruction -> unit,
                     unit,'b,'c,'d,'e) S.stream

end
```

The function makeStream returns an instruction stream. The output, a stream of bytes, is direct to the client supplied structure which satisfy the CODE_STRING[128] interface. This signature is defined as follows:

```
signature CODE_STRING = sig
  type code_string
  val init          : int -> unit
  val update        : int * Word8.word -> unit
  val getCodeString : unit -> code_string
end
```

26.0.2 More Details

Machine code emitters are automatically generated by the MDGen[129] tool. Some specific generated emitters are listed below:

1. Sparc[130]
2. Hppa[131]

[123] **file:** emit/instruction-emitter.sig
[124] url: instructions.html
[125] url: cells.html
[126] url: streams.html
[127] url: pseudo-ops.html
[128] **file:** emit/code-string.sig
[129] url: mlrisc-md.html
[130] **file:** sparc/emit/sparcMC.sml
[131] **file:** hppa/emit/hppaMC.sml

3. `Alpha`[132]
4. `Power PC`[133]
5. `X86`[134]

[132] **file:** `alpha/emit/alphaMC.sml`
[133] **file:** `ppc/emit/ppcMC.sml`
[134] **file:** `x86/emit/x86MC.sml`

27 Delay Slot Filling

27.1 Overview

Superscalar architectures such as the Sparc, MIPS, and PA-RISC contain delayed branch and/or load instructions. Delay slot filling is necessary task of the back end to keep the instruction pipelines busy. To accomodate the intricate semantics of branch delay slot in various architectures, MLRISC uses the following very general framework for dealing with delayed instructions.

Instruction representation To make it easy to deal with instruction with delay slot, MLRISC allow the following extensions to instruction representations.

- Instructions with delay slot may have a nop flag. When this flag is true the delay slot is assumed to be filled with a NOP instruction.
- Instructions with delay slots that can be nullified may have a nullified flag. When this flag is true the branch delay slot is assumed to be nullified.

Nullification semantics Unfortunately, nullification semantics in architectures vary. In general, MLRISC allows the following additional nullification characteristics to be specified.

- Nullification can be specified as illegal; this is needed because some instructions can not be nullified
- When nullification is enabled, the semantics of the delay slot instruction may depend on the direction of the branch, and whether a conditional test succeeds.
- Certain class of instructions may be declared to be illegal to fit into certain class of delay slots.

For example, conditional branch instructions on the Sparc are defined as follows:

```
Bicc of {b:branch, a:bool, label:Label.label, nop:bool}
  asm: ``b<b><a>\t<label><nop>''
  padding: nop = true
  nullified: a = true and (case b of I.BA => false | _ => true)
  delayslot candidate: false
```

where `a` is *annul* flag and `nop` is the nop flag (see `the Sparc machine description`[135]). A constructor term

```
Bicc{b=BE, a=true, label=label, nop=true}
```

denotes the instruction sequence

```
be,a label
nop
```

while

```
Bicc{b=BE, a=false, label=label, nop=false}
```

denotes

```
be label
```

[135] **file:** `sparc/sparc.md`

27.2 The Interface

Architecture information about how delay slot filling is to be performed is described in the signature DELAY_SLOT_PROPERTIES[136].

```
signature DELAY_SLOT_PROPERTIES =
sig
   structure I : INSTRUCTIONS

   datatype delay_slot =
     D_NONE   | D_ERROR   | D_ALWAYS
   | D_TAKEN  | D_FALLTHRU

   val delaySlotSize : int
   val delaySlot : { instr : I.instruction, backward : bool } ->
   { n    : bool,
     nOn  : delay_slot,
     nOff : delay_slot,
     nop  : bool
   }
   val enableDelaySlot :
 {instr : I.instruction, n:bool, nop:bool} -> I.instruction
   val conflict :
         {regmap:int->int,src:I.instruction,dst:I.instruction} -> bool
   val delaySlotCandidate :
         { jmp : I.instruction, delaySlot : I.instruction } -> bool
   val setTarget : I.instruction * Label.label -> I.instruction
end
```

The components of this signature are:

delay_slot This datatype describes properties related to a delay slot.

- **D_NONE** This indicates that no delay slot is possible.
- **D_ERROR** This indicates that it is an error
- **D_ALWAYS** This indicates that the delay slot is always active
- **D_TAKEN** This indicates that the delay slot is only active when branch is taken
- **D_FALLTHRU** This indicates that the delay slot is only active when branch is not taken

delaySlotSize This is size of delay slot in bytes.

delaySlot This method takes an instruction `instr` and a flag indicating whether the branch is `backward`, and returns the delay slot properties of an instruction. The properties is described by four fields.

- **n : bool** This bit is if the nullified bit in the instruction is currently set.
- **nOn : delay_slot** This field indicates the delay slot type when the instruction is nullified.
- **nOff : delay_slot** This field indiciates the delay slot type when the instruction is not nullified.

[136] **file:** `backpatch/delaySlotProps.sig`

nop : bool This bit indicates whether there is an implicit padded nop.

enableDelaySlot This method set the nullification and nop flags of an instruction.

conflict This method checks whether there are any conflicts between instruction `src` and `dst`.

delaySlotCandidate This method checks whether instruction `delaySlot` is within the class of instructions that can fit within the delay slot of instruction `jmp`.

setTarget This method changes the branch target of an instruction.

27.2.1 Examples

For example,

```
delaySlot{instr=instr, backward=true} =
{n=true, nOn=D_ERROR, nOff=D_ALWAYS, nop=true}
```

means that the instruction nullification bit is on, the the nullification cannot be turned off, delay slot is always active (when not nullified), and there is currently an implicit padded nop.

```
delaySlot{instr=instr, backward=false} =
{n=false, nOn=D_NONE, nOff=D_TAKEN, nop=false}
```

means that the nullification bit is off, the delay slot is inactive when the nullification bit is off, the delay slot is only active when the (forward) branch is taken when `instr` is not-nullified, and there is no implicitly padded nop.

```
delaySlot{instr=instr, backward=true} =
{n=true, nOn=D_TAKEN, nOff=D_ALWAYS, nop=true}
```

means that the nullification bit is on, the delay slot is active on a taken (backward) branch when the nullification bit is off, the delay slot is always active when `instr` is not-nullified, and there is currently an implicitly padded nop.

28 Span Dependency Resolution

The span dependency resolution phase is used to resolve the values of client defined constants[137] and labels[138] in a program. An instruction whose immediate operand field contains a constant or label expression[139] which is too large is rewritten into a sequence of instructions to compute the same result. Similarly, short branches referencing labels that are too far are rewritten into the long form. For architectures that require the filling of delay slots, this is performed at the same time as span depedency resolution, to ensure maximum benefit results.

28.0.1 The Interface

The signature `SDI_JUMPS` describes architectural information about span dependence resolution.

```
signature SDI_JUMPS[140] = sig
  structure I : INSTRUCTIONS[141]
  structure C : CELLS[142]
    sharing I.C = C

  val branchDelayedArch : bool
  val isSdi : I.instruction -> bool
  val minSize : I.instruction -> int
  val maxSize : I.instruction -> int
  val sdiSize : I.instruction * (C.cell -> C.cell)
                             * (Label.label -> int) * int -> int
  val expand : I.instruction * int * int -> I.instruction list
end
```

The components in this interface are:

branchDelayedArch A flag indicating whether the architecture contains delay slots. For example, this would be true on the MIPS, Sparc, PA RISC; but would be false on the x86 and on the Alpha.

isSdi This function returns true if the instruction is *span dependent*, i.e. its size depends either on some unresolved constants, or on its position in the code stream.

sdiSize This function takes a span dependent instruction, a regmap[143], a mapping from labels to code stream position, and its current code stream position and returns the size of its expansion in bytes.

expand This function takes a span dependent instruction, its size, and its location and return its expansion.

The signature `BBSCHED` is the signature of the phase that performs span depedennce resolution and code generation.

[137] url: `constants.html`
[138] url: `labels.html`
[139] url: `labexp.html`
[140] **file:** `backpatch/sdi-jumps.sig`
[141] url: `instructions.html`
[142] url: `cells.html`
[143] url: `regmap.html`

```
signature BBSCHED[144] = sig
  structure F : FLOWGRAPH[145]

  val bbsched : F.cluster -> unit
  val finish : unit -> unit
  val cleanUp : unit -> unit
end
```

28.0.2 The Modules

Three different functors are present in the MLRISC system for performing span dependence resolution and code generator. Functor `BBSched2` is the simplest one, which does not perform delay slot filling.

```
functor BBSched2
  (structure Flowgraph : FLOWGRAPH[146]
   structure Jumps : SDI_JUMPS[147]
   structure Emitter : INSTRUCTION_EMITTER[148]
     sharing Emitter.P = Flowgraph.P
     sharing Flowgraph.I = Jumps.I = Emitter.I
  ): BBSCHED
```

Functor `SpanDependencyResolution` performs both span dependence resolution and delay slot filling at the same time.

```
functor SpanDependencyResolution
  (structure Flowgraph : FLOWGRAPH[149]
   structure Emitter : INSTRUCTION_EMITTER[150]
   structure Jumps : SDI_JUMPS[151]
   structure DelaySlot : DELAY_SLOT_PROPERTIES[152]
   structure Props : INSN_PROPERTIES[153]
     sharing Flowgraph.P = Emitter.P
     sharing Flowgraph.I = Jumps.I = DelaySlot.I = Props.I = Emitter.I
  ) : BBSCHED
```

Finally, functor `BackPatch` is a span dependency resolution module specially written for the x86[154] architecture.

```
functor BackPatch
  (structure CodeString : CODE_STRING[155]
```

[144] **file:** `backpatch/bbsched.sig`
[145] url: `cluster.html`
[146] **file:** `cluster/flowgraph.sig`
[147] **file:** `backpatch/sdi-jumps.sig`
[148] url: `mc.html`
[149] **file:** `cluster/flowgraph.sig`
[150] url: `mc.html`
[151] **file:** `backpatch/sdi-jumps.sig`
[152] url: `delayslots.html`
[153] **file:** `instructions/insnProps.sig`
[154] url: `x86.html`
[155] **file:** `emit/code-string.sig`

```
structure Jumps: SDI_JUMPS[156]
structure Props : INSN_PROPERTIES[157]
structure Emitter : MC_EMIT[158]
structure Flowgraph : FLOWGRAPH[159]
structure Asm : INSTRUCTION_EMITTER[160]
   sharing Emitter.I = Jumps.I = Flowgraph.I = Props.I = Asm.I) : BBSCHED
```

[156] **file:** backpatch/sdi-jumps.sig
[157] **file:** instructions/insnProps.sig
[158] **file:** backpatch/vlBatchPatch.sig
[159] url: cluster.html
[160] url: asm.html

29 The MLRISC Machine Description Language

29.1 Overview

MDGen is a machine description language is designed to automate various mundane and error prone tasks in developing a back-end for MLRISC. Currently, to target a new architecture the programmer must provide the following set of modules written in Standard ML:

- `CELLS`[161] – the properties of the register set and (some part of) memory hierarchy.
- `INSTRUCTIONS`[162] – the concrete instruction set representation.
- `INSNS_PROPERTIES`[163] – properties of the instructions.
- `SHUFFLE`[164] – methods to emit linearized code from parallel copies.
- `ASSEMBLER`[165] – the assembler
- `MC`[166] – the machine code emitter
- `SDI_JUMPS`[167] – methods for resolving span-dependent instructions.
- `DELAY_SLOTS_PROPERTIES`[168] – machine properties for delay slot filling, if a machine architecture contains branch delay slots or load delay slots.
- `SSA_PROPERTIES`[169] – semantics properties for performing optimizations in Static Single Assignment form.

In general, writing a backend is tedious even with SML's abstraction capabilities. Furthermore, the machine description is procedural in natural and must be checked by hand.

29.2 What is in MDGen?

The MDGen tool simplifies the process of developing a new MLRISC backend. MDGen provides the following:

- A representation description language for specifying the machine encoding of the instruction set, using an extension of ML's algebraic datatype facility.
- A semantics description language for specifying the abstract semantics of the instructions.

[161] **file:** `instructions/cells.sig`
[162] **file:** `instructions/instructions.sig`
[163] **file:** `instructions/insnProps.sig`
[164] **file:** `instructions/shuffle.sig`
[165] **file:** `emit/instruction-emitter.sig`
[166] **file:** `emit/instruction-emitter.sig`
[167] **file:** `../backpatch/sdi-jumps.sig`
[168] **file:** `../backpatch/delaySlotProps.sig`
[169] **file:** `../SSA/ssaProps.sig`

Both sub-languages are based on ML's syntax and semantics, so they should be readily familiar to all MLRISC users.

A backend developer can specify a new machine architecture using the MDGen language, and in turn, the MDGen tool generates ML modules that are required by the MLRISC system.

The basic concepts of MDGen are inspired largely from Norman Ramsey's New Jersey Machine Code Tool Kit[170] and Ramsey and Davidson's Lambda RTL[171].

29.3 A Sample Description

Here we present a sample MDGen description, using the Alpha as an example. We highlight all keywords in the MDGen language in. A typical machine description is structured as follows:

```
architecture Alpha =
struct
   name "Alpha"
   superscalar
   little endian
   lowercase assembly
end
```

29.3.1 Storage cells and locations.
29.3.3 Instruction encoding formats specification.
29.3.2 Instruction definition.

Here, we declare that the Alpha is a superscalar machine using little endian encoding. Furthermore, assembly output should be displayed in lowercase– this is for personal esthetic reasons only; most assemblers are case insensitive.

29.3.1 Specifying Storage Cells and Locations

A *cell* is an abstract resource location for holding data values. On typical machines, the types of cells include general purpose registers, floating point registers, and condition code registers.

The `storage` declaration defines different *cellkinds*. MLRISC requires the cellkinds `GP`, `FP`, `CC` to be defined. These are the cellkinds for general purpose registers, floating point registers and condition code registers.

In the following sequence of declarations, a few things are defined:

- The cellkinds `GP, FP, CC` are defined. Furthermore, the cellkinds `MEM, CTRL`, which stand for memory and control (dependence), are also implicitly defined.
- The `assembly as` clauses specify how a specific cell type is to be displayed. Here, we specify that register 30, the stack pointer, should be displayed specially as `$sp`.
- The `in cellset` clause, when attached, tells MDGen that the associated cellkind should be part of the cellset . The clause `in cellset GP` tells MDGen that the a cell of type `CC` should be treated the same as a `GP`
- The `locations` declarations define a few abbreviations: `stackptrR` is the stack pointer, `asmTmpR` is the assembly temporary, `fasmTmp` is the floating point assembly temporary etc.

[170] url: `http://www.cs.virginia.edu/ nr/toolkit`
[171] url: `http://www.cs.virginia.edu/zephyr/csdl/lrtlindex.html`

```
storage
  GP = 32 cells of 64 bits in cellset called "register"
     assembly as (fn 30 => "$sp"
                   | r => "$"^Int.toString r)
| FP = 32 cells of 64 bits in cellset called "floating point register"
     assembly as (fn f => "f"^Int.toString f)
| CC = cells of 64 bits in cellset GP called "condition code register"
             assembly as "cc"
locations
    stackptrR = $GP[30]
and asmTmpR   = $GP[28]
and fasmTmp   = $FP[30]
and GPReg r   = $GP[r]
and FPReg f   = $GP[f]
```

29.3.2 Specifying the Representation of Instructions

```
structure Instruction =
struct
datatype ea =
    Direct of $GP
  | FDirect of $FP
  | Displace of base: $GP, disp:int

datatype operand =
    REGop of $GP        ``<GP>'' (GP)
  | IMMop of int        ``<int>''
  | HILABop of LabelExp.labexp       ``hi(<emit_labexp labexp>)''
  | LOLABop of LabelExp.labexp       ``lo(<emit_labexp labexp>)''
  | LABop of LabelExp.labexp        ``<emit_labexp labexp>''
  | CONSTop of Constant.const       ``<emit_const const>''

(*
 * When I say ! after the datatype name XXX, it means generate a
 * function emit_XXX that converts the constructors into the corresponding
 * assembly text.  By default, it uses the same name as the constructor,
 * but may be modified by the lowercase/uppercase assembly directive.
 *
 *)
datatype branch! =
   BR  0x30
             | BSR 0x34
                          | BLBC 0x3
 | BEQ  0x39 | BLT 0x3a | BLE  0x3b
 | BLBS 0x3c | BNE 0x3d | BGE  0x3e
 | BGT  0x3f

datatype fbranch! =
```

```
              FBEQ 0x31 | FBLT 0x32
  | FBLE 0x33             | FBNE 0x35
  | FBGE 0x36 | FBGT 0x37

datatype load! = LDL 0x28 | LDL_L 0x2A | LDQ 0x29 | LDQ_L 0x2B | LDQ_U 0x0B
datatype store! = STL 0x2C | STQ 0x2D | STQ_U 0x0F
datatype fload[0x20..0x23]! = LDF | LDG | LDS | LDT
datatype fstore[0x24..0x27]! = STF | STG | STS | STT

(* non-trapping opcodes *)
datatype operate! = (* table C-5 *)
    ADDL  (0wx10,0wx00)                       | ADDQ (0wx10,0wx20)
                        | CMPBGE(0wx10,0wx0f) | CMPEQ (0wx10,0wx2d)
  | CMPLE (0wx10,0wx6d) | CMPLT (0wx10,0wx4d) | CMPULE (0wx10,0wx3d)
  | CMPULT(0wx10,0wx1d) | SUBL  (0wx10,0wx09)
  | SUBQ  (0wx10,0wx29)
  | S4ADDL(0wx10,0wx02) | S4ADDQ (0wx10,0wx22) | S4SUBL (0wx10,0wx0b)
  | S4SUBQ(0wx10,0wx2b) | S8ADDL (0wx10,0wx12) | S8ADDQ (0wx10,0wx32)
  | S8SUBL(0wx10,0wx1b) | S8SUBQ (0wx10,0wx3b)

  | AND   (0wx11,0wx00) | BIC    (0wx11,0wx08) | BIS    (0wx11,0wx20)
  | CMOVEQ(0wx11,0wx24) | CMOVLBC(0wx11,0wx16) | CMOVLBS(0wx11,0wx14)
  | CMOVGE(0wx11,0wx46) | CMOVGT (0wx11,0wx66) | CMOVLE (0wx11,0wx64)
  | CMOVLT(0wx11,0wx44) | CMOVNE (0wx11,0wx26) | EQV (0wx11,0wx48)
  | ORNOT (0wx11,0wx28) | XOR    (0wx11,0wx40)

  | EXTBL (0wx12,0wx06) | EXTLH  (0wx12,0wx6a) | EXTLL(0wx12,0wx26)
  | EXTQH (0wx12,0wx7a) | EXTQL  (0wx12,0wx36) | EXTWH(0wx12,0wx5a)
  | EXTWL (0wx12,0wx16) | INSBL  (0wx12,0wx0b) | INSLH(0wx12,0wx67)
  | INSLL (0wx12,0wx2b) | INSQH  (0wx12,0wx77) | INSQL(0wx12,0wx3b)
  | INSWH (0wx12,0wx57) | INSWL  (0wx12,0wx1b) | MSKBL(0wx12,0wx02)
  | MSKLH (0wx12,0wx62) | MSKLL  (0wx12,0wx22) | MSKQH(0wx12,0wx72)
  | MSKQL (0wx12,0wx32) | MSKWH  (0wx12,0wx52) | MSKWL(0wx12,0wx12)
  | SLL   (0wx12,0wx39) | SRA    (0wx12,0wx3c) | SRL  (0wx12,0wx34)
  | ZAP   (0wx12,0wx30) | ZAPNOT (0wx12,0wx31)
  | MULL  (0wx13,0wx00)                       | MULQ (0wx13,0wx20)
                        | UMULH  (0wx13,0wx30)
  | SGNXL "addl" (0wx10,0wx00) (* same as ADDL *)

(* conditional moves *)

datatype pseudo_op! = DIVL | DIVLU

datatype operateV! = (* table C-5 opc/func *)
     ADDLV (0wx10,0wx40) | ADDQV (0wx10,0wx60)
   | SUBLV (0wx10,0wx49) | SUBQV (0wx10,0wx69)
   | MULLV (0wx13,0wx00) | MULQV (0wx13,0wx60)
```

```
datatype foperate! =   (* table C-6 *)
   CPYS    (0wx17,0wx20)  | CPYSE (0wx17,0wx022)    | CPYSN   (0wx17,0wx021)
 | CVTLQ   (0wx17,0wx010) | CVTQL (0wx17,0wx030)    | CVTQLSV (0wx17,0wx530)
 | CVTQLV  (0wx17,0wx130)
 | FCMOVEQ (0wx17,0wx02a) | FCMOVEGE (0wx17,0wx02d) | FCMOVEGT (0wx17,0wx02f)
 | FCMOVLE (0wx17,0wx02e) | FCMOVELT (0wx17,0wx02c) | FCMOVENE (0wx17,0wx02b)
 | MF_FPCR (0wx17,0wx025) | MT_FPCR  (0wx17,0wx024)

                      (* table C-7 *)
 | CMPTEQ  (0wx16,0wx0a5) | CMPTLT (0wx16,0wx0a6)   | CMPTLE  (0wx16,0wx0a7)
 | CMPTUN  (0wx16,0wx0a4)

datatype foperateV! =
       ADDSSUD  0wx5c0
     | ADDTSUD  0wx5e0
     | CVTQSC   0wx3c
     | CVTQTC   0wx3e
     | CVTTSC   0wx2c
     | CVTTQC   0wx2f
     | DIVSSUD  0wx5ec
     | DIVTSUD  0wx5c3
     | MULSSUD  0wx5c2
     | MULTSUD  0wx5e2
     | SUBSSUD  0wx5c1
     | SUBTSUD  0wx5e1

datatype osf_user_palcode! =
   BPT 0x80 | BUGCHK 0x81 | CALLSYS 0x83
 | GENTRAP 0xaa | IMB 0x86 | RDUNIQUE 0x9e | WRUNIQUE 0x9f

end (* Instruction *)
```

29.3.3 Specifying the Instruction Encoding Formats

The Alpha has very simple instruction encoding formats.

```
instruction formats 32 bits
  Memoryopc:6, ra:5, rb:GP 5, disp: signed 16 (* p3-9 *)
   (* derived from Memory *)
| LoadStoreopc,ra,rb,disp =
    let val disp =
        case disp of
          I.REGop rb => emit_GP rb
        | I.IMMop i  => itow i
        | I.HILABop le => itow(LabelExp.valueOf le)
        | I.LOLABop le => itow(LabelExp.valueOf le)
        | I.LABop le => itow(LabelExp.valueOf le)
        | I.CONSTop c => itow(Constant.valueOf c)
```

```
    in  Memoryopc,ra,rb,disp
    end
| ILoadStoreopc,r:GP,b,d = LoadStoreopc,ra=r,rb=b,disp=d
| FLoadStoreopc,r:FP,b,d = LoadStoreopc,ra=r,rb=b,disp=d

| Jumpopc:6,ra:GP 5,rb:GP 5,h:2,disp:int signed 14   (* table C-3 *)
| Memory_funopc:6, ra:GP 5, rb:GP 5, func:16     (* p3-9 *)
| Branchopc:branch 6, ra:GP 5, disp:signed 21           (* p3-10 *)
| Fbranchopc:fbranch 6, ra:FP 5, disp:signed 21          (* p3-10 *)
    (* p3-11 *)
| Operate0opc:6,ra:GP 5,rb:GP 5,sbz:13..15,_:1=0,func:5..11,rc:GP 5
    (* p3-11 *)
| Operate1opc:6,ra:GP 5,lit:signed 13..20,_:1=1,func:5..11,rc:GP 5
| Operateopc,ra,rb,func,rc =
    (case rb of
      I.REGop rb => Operate0opc,ra,rb,func,rc,sbz=0w0
    | I.IMMop i  => Operate1opc,ra,lit=itow i,func,rc
    | I.HILABop le => Operate1opc,ra,lit=itow(LabelExp.valueOf le),func,rc
    | I.LOLABop le => Operate1opc,ra,lit=itow(LabelExp.valueOf le),func,rc
    | I.LABop le => Operate1opc,ra,lit=itow(LabelExp.valueOf le),func,rc
    | I.CONSTop c => Operate1opc,ra,lit=itow(Constant.valueOf c),func,rc
    )
| Foperateopc:6,fa:FP 5,fb:FP 5,func:5..15,fc:FP 5
| Palopc:6=0,func:26
```

29.3.4 Specifying the instruction set

```
structure MC =
struct
   (* compute displacement address *)
   fun disp lab = itow(Label.addrOf lab - !loc - 4) ~>> 0w2
end

(*
 * The main instruction set definition consists of the following:
 *  1) constructor-like declaration defines the view of the instruction,
 *  2) assembly directive in funny quotes `` '',
 *  3) machine encoding expression,
 *  4) semantics expression in [[ ]],
 *  5) delay slot directives etc (not necessary in this architecture!)
 *)
instruction
  DEFFREG of $FP       (* define a floating point register *)
    ``deffreg <FP>''
     (* Pseudo instruction for the register allocator *)

(* Load/Store *)
| LDA of r: $GP, b: $GP, d:operand       (* use of REGop is illegal *)
```

```
    ``lda ^r>, <d>()''
    ILoadStoreopc=0w08,r,b,d

  | LDAH of r: $GP, b: $GP, d:operand (* use of REGop is illegal *)
    ``ldah ^r>, <d>()''
    ILoadStoreopc=0w09,r,b,d

  | LOAD of ldOp:load, r: $GP, b: $GP, d:operand, mem:Region.region
    ``<ldOp> ^r>, <d>()''
    ILoadStoreopc=emit_load ldOp,r,b,d

  | STORE of stOp:store, r: $GP, b: $GP, d:operand, mem:Region.region
    ``<stOp> ^r>, <d>()''
    ILoadStoreopc=emit_store stOp,r,b,d

  | FLOAD of ldOp:fload, r: $FP, b: $GP, d:operand, mem:Region.region
    ``<ldOp> ^r>, <d>()''
    FLoadStoreopc=emit_fload ldOp,r,b,d

  | FSTORE of stOp:fstore, r: $FP, b: $GP, d:operand, mem:Region.region
    ``<stOp> ^r>, <d>()''
    FLoadStoreopc=emit_fstore stOp,r,b,d

  (* Control Instructions *)
  | JMPL of r: $GP, b: $GP, d:int * Label.label list
    ``jmpl ^r>, <d>()''
    Jumpopc=0wx1a,h=0w0,ra=r,rb=b,disp=d   (* table C-3 *)

  | JSR of r: $GP, b: $GP, d:int * C.cellset * C.cellset
    ``jsr ^r>, <d>()''
    Jumpopc=0wx1a,h=0w1,ra=r,rb=b,disp=d

  | RET of r: $GP, b: $GP, d:int
    ``ret ^r>, <d>()''
    Jumpopc=0wx1a,h=0w2,ra=r,rb=b,disp=d

  | BRANCH of branch * $GP * Label.label
    ``<branch> <GP>, <label>''
    Branchopc=branch,ra=GP,disp=disp label

  | FBRANCH of fbranch * $FP * Label.label
    ``<fbranch> <FP>, <label>''
    Fbranchopc=fbranch,ra=FP,disp=disp label

  (* Integer Operate *)
  | OPERATE of oper:operate, ra: $GP, rb:operand, rc: $GP
      ``<oper> ^ra>, <rb>, <rc>''
       (let val (opc,func) = emit_operate oper
```

```
        in  Operateopc,func,ra,rb,rc
        end)

  | OPERATEV of oper:operateV, ra: $GP, rb:operand, rc: $GP
      ``<oper> ^ra>, <rb>, <rc>''
       (let val (opc,func) = emit_operateV oper
        in  Operateopc,func,ra,rb,rc
        end)

  | PSEUDOARITH of oper: pseudo_op, ra: $GP, rb:operand, rc: $GP,
            tmps: C.cellset
      ``<oper> ^ra>, <rb>, <rc>''

  (* Copy instructions *)
  | COPY of dst: $GP list, src: $GP list,
             impl:instruction list option ref, tmp: ea option
      ``<app emitInstr (Shuffle.shuffleregmap,tmp,dst,src)>''
  | FCOPY of dst: $FP list, src: $FP list,
              impl:instruction list option ref, tmp: ea option
      ``<app emitInstr (Shuffle.shufflefpregmap,tmp,dst,src)>''

  (* Floating Point Operate *)
  | FOPERATE of oper:foperate, fa: $FP, fb: $FP, fc: $FP
      ``<oper> ^fa>, <fb>, <fc>''
      (let val (opc,func) = emit_foperate oper
       in  Foperateopc,func,fa,fb,fc
       end)

  (* Trapping versions of the above *)
  | FOPERATEV of oper:foperateV, fa: $FP, fb: $FP, fc: $FP
      ``<oper> ^fa>, <fb>, <fc>''
       Foperateopc=0wx16,func=emit_foperateV oper,fa,fb,fc

  (* Misc *)
  | TRAPB        (* Trap barrier *)
      ``trapb''
       Memory_funopc=0wx18,ra=31,rb=31,func=0wx0

  | CALL_PAL of code:osf_user_palcode, def: $GP list, use: $GP list
      ``call_pal <code>''
       Palfunc=emit_osf_user_palcode code
end
```

29.4 4 Machine Descriptions

Here are some machine descriptions in varing degree of completion.

- `Sparc` [172]
- `Hppa` [173]
- `Alpha` [174]
- `PowerPC` [175]
- `X86` [176]

29.5 Syntax Highlighting Macros

- For vim 5.3

[172] **file:** `../sparc/sparc.md`
[173] **file:** `../hppa/hppa.md`
[174] **file:** `../alpha/alpha.md`
[175] **file:** `../ppc/ppc.md`
[176] **file:** `../X86/X86.md`

30 The Graph Library

30.1 Overview

Graphs are the most fundamental data structure in the MLRISC system, and in fact in many optimizing compilers. MLRISC now contains an extensive library for working with graphs.

All graphs in MLRISC are modeled as edge- and node-labeled directed multi-graphs. Briefly, this means that nodes and edges can carry user supplied data, and multiple directed edges can be attached between any two nodes. Self-loops are also allowed.

A node is uniquely identified by its `node_id`, which is simply an integer. Node ids can be assigned externally by the user, or else generated automatically by a graph. All graphs keep track of all node ids that are currently used, and the method `new_id : unit -> node_id` generates a new unused id.

A node is modeled as a node id and node label pair, (i,l). An edge is modeled as a triple $i \rightarrow_l j$, which contains the *source* and *target* node ids i and j, and the edge label l. These types are defined as follows:

```
type 'n node = node_id * 'n
type 'e edge = node_id * node_id * 'e
```

30.1.1 The graph signature

All graphs are accessed through an abstract interface of the polymorphic type `('n,'e,'g) graph`. Here, `'n` is the type of the node labels, `'e` is the type of the edge labels, and `'g` is the type of any extra information embedded in a graph. We call the latter `graph info`.

Formally, a graph G is a quadruple (V,L,E,I) where V is a set of node ids, $L : V- >' a$ is a node labeling function from vertices to node labels, E is a multi-set of labeled-edges of type $V*V*'e$, and $I :' g$ is the graph info.

The interface of a graph is packaged into a record of methods that manipulate the base representation:

```
signature GRAPH[177] = sig
  type node_id = int
  type 'n node = node_id * 'n
  type 'e edge = node_id * node_id * 'e

  exception Graph of string
  exception Subgraph
  exception NotFound
  exception Unimplemented
  exception Readonly

  datatype ('n,'e,'g) graph = GRAPH of ('n,'e,'g) graph_methods
  withtype ('n,'e,'g) graph_methods =
      { name            : string,
        graph_info      : 'g,
        (* selectors *)
        (* mutators *)
        (* iterators *)
      }
end
```

[177] **file:** `graphs/graph.sig`

A few exceptions are predefined in this signature, which have the following informal interpretation. Exception `Graph` is raised when a bug is encountered. The exception `Subgraph` is raised if certain semantics constraints imposed on a graph are violated. The exception `NotFound` is raised if lookup of a node id fails. The exception `Unimplemented` is raised if a certain feature is accessed but is undefined on the graph. The exception `Readonly` is raised if the graph is readonly and an update operation is attempted.

30.1.2 Selectors

Methods that access the structure of a graph are listed below:

`nodes : unit -> 'n node list`	Return a list of all nodes in a graph em
`edges : unit -> 'e edge list`	Return a list of all edges in a graph
`order : unit -> int`	Return the number of nodes in a graph. The graph is empty if its order is zer
`size : unit -> int`	Return the number of edges in a graph
`capacity : unit -> int`	Return the maximum node id in the graph, plus 1. This can be used as a new
`succ : node_id -> node_id list`	Given a node id i, return the node ids of all its successors, i.e. $\{j \mid i \rightarrow_l j \in E\}$
`pred : node_id -> node_id list`	Given a node id j, return the node ids of all its predecessors, i.e. $\{i \mid i \rightarrow_l j \in$
`out_edges : node_id -> 'e edge list`	Given a node id i, return all the out-going edges from node i, i.e. all edges wh
`in_edges : node_id -> 'e edge list`	Given a node id j, return all the in-coming edges from node j, i.e. all edges v
`has_edge : node_id * node_id -> bool`	Given two node ids i and j, find out if an edge with source i and target j exis
`has_node : node_id -> bool`	Given a node id i, find out if a node of id i exists.
`node_info : node_id -> 'n`	Given a node id, return its node label. If the node does not exist, raise excep

30.1.3 Graph hierarchy

A graph G may in fact be a subgraph of a *base graph* G', or obtained from G' via some transformation T. In such cases the following methods may be used to determine of the relationship between G and G'. An *entry edge* is an edge in G' that terminates at a node in G, but is not an edge in G. Similarly, an *exit edge* is an edge in G' that originates from a node in G, but is not an edge in G. An *entry node* is a node in G that has an incoming entry edge. An *exit node* is a node in G that has an out-going exit edge. If G is not a subgraph, all these methods will return nil.

`entries : unit -> node_id list`	Return the node ids of all the entry nodes.
`exits : unit -> node_id list`	Return the node ids of all the exit nodes.
`entry_edges : node_id -> 'e edge list`	Given a node id i, return all the entry edges whose sources are i.
`exit_edges : node_id -> 'e edge list`	Given a node id i, return all the exit edges whose targets are i.

30.1.4 Mutators

Methods to update a graph are listed below:

`new_id : unit -> node_id`	Return a unique node id guaranteed to be absent in the current g
`add_node : 'n node -> unit`	Insert node into the graph. If a node of the same node id already
`add_edge : 'e edge -> unit`	Insert an edge into the graph.
`remove_node : node_id -> unit`	Given a node id n, remove the node with the node id from the gr
`set_out_edges : node_id * 'e edge list -> unit`	Given a node id n, and a list of edges $e_1, \ldots, e_n$ with sources n, re
`set_in_edges : node_id * 'e edge list -> unit`	Given a node id n, and a list of edges $e_1, \ldots, e_n$ with targets n, rep
`set_entries : node_id list -> unit`	Set the entry nodes of a graph.
`set_exits : node_id list -> unit`	Set the exit nodes of a graph.
`garbage_collect : unit -> unit`	Reclaim all node ids of nodes that have been removed by `remove`

30.1.5 Iterators

Two primitive iterators are supported in the graph interface. Method `forall_nodes` iterates over all the nodes in a graph, while method `forall_edges` iterates over all the edges. Other more complex iterators can be found in other modules.

`forall_nodes : ('n node -> unit) -> unit`	Given a function f on nodes, apply f on all the nodes in the graph.
`forall_edges : ('e edge -> unit) -> unit`	Given a function f on edges, apply f on all the edges in the graph.

30.1.6 Manipulating a graph

Since operations on the graph type are packaged into a record, an "object oriented" style of graph manipulation should be used. For example, if `G` is a graph object, then we can obtain all the nodes and edges of `G` as follows.

```
val GRAPH g = G
val edges = #edges g ()
val nodes = #nodes g ()
```

We can view `#edges g` as sending the message to `G`. While all this seems like mere syntactic deviation from the usual signature/structure approach, there are two crucial differences which we will exploit: *(i)* records are first class objects while structures are not (consequently late binding of "methods" and cannot be easily simulated on the structure level); *(ii)* recursion is possible on the type level, while recursive structures are not available. The extra flexibility of this choice becomes apparent with the introduction of views later.

30.1.7 Creating a Graph

A graph implementation has the following signature

```
signature GRAPH_IMPLEMENTATION[178] = sig
  val graph : string * 'g * int -> ('n,'e,'g) graph
end
```

The function `graph` takes a string (the name of the graph), graph info, and a default size as arguments and create an empty graph.

The functor `DirectedGraph`:

[178] **file:** `graphs/graphimpl.sig`

```
functor DirectedGraph(A : ARRAY_SIG) : GRAPH_IMPLEMENTATION
```

implements a graph using adjacency lists as internal representation. It takes an array type as a parameter. For graphs with node ids that are dense enumerations, the `DynamicArray` structure should be used as the parameter to this functor. The structure `DirectedGraph` is predefined as follows:

```
structure DirectedGraph[179] = DirectedGraph(DynamicArray)
```

For node ids that are sparse enumerations, the structure `HashArray`, which implements integer-keyed hash tables with the signature of arrays, can be used as argument to `DirectedGraph`. For graphs with fixed sizes determined at creation time, the structure `Array` can be used (see also functor `UndoableArray`[180], which creates arrays with undoable updates, and transaction-like semantics.)

30.1.8 Basic Graph Algorithms

30.1.9 Depth-/Breath-First Search

```
val dfs : ('n,'e,'g) graph  ->
          (node_id -> unit) ->
          ('e edge -> unit) ->
          node_id list -> unit
```

The function `dfs` takes as arguments a graph, a function `f : node_id -> unit`, a function `g : 'e edge -> unit`, and a set of source vertices. It performs depth first search on the graph. The function `f` is invoked whenever a new node is being visited, while the function `g` is invoked whenever a new edge is being traversed. This algorithm has running time $O(|V| + |E|)$.

```
val dfsfold : ('n,'e,'g) graph  ->
              (node_id * 'a -> 'a) ->
              ('e edge * 'b -> 'a) ->
              node_id list -> 'a * 'b -> 'a * 'b
val dfsnum :  ('n,'e,'g) graph  ->
              (node_id * 'a -> 'a) ->
               dfsnum : int array, compnum : int array
```

The function `bfs` is similar to `dfs` except that breath first search is performed.

```
val bfs : ('n,'e,'g) graph  ->
          (node_id -> unit) ->
          ('e edge -> unit) ->
          node_id list -> unit
val bfsdist : ('n,'e,'g) graph -> node_id list -> int array
```

30.1.10 Preorder/Postorder numbering

```
val preorder_numbering  : ('n,'e,'g) graph -> int -> int array
val postorder_numbering : ('n,'e,'g) graph -> int -> int array
```

Both these functions take a tree T and a root v, and return the preorder numbering and the postorder numbering of the tree respectively.

[179] **file:** `graphs/digraph.sml`
[180] **file:** `library/undoable-array.sml`

30.1.11 Topological Sort

```
val topsort : ('n,'e,'g) graph -> node_id list -> node_id list
```

The function `topsort` takes a graph G and a set of source vertices S as arguments. It returns a topological sort of all the nodes reachable from the set S. This algorithm has running time $O(|S| + |V| + |E|)$.

30.1.12 Strongly Connected Components

```
val strong_components : ('n,'e,'g) graph ->
  (node_id list * 'a -> 'a) -> 'a -> 'a
```

The function `strong_components` takes a graph G and an aggregate function f with type

```
node_id list * 'a -> 'a
```

and an identity element `x : 'a` as arguments. Function f is invoked with a strongly connected component (represented as a list of node ids) as each is discovered. That is, the function `strong_components` computes

$$f(SCC_n, f(SCC_{n-1}, \ldots, f(SCC_1, x)))$$

where $SCC_1, \ldots, SCC_n$ are the strongly connected components in topological order. This algorithm has running time $O(|V| + |E|)$.

30.1.13 Biconnected Components

```
val biconnected_components : ('n,'e,'g) graph ->
       ('e edge list * 'a -> 'a) -> 'a -> 'a
```

The function `biconnected_components` takes a graph G and an aggregate function f with type

```
'e edge list * 'a -> 'a
```

and an identity element `x : 'a` as arguments. Function f is invoked with a biconnected component (represented as a list of edges) as each is discovered. That is, the function `biconnected_components` computes

$$f(BCC_n, f(BCC_{n-1}, \ldots, f(BCC_1, x)))$$

where $BCC_1, \ldots, BCC_n$ are the biconnected components. This algorithm has running time $O(|V| + |E|)$.

30.1.14 Cyclic Test

```
val is_cyclic : ('n,'e,'g) graph -> bool
```

Function `is_cyclic` tests if a graph is cyclic. This algorithm has running time $O(|V| + |E|)$.

30.1.15 Enumerate Simple Cycles

```
val cycles : ('n,'e,'g) graph -> ('e edge list * 'a -> 'a) -> 'a ->'a
```

A simple cycle is a circuit that visits each vertex only once. The function `cycles` enumerates all simple cycles in a graph G. It takes as argument an aggregate function f of type

```
'e edge list * 'a -> 'a
```

and an identity element e, and computes as result the expression

$$f(c_n, f(c_{n-1}, f(c_{n-2}, \ldots, f(c_1, e))))$$

where $c_1, \ldots, c_n$ are all the simple cycles in the graph. All cycles $c_1, \ldots, c_n$ are guaranteed to be distinct. A cycle is represented as a sequence of adjacent edges, i.e. in the order of

$$v_1 -> v_2, v_2 -> v_3, v_3 -> v_4, \ldots, v_{n-1} -> v_n, v_n -> v_1$$

Our implementation works by first decomposing the graph into its strongly connected components, then uses backtracking to enumerate simple cycles in each component.

30.1.16 Minimal Cost Spanning Tree

```
signature MIN_COST_SPANNING_TREE[181] = sig
  exception Unconnected

  val spanning_tree : { weight    : 'e edge -> 'a,
                        <         : 'a * 'a -> bool
                      } -> ('n, 'e, 'g) graph
                        -> ('e edge * 'a -> 'a) -> 'a -> 'a
end
structure Kruskal[182] : MIN_COST_SPANNING_TREE
```

Structure `Kruskal` implements Kruskal's algorithm for computing a minimal cost spanning tree of a graph. The function `spanning_tree` takes as arguments:

- a `weight` function which when given an edge returns its weight
- an ordering function <, which is used to compare the weights
- a graph G
- an accumulator function f, and
- an identity element x

The function `spanning_tree` computes

$$f(e_n, f(e_{n-1}, \ldots, f(e_1, x)))$$

where $e_1, \ldots, e_n$ are the edges in a minimal cost spanning tree of the graph. The exception `Unconnected` is raised if the graph is unconnected.

[181] **file:** `graphs/spanning-tree.sig`
[182] **file:** `graphs/kruskal.sml`

30.1.17 Abelian Groups

Graph algorithms that deal with numeric weights or distances are parameterized with respect to the signatures `ABELIAN_GROUP` or `ABELIAN_GROUP_WITH_INF`. These are defined as follows:

```
signature ABELIAN_GROUP[183] = sig
  type elem
  val +    : elem * elem -> elem
  val -    : elem * elem -> elem
  val      : elem -> elem
  val zero : elem
  val <    : elem * elem -> bool
  val ==   : elem * elem -> bool
end
signature ABELIAN_GROUP_WITH_INF[184] = sig
  include ABELIAN_GROUP
  val inf : elem
end
```

Signature `ABELIAN_GROUP` specifies an ordered commutative group, while signature `ABELIAN_GROUP_WITH_INF` specifies an ordered commutative group with an infinity element `inf`.

30.1.18 Single Source Shortest Paths

```
signature SINGLE_SOURCE_SHORTEST_PATHS[185] = sig
  structure Num : ABELIAN_GROUP_WITH_INF
  val single_source_shortest_paths :
                { graph : ('n,'e,'g) graph,
                  weight : 'e edge -> Num.elem,
                  s : node_id
                } ->
                { dist : Num.elem array,
                  pred :  node_id array
                }
end
functor Dijkstra[186](Num : ABELIAN_GROUP_WITH_INF)
   : SINGLE_SOURCE_SHORTEST_PATHS
```

The functor `Dijkstra` implements Dijkstra's algorithm for single source shortest paths. The function `single_source_shortest_paths` takes as arguments:

- a graph G,
- a `weight` function on edges, and
- the source vertex s.

[183] **file:** `graphs/groups.sig`
[184] **file:** `graphs/groups.sig`
[185] **file:** `graphs/shortest-paths.sig`
[186] **file:** `graphs/dijkstra.sml`

It returns two arrays `dist` and `pred` indexed by vertices. These arrays have the following interpretation. Given a vertex v,

- `dist`[v] contains the distance of v from the source s
- `pred`[v] contains the predecessor of v in the shortest path from s to v, or -1 if $v = s$.

Dijkstra's algorithm fails to work on graphs that have negative edge weights. To handle negative weights, Bellman-Ford's algorithm can be used. The exception `NegativeCycle` is raised if a cycle of negative total weight is detected.

```
functor BellmanFord[187](Num : ABELIAN_GROUP_WITH_INF) : sig
   include SINGLE_SOURCE_SHORTEST_PATHS
   exception NegativeCycle
end
```

30.1.19 All Pairs Shortest Paths

```
signature ALL_PAIRS_SHORTEST_PATHS[188] = sig
  structure Num : ABELIAN_GROUP_WITH_INF
  val all_pairs_shortest_paths :
                { graph : ('n,'e,'g) graph,
                  weight : 'e edge -> Num.elem
                } ->
                { dist : Num.elem Array2.array,
                  pred :  node_id Array2.array
                }
end
functor FloydWarshall[189](Num : ABELIAN_GROUP_WITH_INF)
   : ALL_PAIRS_SHORTEST_PATHS
```

The functor `FloydWarshall` implements Floyd-Warshall's algorithm for all pairs shortest paths. The function `all_pairs_shortest_paths` takes as arguments:

- a graph G, and
- a `weight` function on edges

It returns two 2-dimensional arrays `dist` and `pred` indexed by vertices (u, v). These arrays have the following interpretation. Given a pair (u, v),

- `dist`[u, v] contains the distance from u to v.
- `pred`[u, v] contains the predecessor of v in the shortest path from u to v, or -1 if $u = v$.

This algorithm runs in time $O(|V|^3 + |E|)$.

An alternative implementation is available that uses Johnson's algorithm, which works better for sparse graphs:

[187] **file:** `graphs/bellman-ford.sml`
[188] **file:** `graphs/shortest-paths.sig`
[189] **file:** `graphs/floyd-warshall.sml`

```
functor Johnson[190](Num : ABELIAN_GROUP_WITH_INF)
   : sig include ALL_PAIRS_SHORTEST_PATHS
         exception Negative Cycle
     end
```

30.1.20 Transitive Closure

```
signature TRANSITIVE_CLOSURE[191] = sig
   val acyclic_transitive_closure :   + : ('e * 'e -> 'e), simple : bool
       -> ('n,'e,'g) graph -> unit
   val acyclic_transitive_closure2 :
      {  + : 'e * 'e -> 'e,
         max : 'e * 'e -> 'e
      }  -> ('n,'e,'g) graph -> unit
   val transitive_closure : ('e * 'e -> 'e) -> ('n,'e,'g) graph -> unit
structure TransitiveClosure[192] : TRANSITIVE_CLOSURE
```

Structure `TransitiveClosure` implements in-place transitive closures on graphs. Three functions are implemented. Functions `acyclic_transitive_closure` and `acyclic_transitive_closure2` can be used to compute the transitive closure of an acyclic graph, whereas the function `transitive_closure` computes the transitive closure of a cyclic graph. All take a binary function

```
+ : 'e * 'e -> 'e
```

defined on edge labels. Transitive edges are inserted in the following manner:

- `acyclic_transitive_closure`: given $u \rightarrow_l v$ and $v \rightarrow_{l'} w$, if the flag `simple` is false or if the transitive edge $u \rightarrow w$ does not exists, then $u \rightarrow_{l+l'} w$ is added to the graph.
- `acyclic_transitive_closure2`: given $u \rightarrow_l v$ and $v \rightarrow_{l'} w$, the transitive $u \rightarrow_{l+l'} w$ is added to the graph. Furthermore, all parallel edges

  $$u \rightarrow_{l_1} w, \ldots, u \rightarrow_{l_n} w$$

  are coalesced into a single edge $u \rightarrow_l w$, where $l = \mathtt{max}_{i=1\ldots n} l_i$

30.1.21 Max Flow

The function `max_flow` computes the maximum flow between the source vertex `s` and the sink vertex `t` in the `graph` when given a `capacity` function.

```
signature MAX_FLOW[193] = sig
  structure Num : ABELIAN_GROUP
  val max_flow : { graph    : ('n,'e,'g) graph,
                   s        : node_id,
                   t        : node_id,
```

[190] **file:** `graphs/johnson.sml`
[191] **file:** `graphs/trans-closure.sml`
[192] **file:** `graphs/trans-closure.sml`
[193] **file:** `graphs/max-flow.sig`

```
                  capacity : 'e edge -> Num.elem,
                  flows    : 'e edge * Num.elem -> unit
                } -> Num.elem
end
functor MaxFlow[194](Num : ABELIAN_GROUP) : MAX_FLOW
```

The function `max_flow` returns its result in the follow manner: The function returns the total flow as its result value. Furthermore, the function `flows` is called once for each edge e in the graph with its associated flow f_e.

This algorithm uses Goldberg's preflow-push approach, and runs in $O(|V|^2|E|)$ time.

30.1.22 Min Cut

The function `min_cut` computes the minimum (undirected) cut in a `graph` when given a `weight` function on its edges.

```
signature MIN_CUT[195] = sig
  structure Num : ABELIAN_GROUP
  val min_cut : { graph    : ('n,'e,'g) graph,
                  weight : 'e edge -> Num.elem
                } -> node_id list * Num.elem
end
functor MinCut[196](Num : ABELIAN_GROUP) : MIN_CUT
```

The function `min_cut` returns a list of node ids denoting one side of the cut C (the other side of the cut is $(V - C)$ and the weight cut.

30.1.23 Max Cardinality Matching

```
val matching : ('n,'e,'g) graph -> ('e edge * 'a -> 'a) -> 'a -> 'a * int
```

The function `BipartiteMatching.matching` computes the maximal cardinality matching of a bipartite graph. As result, the function iterates over all the matched edges and returns the number of matched edges. The algorithm runs in time $O(|V||E|)$.

30.1.24 Node Partition

```
signature NODE_PARTITION = sig
  type 'n node_partition

  val node_partition : ('n,'e,'g) graph -> 'n node_partition
  val !!    : 'n node_partition -> node_id -> 'n node
  val ==    : 'n node_partition -> node_id * node_id -> bool
  val union : 'n node_partition -> ('n node * 'n node -> 'n node) ->
                                      node_id * node_id -> bool
  val union': 'n node_partition -> node_id * node_id -> bool

end
```

[194] **file:** `graphs/max-flow.sml`
[195] **file:** `graphs/min-cut.sig`
[196] **file:** `graphs/min-cut.sml`

30.1.25 Node Priority Queue

```
signature NODE_PRIORITY_QUEUE = sig
  type node_priority_queue

  exception EmptyPriorityQueue

  val create         : (node_id * node_id -> bool) -> node_priority_queue
  val fromGraph      : (node_id * node_id -> bool) ->
     ('n,'e,'g) graph -> node_priority_queue
  val isEmpty        : node_priority_queue -> bool
  val clear          : node_priority_queue -> unit
  val min            : node_priority_queue -> node_id
  val deleteMin      : node_priority_queue -> node_id
  val decreaseWeight : node_priority_queue * node_id -> unit
  val insert         : node_priority_queue * node_id -> unit
  val toList         : node_priority_queue -> node_id list
end
```

30.2 Views

Simply put, a view is an alternative presentation of a data structure to a client. A graph, such as the control flow graph, frequently has to be presented in different ways in a compiler. For example, when global scheduling is applied on a region (a subgraph of the CFG), we want to be able to concentrate on just the region and ignore all nodes and edges that are not part of the current focus. All transformations that are applied on the current region view should be automatically reflected back to the entire CFG as a whole. Furthermore, we want to be able to freely intermix graphs and subgraphs of the same type in our program, without having to introducing sums in our type representations.

The `subgraph_view` view combinator accomplishes this. `Subgraph` takes a list of nodes and produces a graph object which is a view of the node induced subgraph of the original graph. All modification to the subgraph are automatically reflected back to the original graph. From the client point of view, a graph and a subgraph are entirely indistinguishable, and furthermore, graphs and subgraphs can be freely mixed together (they are the same type from ML's point of view.)

This transparency is obtained by selective method overriding, composition, and delegation. For example, a generic graph object provides the following methods for setting and looking up the entries and exits from a graph.

```
set_entries  : node_id list -> unit
set_exits    : node_id list -> unit
entries      : unit -> node_id list
exits        : unit -> node_id list
```

For example, a CFG usually has a single entry and a single exit. These methods allow the client to destinate one node as the entry and another as the exit. In the case of subgraph view, these methods are overridden so that the proper conventions are preserved: a node in a subgraph is an entry (exit) iff there is an in-edge (out-edge) from (to) outside the (sub-)graph. Similarly, the methods `entry_edges` and `exit_edges` can be used return the entry and exit edges associated with a node in a subgraph.

```
entry_edges  : node_id -> 'e edge list
exit_edges   : node_id -> 'e edge list
```

These methods are initially defined to return `[]` in a graph and subsequently overridden in a subgraph.

30.2.1 Update Transparency

Suppose a view G' is created from some base graphs or views. *Update transparency* refers to the fact that G' behaves consistently according to its conventions and semantics when updates are performed. There are 4 different type of update transparencies:

- *update opaque* A update opaque view disallows updates to both itself and its base graphs.
- *globally update transparent* A globally update transparent view allows updates to its base graphs but not to itself. Changes will then be automatically reflected in the view.
- *locally update transparent* A locally update transparent view allows updates to itself but not to its base graphs. Changes will be automatically reflected to the base graphs.
- *fully update transparent* A fully update transparent view allows updates through its methods or through its base graphs'.

30.2.2 Structural Views

30.2.3 Reversal

```
val ReversedGraphView.rev_view[197] : ('n,'e,'g) graph -> ('n,'e,'g) graph
```

This combinator takes a graph G and produces a view G^R which reverses the direction of all its edges, including entry and exit edges. Thus the edge $i \rightarrow_l j$ in G becomes the edge $j \rightarrow_l i$ in G^R. This view is fully update transparent.

30.2.4 Readonly

```
val ReadOnlyGraphView.readonly_view[198] : ('n,'e,'g) graph -> ('n,'e,'g) graph
```

This function takes a graph G and produces a view G' in which no mutator methods can be used. Invoking a mutator method raises the exception `Readonly`. This view is globally update transparent.

30.2.5 Snapshot

```
functor GraphSnapShot[199](GI : GRAPH_IMPLEMENTATION) : GRAPH_SNAPSHOT
signature GRAPH_SNAPSHOT = sig
   val snapshot : ('n,'e,'g) graph ->
     { picture : ('n,'e,'g) graph, button : unit -> unit }
end
```

The function `snapshot` can be used to keep a cached copy of a view a.k.a the `picture`. This cached copy can be updated locally but the modification will not be reflected back to the base graph. The function `button` can be used to keep the view and the base graph up-to-date.

[197] **file:** `graphs/revgraph.sml`
[198] **file:** `graphs/readonly.sml`
[199] **file:** `graphs/snap-shot.sml`

30.2.6 Map

```
val IsomorphicGraphView.map[200] :
  ('n node -> 'n') -> ('e edge -> 'e') -> ('g -> 'g') ->
  ('n,'e,'g) graph -> ('n','e','g') graph
```

The function `map` is a generalization of the `map` function on lists. It takes three functions

```
f : 'n node -> 'n
g : 'e edge -> 'e
h : 'g -> g'
```

and a graph $G = (V, L, E, I)$ as arguments. It computes the view $G' = (V, L', E', I')$ where

$$\begin{aligned} L'(v) &= f(v, L(v)) \text{ for all } v \in V \\ E' &= i \rightarrow_{g(i,j,l)} j | i \rightarrow_l j \in E \\ I' &= h(I) \end{aligned}$$

30.2.7 Singleton

```
val SingletonGraphView.singleton_view[201] : ('n,'e,'g) graph -> node_id -> ('n,'e,'g) graph
```

Function `singleton_view` takes a graph G and a node id v (which must exists in G) and return an edge-free graph with only one node (v). This view is opaque.

30.2.8 Node id renaming

```
val RenamedGraphView.rename_view[202] : int -> ('n,'e,'g) graph -> ('n','e','g') graph
```

The function `rename_view` takes an integer n and a graph G and create a fully update transparent view where all node ids are incremented by n. Formally, given graph $G = (V, E, L, I)$ it computes the view $G' = (V', E', L', I)$ where

$$\begin{aligned} V' &= v + n | v \in V \\ E' &= i + n \rightarrow_l j + n | i \rightarrow_l j \in E \\ L' &= \lambda v.L(v - n) \end{aligned}$$

30.2.9 Union and Sum

```
val UnionGraphView.union_view[203] : ('g * 'g') -> 'g'') ->
   ('n,'e,'g) graph * ('n,'e,'g') graph -> ('n','e','g'') graph
GraphCombinations.unions : ('n,'e,'g) graph list -> ('n,'e,'g) graph
GraphCombinations.sum : ('n,'e,'g) graph * ('n,'e,'g) graph -> ('n,'e,'g) graph
GraphCombinations.sums : ('n,'e,'g) graph list -> ('n,'e,'g) graph
```

[200] **file:** `graphs/isograph.sml`
[201] **file:** `graphs/singleton.sml`
[202] **file:** `graphs/renamegraph.sml`
[203] **file:** `graphs/uniongraph.sml`

Function `union_view` takes as arguments a function f, and two graphs $G = (V, L, E, I)$ and $G' = (V', L', E', I')$, it computes the union $G + G'$ of these graphs. Formally, $G \cup G' = (V'', L'', E'', I'')$ where

$$\begin{aligned} V'' &= V \cup V' \\ L'' &= L overrides L' \\ E'' &= E \cup E' \\ I'' &= f(I, I') \end{aligned}$$

The function `sum` constructs a *disjoint sum* of two graphs.

30.2.10 Simple Graph View

```
val SimpleGraph.simple_graph[204] : (node_id * node_id * 'e list -> 'e) ->
 ('n,'e,'g) graph -> ('n,'e,'g) graph
```

Function `simple_graph` takes a merge function f and a multi-graph G as arguments and return a view in which all parallel multi-edges (edges with the same source and target) are combined into a single edge: i.e. any collection of multi-edges between the same source s and target t and with labels $l_1, \ldots, l_n$, are replaced by the edge $s \rightarrow_{l_{st}} t$ in the view, where $l_{st} = f(s, t, [l_1, \ldots, l_n])$. The function f is assumed to satisfy the equality $l = f(s, t, [l])$ for all l, s and t.

30.2.11 No Entry or No Exit

```
val NoEntryView.no_entry_view[205] : ('n,'e,'g) graph -> ('n,'e,'g) graph
NoEntryView.no_exit_view : ('n,'e,'g) graph -> ('n,'e,'g) graph
```

The function `no_entry_view` creates a view in which all entry edges (and thus entry nodes) are removed. The function `no_exit_view` is the dual of this and creates a view in which all exit edges are removed. This view is fully update transparent. It is possible to remove all entry and exit edges by composing these two functions.

30.2.12 Subgraphs

```
val SubgraphView.subgraph_view[206] : node_id list -> ('e edge -> bool) ->
  ('n,'e,'g) graph -> (n','e','g') graph
```

The function `subgraph_view` takes as arguments a set of node ids S, an edge predicate p and a graph $G = (V, L, E, I)$. It returns a view in which only the visible nodes are S and the only visible edges e are those that satisfy $p(e)$ and with sources and targets in S. S must be a subset of V.

```
val Subgraph_P_View.subgraph_p_view[207] : node_id list ->
  (node_id -> bool) -> (node_id * node_id -> bool) ->
  ('n,'e,'g) graph -> ('n','e','g') graph
```

The function `subgraph_view` takes as arguments a set of node ids S, a node predicate p, an edge predicate q and a graph $G = (V, L, E, I)$. It returns a view in which only the visible nodes v are those in S satisfying $p(v)$, and the only visible edges e are those that satisfy $q(e)$ and with sources and targets in S. S must be a subset of V.

[204] **file:** `graphs/simple-graph.sml`
[205] **file:** `graphs/no-exit.sml`
[206] **file:** `graphs/subgraph.sml`
[207] **file:** `graphs/subgraph-p.sml`

30.2.13 Trace

```
val TraceView.trace_view[208] : node_id list -> ('n,'e,'g) graph -> ('n','e','g') graph
```

A *trace* is an acyclic path in a graph. The function `trace_view` takes a trace of node ids $v_1, \ldots, v_n$ and a graph G and returns a view in which only the nodes are visible. Only the edges that connected two adjacent nodes on the trace, i.e. $v_i -> v_{i+1}$ for some $i = 1 \ldots n-1$ are considered be within the view. Thus if there is an edge $v_i -> v_j$ in G where $j \neq i+1$ this edge is not considered to be within the view — it is considered to be an exit edge from v_i and an entry edge from v_j however. Trace views can be used to construct a CFG region suitable for trace scheduling [Fis81, Ell85].

Figure 1 illustrates this concept graphically. Here, the trace view is formed from the nodes `A, C, D, F` and `G`. The solid edges linking the trace is visible within the view. All other dotted edges are considered to be either entry of exit edges into the trace. The edge from node `G` to `A` is considered to be both since it exits from `G` and enters into `A`.

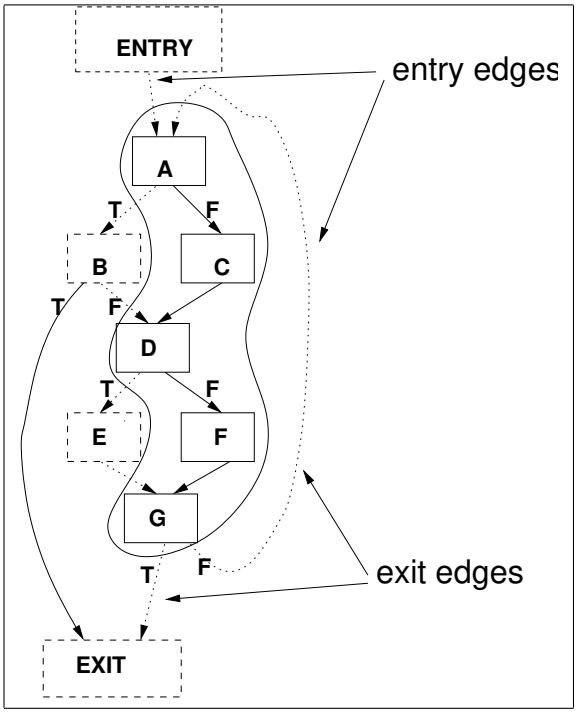


Figure 1: A trace view

30.2.14 Acyclic Subgraph

```
val AcyclicSubgraphView.acyclic_view[209] :
  node_id list ->
  ('n,'e,'g) graph -> ('n,'e,'g) graph
```

The function `acyclic_view` takes an ordered list of node ids $v_1, \ldots, v_n$ and a graph G as arguments and return a view G' such that only the nodes $v_1, \ldots, v_n$ are visible. In addition, only the edges with directions consistent with the order list are considered to be within the view. Thus an edge $v_i -> v_j$ from G is in G' iff $1 \leq i < j \leq n$. Acyclic views can be used to construct a CFG region suitable for DAG scheduling. Figure 2 illustrates this concept graphically.

30.2.15 Start and Stop

```
val StartStopView.start_stop_view[210] :
  { start : 'n node,
    stop  : 'n node,
    edges : 'e edge list
  } -> ('n,'e,'g) graph -> ('n','e','g') graph
```

[208] **file:** `graphs/trace-graph.sml`
[209] **file:** `graphs/acyclic-graph.sml`
[210] **file:** `graphs/start-stop.sml`

The function `start_stop_view`

30.2.16 Single-Entry/Multiple-Exits

```
SingleEntryMultipleExit.SEME[211]
  exit : 'n node -> ('n,'e,'g) graph -> ('n,'e,'g) graph
```

The function `SEME` converts a single-entry/multiple-exits graph G into a single entry/single exit graph. It takes an exit node e and a graph G and returns a view G'. Suppose $i \rightarrow_l j$ is an exit edge in G. In view G this edge is replaced by a new normal edge $i \rightarrow_l e$ and a new exit edge $e \rightarrow_l j$. Thus e becomes the sole exit node in the new view.

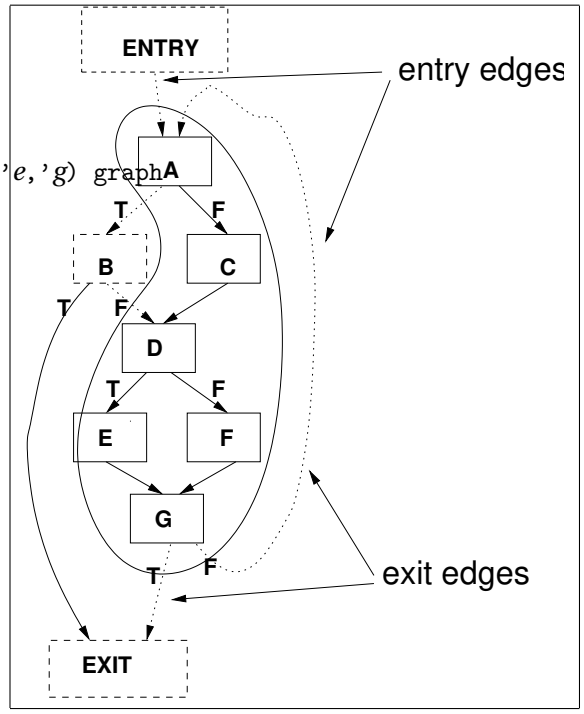


Figure 2: An acyclic subgraph

30.2.17 Behavioral Views

30.2.18 Behavioral Primitives

Figure 3 lists the set of behavioral primitives defined in structure `GraphWrappers`[212]. These functions allow the user to attach an action a to a mutator method m such that whenever m is invoked so does a. Given a graph G, the combinator

```
do_before_xxx : f -> ('n,'e,'g) graph -> ('n,'e,'g) graph
```

returns a view G' such that whenever method xxx is invoked in G', the function f is called. Similarly, the combinator

```
do_after_xxx : f -> ('n,'e,'g) graph -> ('n,'e,'g) graph
```

creates a new view G'' such that the function f is called after the method is invoked.

Frequently it is not necessary to know precisely by which method a graph's structure has been modified, only that it is. The following two methods take a notification function f and returns a new view. f is invoked before a modification is attempted in a view created by `do_before_changed`. It is invoked after the modification in a view created by `do_after_changed`.

```
do_before_changed : (('n,'e,'g) graph -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_after_changed : (('n,'e,'g) graph -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
```

Behavioral views created by the above functions are all fully update transparent.

[211] **file:** `graphs/SEME.sml`
[212] **file:** `graphs/wrappers.sml`

```
do_before_new_id : (unit -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_after_new_id : (node_id -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_before_add_node : ('n node -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_after_add_node : ('n node -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_before_add_edge : ('e edge -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_after_add_edge : ('e edge -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_before_remove_node : (node_id -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_after_remove_node : (node_id -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_before_set_in_edges : (node_id * 'e edge list -> unit) ->
   ('n,'e,'g) graph -> ('n,'e,'g) graph
do_after_set_in_edges : (node_id * 'e edge list -> unit) ->
   ('n,'e,'g) graph -> ('n,'e,'g) graph
do_before_set_out_edges : (node_id * 'e edge list -> unit) ->
   ('n,'e,'g) graph -> ('n,'e,'g) graph
do_after_set_out_edges : (node_id * 'e edge list -> unit) ->
   ('n,'e,'g) graph -> ('n,'e,'g) graph
do_before_set_entries : (node_id list -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_after_set_entries : (node_id list -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_before_set_exits : (node_id list -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
do_after_set_exits : (node_id list -> unit) -> ('n,'e,'g) graph -> ('n,'e,'g) graph
```

Figure 3: Behavioral view primitives

31 The Graph Visualization Library

31.1 Overview

Visualization is an important aid for debugging graph algorithms. MLRISC provides a simple facility for displaying graphs that adheres to the graph interface. Two graph viewer back-ends are currently supported. (An interface to the *dot* tool is still available but is unsupported.)

- vcg[213] – this tool supports the browsing of hierarchical graphs, zoom in/zoom out functions. It can handle up to around 5000 nodes in a graph.
- daVinci[214] – this tool supports a separate "survey" view from the main view and text searching. This tool is slower than vcg but it has a nicer interface, and can handle up to around 500 nodes in a graph.

All graph viewing back-ends work in the same manner. They take a graph whose nodes and edges are annotated with *layout* instructions and translate these layout instructions into the target description language. For vcg, the target description language is GDL. For daVinci, it is a language based on s-expressions.

31.2 Graph Layout

Some basic layout formats are defined structure `GraphLayout` are:

```
structure GraphLayout[215] = struct
  datatype format =
    LABEL of string
  | COLOR of string
  | NODE_COLOR of string
  | EDGE_COLOR of string
  | TEXT_COLOR of string
  | ARROW_COLOR of string
  | BACKARROW_COLOR of string
  | BORDER_COLOR of string
  | BORDERLESS
  | SHAPE of string
  | ALGORITHM of string
  | EDGEPATTERN of string

  type ('n,'e,'g) style =
     { edge  : 'e edge -> format list,
       node  : 'n node -> format list,
       graph : 'g -> format list
     }
  type layout = (format list, format list, format list) graph
end
```

The interpretation of the layout formats are as follows:

[213] url: `http://www.cs.uni-sb.de/RW/users/sander/html/gsvcg1.html`
[214] url: `http://www.Informatik.Uni-Bremen.DE/ davinci/`
[215] **file:** `visualization/graphLayout.sml`

LABEL l	Label a node or an edge with the string l
COLOR c	Use color c for a node or an edge
NODE_COLOR c	Use color c for a node
EDGE_COLOR c	Use color c for an edge
TEXT_COLOR c	Use color c for the text within a node
ARROW_COLOR c	Use color c for the arrow of an edge
BACKARROW_COLOR c	Use color c for the arrow of an edge
BORDER_COLOR c	Use color c for the border in a node
BORDERLESS	Disable border for a node
SHAPE s	Use shape s for a node
ALGORITHM a	Use algorithm a to layout the graph
EDGEPATTERN p	Use pattern p to layout an edge

Exactly how these formats are interpreted is determined by the visualization tool that is used. If a feature is unsupported then the corresponding format will be ignored. Please see the appropriate reference manuals of vcg and daVinci for details.

31.3 Layout style

How a graph is layout is determined by its *layout style*:

```
type ('n,'e,'g) style =
   { edge  : 'e edge -> format list,
     node  : 'n node -> format list,
     graph : 'g -> format list
   }
```

which is simply three functions that convert nodes, edges and graph info into layout formats. The function `makeLayout` can be used to convert a layout style into a layout, which can then be passed to a graph viewer to be displayed.

```
GraphLayout.makeLayout : ('n,'e,'g) style -> ('n,'e,'g) graph -> layout
```

31.4 Graph Displays

A *graph display* is an abstraction for the interface that converts a layout graph into an external graph description language. This abstraction is defined in the signature below.

```
signature GRAPH_DISPLAY[216] = sig
  val suffix    : unit -> string
  val program   : unit -> string
  val visualize : (string -> unit) -> GraphLayout.layout -> unit
end
```

- `suffix` is the common file suffix used for the graph description language
- `program` is the common name of the graph visualization tool
- `visualize` is a function that takes a string output function and a layout graph G as arguments and generates a graph description based on G

[216] **file:** `visualization/graphDisplay.sig`

31.5 Graph Viewers

The graph viewer functor `GraphViewer`[217] takes a graph display back-end and creates a graph viewer that can be used to display any layout graph.

```
signature GRAPH_VIEWER[218] = sig
   val view : GraphLayout.layout -> unit
end
functor GraphViewer(D : GRAPH_DISPLAY) : GRAPH_VIEWER
```

[217] **file:** `visualization/graphViewer.sml`
[218] **file:** `visualization/graphViewer.sig`

32 Basic Compiler Graphs

32.1 Introduction

In this section we describe the set of core compiler specific graphs and algorithms implemented in ML-RISC. Mostly of these algorithms are parameterized with respect to the actual intermediate representation, and as such they do not provide many facilities that are provided by higher abstraction layers, such as in MLRISC IR[219], or in SSA[220].

32.1.1 Dominator/Post-dominator Trees

Dominance is a fundamental concept in compiler optimizations. Node A $dominates$ B iff all paths from the start node to B intersects A. A dual notion is the concept of $post-dominance$: A ***post-dominates*** B iff all paths from B to the stop node intersects A. A (post-)dominator tree can be used to summarize the dominance/post-dominance relationship.

```
functor DominatorTree[221]
   (GraphImpl : GRAPH_IMPLEMENTATION) : DOMINATOR_TREE
```

The functor implements dominator analysis and creates a dominator/post-dominator tree from a graph G. A dominator tree is implemented as a graph with the following definition:

```
signature DOMINATOR_TREE[222] = sig
   exception Dominator
   datatype 'n dom_node =
      DOM of { node : 'n, level : int, preorder : int, postorder : int }
   type ('n,'e,'g) dom_info
   type ('n,'e,'g) dominator_tree = ('n dom_node,unit,('n,'e,'g) dom_info) graph
   type ('n,'e,'g) postdominator_tree = ('n dom_node,unit,('n,'e,'g) dom_info) graph
```

We annotated each node in a dominator tree with three extra fields of information, which is useful for other algorithms:

- `level` is the nesting level of the tree. The root node has level 0, children of the root has level 1 and so on.
- `preorder` is the preorder numbering of a node
- `preorder` is the postorder numbering of a node.

To create a dominator tree and a postdominator tree from a graph, the following function should be called.

```
val dominator_trees : ('n,'e,'g) graph ->
        ('n,'e,'g) dominator_tree * ('n,'e,'g) postdominator_tree
```

[219] url: `mlrisc-ir.html`
[220] url: `SSA.html`
[221] **file:** `ir/dominator.sml`
[222] **file:** `ir/dominator.sig`

We use the algorithm of Tarjan and Lengauer, which runs in time $O(|V+E|\alpha(|V+E|))$ where α is the functional inverse of the Ackermann function.

To perform many common queries on a dominator tree, we first call the function `methods` to obtain a method object.

```
val methods : ('n,'e,'g) dominator_tree -> dominator_methods
```

The methods are packed into the following type:

```
type dominator_methods =
      { dominates              : node_id * node_id -> bool,
        immediately_dominates  : node_id * node_id -> bool,
        strictly_dominates     : node_id * node_id -> bool,
        postdominates          : node_id * node_id -> bool,
        immediately_postdominates : node_id * node_id -> bool,
        strictly_postdominates : node_id * node_id -> bool,
        control_equivalent     : node_id * node_id -> bool,
        idom         : node_id -> node_id, $(* ~1 if none *)$
        idoms        : node_id -> node_id list,
        doms         : node_id -> node_id list,
        ipdom        : node_id -> node_id, $(* ~1 if none *)$
        ipdoms       : node_id -> node_id list,
        pdoms        : node_id -> node_id list,
        dom_lca      : node_id * node_id -> node_id,
        pdom_lca     : node_id * node_id -> node_id,
        dom_level    : node_id -> int,
        pdom_level   : node_id -> int,
        control_equivalent_partitions : unit -> node_id list list
      }
```

The query methods are as follows:

`dominates`(a,b)	returns true iff a dominates b
`immediately_dominates`(a,b)	returns true iff a immediately dominates b
`strictly_dominates`(a,b)	returns true iff a strictly dominates b
`postdominates`(a,b)	returns true iff a post-dominates b
`immediately_postdominates`(a,b)	returns true iff a immediately post-dominates b
`strictly_postdominates`(a,b)	returns true iff a strictly post-dominates b
`control_equivalent`(a,b)	returns true iff a dominates b and vice versa
`idom`(a)	returns the immediate dominator of a, or -1 if none exists
`idoms`(a)	returns all nodes that a immediately dominates
`doms`(a)	returns all nodes that a dominates (including a itself)
`ipdom`(a)	returns the immediate post-dominator of a, or -1 if none exists
`ipdoms`(a)	returns all nodes that a immediately post-dominates
`pdoms`(a)	returns all nodes that a post-dominates (including a itself)
`dom_lca`(a,b)	returns the least common ancestor of a and b in the dominator tree
`pdom_lca`(a,b)	returns the least common ancestor of a and b in the post-dominator tree
`dom_level`(a)	returns the nesting level of a in the dominator tree
`pdom_level`(b)	returns the nesting level of a in the post-dominator tree
`control_equivalent_partitions`	partitions the graph into a set of control equivalent nodes.

The methods `dom_lca`, `pdom_lca` and `control_equivalent_partitions` executes in $O(n)$ time, where n is the size of the dominator tree. The other methods run in $O(1)$ time.

32.1.2 Control Dependence Graph

Given two nodes A and B in a control flow graph G, we say that B is *control dependent* on A iff

- B post-dominates a successor of A
- B does not strictly post-dominates A

Intuitively, B is control dependent on A means that some path in the program that goes through A can by-passed B, and furthermore, A is the point in which this divergence can occur. Control dependence is used to various kinds of analysis and optimizations in a compiler, such as code motion and global scheduling [BR91].

To build a control dependence graph, the functor `ControlDependenceGraph` can be used:

```
signature CONTROL_DEPENDENCE_GRAPH[223] = sig
   type ('n,'e,'g) cdg = ('n,'e,'g) graph

   val control_dependence_graph :
         ('e -> bool) ->
         ('n,'e,'g) dominator_tree *
         ('n,'e,'g) postdominator_tree ->
         ('n,'e,'g) cdg
end
functor ControlDependenceGraph[224]
   (structure Dom : DOMINATOR_TREE
    structure GraphImpl : GRAPH_IMPLEMENTATION
   ) : CONTROL_DEPENDENCE_GRAPH
```

The control depedence graph is a subcomponent of the program dependence graph commonly used in modern compiler optimizations.

32.1.3 Dominance Frontiers

Many algorithms involving the notion of control dependence or dominance can be rephrased in terms of *dominance frontiers*. A node A is in the dominance frontiers of B iff B dominates a predecessor of A but B does not strictly-dominate A. We denote this as $A \in DF(B)$. The dual notion of *post-dominance frontiers* can be defined analogously using the post-dominator tree[225].

```
functor DominanceFrontiers[226](Dom : DOMINATOR_TREE) : DOMINANCE_FRONTIERS
```

The functor `DominanceFrontiers` can be used to compute all the dominance frontiers of all the nodes in a graph. It has the following signature.

[223] **file:** `ir/cdg.sig`
[224] **file:** `ir/cdg.sml`
[225] Control dependence can be defined in terms of post-dominance frontiers.
[226] **file:** `ir/dominance-frontier.sml`

```
signature DOMINANCE_FRONTIERS[227] = sig
  structure Dom : DOMINATOR_TREE
  type dominance_frontiers = node_id list array
  val DFs : ('n,'e,'g) Dom.dominator_tree -> dominance_frontiers
end
```

32.1.4 Iterated Dominance Frontiers

Iterated dominance frontiers (denoted as DF^+) are defined as the least fixed point of iterating the operation DF. Formally, define the dominance frontiers on a set S as follows:

$$DF(S) \stackrel{\text{as}}{=} \bigcup_{A \in S} DF(A)$$

Define iteration of DF, denoted as DF^n, as follows:

$$\begin{aligned} DF^1(S) &\stackrel{\text{as}}{=} DF(S) \\ DF^{n+1}(S) &\stackrel{\text{as}}{=} DF(S \cup DF^n(S)) \end{aligned}$$

The iterated dominance frontiers $DF^+(S)$ on a set S are defined as the limit:

$$DF^+(S) \stackrel{\text{as}}{=} \lim_{n \to \infty} DF^n(S)$$

Iterated dominance frontiers of a set S can be computed in time $O(|S|+|V|+|E|)$ using the algorithm by Sreedhar and Gao [SG95][228].

```
functor DJGraph[229](Dom : DOMINATOR_TREE) : DJ_GRAPH
```

The functor `DJGraph` implements this algorithm. It satisfies the signature below:

```
signature DJ_GRAPH[230] = sig
   structure Dom : DOMINATOR_TREE
   type ('n,'e,'g) dj_graph = ('n,'e,'g) Dom.dominator_tree
   val dj_graph : ('n,'e,'g) dj_graph ->
       { DF   : node_id -> node_id list,
         IDF  : node_id -> node_id list,
         IDFs : node_id list -> node_id list
       }
end
```

The function `dj_graph` takes a dominator tree and returns three query methods for computing dominance and iterated dominance frontiers. Method `DF` computes $DF(v)$ for a single node v. Method `IDF` computes the $DF^+(v)$, and method `IDFs` computes $DF^+(S)$ when given a set of node ids. The dominator tree must not be updated while these operations are being performed.

[227] **file:** `ir/dominance-frontier.sig`
[228] In practice it is often sub-linear in $|V|+|E|$.
[229] **file:** `ir/djgraph.sml`
[230] **file:** `ir/djgraph.sig`

Sreedhar's original algorithm is phrased in terms of the DJ-graph, which is a fusion of the dominator tree with its underlying flowgraph. Our variant operates on the dominator tree and the flowgraph at the same time, without building an intermediate data structure.

Iterated dominance frontiers are used in many algorithms that deal with the notion of dominance. For example, our SSA construction algorithm uses iterated dominance frontiers to identify confluent points in the program where phi-functions are to be placed.

32.1.5 Loop Nesting Tree

A *natural loop* L in a graph is a maximal strongly connected component such that all nodes in L are dominated by a single node h, called the *loop header*. Loops tend to form good optimization candidates and consequently *loop detection* is an essential task in a compiler. The functor

```
functor LoopStructure[231]
  (structure GraphImpl : GRAPH_IMPLEMENTATION
   structure Dom       : DOMINATOR_TREE
  ) : LOOP_STRUCTURE
```

recognizes all natural loops in a graph and built a *loop nesting tree* that describes the loop nesting relationship between graphs.

```
signature LOOP_STRUCTURE[232] = sig
  structure Dom : DOMINATOR_TREE
  datatype ('n,'e,'g) loop =
     LOOP of { nesting    : int,
               header     : node_id,
               loop_nodes : node_id list,
               backedges  : 'e edge list,
               exits      : 'e edge list
             }

  type ('n,'e,'g) loop_info
  type ('n,'e,'g) loop_structure = (('n,'e,'g) loop,unit, ('n,'e,'g) loop_info) graph

  val loop_structure : ('n,'e,'g) Dom.dominator_tree -> ('n,'e,'g) loop_structure
  val nesting_level : ('n,'e,'g) loop_structure -> node_id array
  val header        : ('n,'e,'g) loop_structure -> node_id array
end
```

Our algorithm computes the loop nesting tree in time $O((|V|+|E|)\alpha(|V|+|E|))$. Each node in this tree represents a loop in the flowgraph, except the root of the tree, which represents the entire graph. Given a flowgraph G, the root of the loop nesting tree is defined to be the sole vertex in `#entry` G. Other nodes in the tree are indexed by the loop header node ids.

Loop detection classifies each loop and for each loop L, the following information is obtained:

- An integer `nesting`. The root of the tree has nesting depth 0. The top level loops have nesting depth 1, etc.

[231] **file:** `ir/loop-structure.sml`
[232] **file:** `ir/loop-structure.sig`

- The node id of the loop `header` h.
- A set of `loop_nodes`. Loop nodes are nodes that are in the strongly connected component L, but excluding the header h and all nodes that are part of any nested loops. Thus all nodes are uniquely partitioned in header nodes and loop nodes, and loop nodes are further partitioned into different sets according to which headers they are immediately nested under.
- A set of `backedges`. A back-edge is an edge that targets the header h and originates from a loop node in L.
- A set of loop `exits`. An exit-edge is an edge that originates from a loop node within L targets a node outside of L. Note that this set does not include any exit-edges contained in loops nested in L but target a node out of L.

32.1.6 Static Single Assignment

An SSA construction algorithm based on [CFR+89, BCHS88, SG95] is implemented in the following functor:

```
functor StaticSingleAssignmentForm[233]
   (Dom : DOMINATOR_TREE) : STATIC_SINGLE_ASSIGNMENT_FORM
```

SSA-based optimizations in MLRISC are actually implemented on top of a high-level SSA layer described in Section 34. So it is not necessary to use this module directly. Nevertheless, there can be situations in which this module can be specialized in other ways; for example, in the construction of sparse evaluation graphs.

```
signature STATIC_SINGLE_ASSIGNMENT_FORM[234] = sig
  structure Dom : DOMINATOR_TREE
  type var     = int
  type phi  = var * var * var list $(* orig def/def/uses *)$
  type renamer = {defs : var list, uses: var list} ->
                 {defs : var list, uses: var list}
  type copy    = {dst : var list, src: var list} -> unit

  val compute_ssa :
      ('n,'e,'g) Dom.dominator_tree ->
      { max_var      : var,
        defs         : 'n node -> var list,
        is_live      : var * int -> bool,
        rename_var   : var -> var,
        rename_stmt  : {rename:renamer,copy:copy} -> 'n node -> unit,
        insert_phi   : {block    : 'n node,
                        in_edges : 'e edge list,
                        phis     : phi list
                       } -> unit
      } -> unit
end
```

[233] **file:** `ir/ssa.sml`
[234] **file:** `ir/ssa.sig`

This module defines the function `compute_ssa`, which constructs an SSA graph. It requires the following information from the client:

- A dominator tree of the flowgraph.
- `max_var` – the maximum variable id (integer) that exists in the flowgraph. All variables are assumed to be indexed by non-negative integers.
- `defs`(X) – a function that returns $defs(X)$, i.e. the set of variable names defined in block X. If a minimal SSA form is desired, this set should include all the definitions in X. If a pruned SSA form is required, this set should include only the set of names that are live-out in X.
- `is_live`(v, X) – a function that determines if variable v is live-in into block X. If not, a ϕ-function will not be placed in X. For example, to compute the minimal-SSA form, this function should always return true.
- `rename_var`(v) – a function that returns a new unique name for variable v.
- `rename_stmt` – a function of type `rename:renamer,copy:copy -> 'n node -> unit` where

```
type renamer = {defs : var list, uses: var list} ->
               {defs : var list, uses: var list}
type copy    = {dst : var list, src: var list} -> unit
```

  Function `rename_stmt` is called for each block in the flowgraph in the order of the dominator tree, and is responsible for renaming all the variables in X by calling the functions `renamer` or `copy`. Function `renamer` renames all definitions and uses of a statement, while function `copy` renames of a set of parallel assignments

- `insert_phi`(X,es,$phis$) – a function that inserts a set of ϕ-definitions $phis$ in block X, where es is the list of control flow edges that merge into block X.

32.1.7 IDEFS/IUSE sets

Reif and Tarjan define the following useful notions for computing approximate birth-points for expressions, which in turn can be used to drive other optimizations. Given a node X, let $idom(X)$ denote the immediate dominator of X. Let $def(X)$ ($use(X)$) denote all the definitions (uses) in X. Given a path $p \equiv v_1 \ldots v_n$, define $def(p)$ ($use(p)$) as

$$\begin{aligned} def(v_1 \ldots v_n) &\equiv \cup_{i \in 1 \ldots n} def(v_i) \\ use(v_1 \ldots v_n) &\equiv \cup_{i \in 1 \ldots n} use(v_i) \end{aligned}$$

Let $P(X)$ denotes all the paths from $idom(X)$ to X that does not cross $idom(X)$ internally. Then define $idef(X)$ ($iuse(X)$) as:

$$\begin{aligned} idef(X) &\equiv \bigcup_{idom(X) v_1 \ldots v_n X \in P(X)} def(v_1 \ldots v_n) \\ iuse(X) &\equiv \bigcup_{idom(X) v_1 \ldots v_n X \in P(X)} use(v_1 \ldots v_n) \end{aligned}$$

The sets $ipostdef(X)$ and $ipostuse(X)$ are defined analogously using the postdominator tree.

```
signature IDEFS[235] = sig
  type var = int
  val compute_idefs :
      {def_use : 'n Graph.node -> var list * var list,
       cfg     : ('n,'e,'g) Graph.graph
      } ->
      { idefuse      : unit -> (RegSet.regset * RegSet.regset) Array.array,
        ipostdefuse  : unit -> (RegSet.regset * RegSet.regset) Array.array
      }
end
structure IDefs[236] : IDEFS
```

Structure `IDefs` implements the function `comput_idefs` for computing the $idef$, $iuse$, $ipostdef$ and $ipostuse$ sets of a control flow graph. It takes as arguments a flowgraph and a function `def_use`, which takes a graph node and returns the def/use sets of the node. It returns two functions `idefuse` and `ipostdefuse` which compute the $idef/iuse$ and $ipostdef/ipostuse$ sets. These sets are returned as arrays indexed by node ids.

[235] **file:** `ir/idefs2.sig`
[236] **file:** `ir/idefs2.sml`

33 The MLRISC IR

33.1 Introduction

In this section we will describe the MLRISC intermediate representation.

33.1.1 Control Flow Graph

The control flow graph is the main view of the IR. A control flow graph satisfies the following signature:

```
signature CONTROL_FLOW_GRAPH[237] = sig
  structure I : INSTRUCTIONS
  structure P : PSEUDO_OPS
  structure C : CELLS
  structure W : FIXED_POINT
     sharing I.C = C

  definitions
end
```

The following structures nested within a CFG:

- `I : INSTRUCTIONS` is the instruction structure.
- `P : PSEUDO_OPS` is the structure with the definition of pseudo ops.
- `C : CELLS` is the cells structure describing the register conventions of the architecture.
- `W : FIXED_POINT` is a structure that contains a fixed point type used in execution frequency annotations.

The type `weight` below is used in execution frequency annotations:

```
type weight = W.fixed_point
```

There are a few different kinds of basic blocks, described by the type `block_kind` below:

```
datatype block_kind =
    START
  | STOP
  | FUNCTION_ENTRY
  | NORMAL
  | HYPERBLOCK
```

A basic block is defined as the datatype `block`, defined below:

```
and data = LABEL  of Label.label
         | PSEUDO of P.pseudo_op

and block =
```

[237] **file:** `IR/mlrisc-cfg.sig`

```
BLOCK of
{  id          : int,
   kind        : block_kind,
   name        : B.name,
   freq        : weight ref,
   data        : data list ref,
   labels      : Label.label list ref,
   insns       : I.instruction list ref,
   annotations : Annotations.annotations ref
}
```

Edges in a CFG are annotated with the type `edge_info`, defined below:

```
and edge_kind = ENTRY
              | EXIT
              | JUMP
              | FALLSTHRU
              | BRANCH of bool
              | SWITCH of int
              | SIDEEXIT of int

and edge_info =
    EDGE of { k : edge_kind,
              w : weight ref,
              a : Annotations.annotations ref
            }
```

Type `cfg` below defines a control flow graph:

```
type edge = edge_info edge
type node = block node

datatype info =
    INFO of { regmap      : C.regmap,
              annotations : Annotations.annotations ref,
              firstBlock  : int ref,
              reorder     : bool ref
            }
type cfg = (block,edge_info,info) graph
```

33.1.2 Low-level Interface

The following subsection describes the low-level interface to a CFG. These functions should be used with care since they do not always maintain high-level structural invariants imposed on the representation. In general, higher level interfaces exist so knowledge of this interface is usually not necessary for customizing MLRISC.

Various kinds of annotations on basic blocks are defined below:

```
exception LIVEOUT of C.cellset
```

```
exception CHANGED of unit -> unit
exception CHANGEDONCE of unit -> unit
```

The annotation `LIVEOUT` is used record live-out information on an escaping block. The annotations `CHANGED` and `CHANGEDONCE` are used internally for maintaining views on a CFG. These should not be used directly.

The following are low-level functions for building new basic blocks. The functions `new`*XXX* build empty basic blocks of a specific type. The function `defineLabel` returns a label to a basic block; and if one does not exist then a new label will be generated automatically. The functions `emit` and `show_block` are low-level routines for displaying a basic block.

```
val newBlock          : int * B.name -> block
val newStart          : int -> block
val newStop           : int -> block
val newFunctionEntry  : int -> block
val copyBlock         : int * block -> block
val defineLabel       : block -> Label.label
val emit              : C.regmap -> block -> unit
val show_block        : C.regmap -> block -> string
```

Methods for building a CFG are listed as follows:

```
val cfg      : info -> cfg
val new      : C.regmap -> cfg
val subgraph : cfg -> cfg
val init     : cfg -> unit
val changed  : cfg -> unit
val removeEdge : cfg -> edge -> unit
```

Again, these methods should be used only with care.

The following functions allow the user to extract low-level information from a flowgraph. Function `regmap` returns the current register map. Function `regmap` returns a function that lookups the current register map. Function `liveOut` returns liveOut information from a block; it returns the empty cellset if the block is not an escaping block. Function `fallsThruFrom` takes a node id v and locates the block u (if any) that flows into v without going through a branch instruction. Similarly, the function `fallsThruTo` takes a node id u and locates the block (if any) that u flows into with going through a branch instruction. If u falls through to v in any feasible code layout u must preceed v.

```
val regmap    : cfg -> C.regmap
val reglookup : cfg -> C.register -> C.register
val liveOut   : block -> C.cellset
val fallsThruFrom : cfg * node_id -> node_id option
val fallsThruTo   : cfg * node_id -> node_id option
```

To support graph viewing of a CFG, the following low-level primitives are provided:

```
val viewStyle      : cfg -> (block,edge_info,info) GraphLayout.style
val viewLayout     : cfg -> GraphLayout.layout
val headerText     : block -> string
val footerText     : block -> string
val subgraphLayout :  cfg : cfg, subgraph : cfg  -> GraphLayout.layout
```

Finally, a miscellany function for control dependence graph building.

```
val cdgEdge : edge_info -> bool
```

33.1.3 IR

The MLRISC intermediate representation is a composite view of various compiler data structures, including the control flow graph, (post-)dominator trees, control dependence graph, and loop nesting tree. Basic compiler optimizations in MLRISC operate on this data structure; advance optimizations operate on more complex representations which use this representation as the base layer.

This IR provides a few additional functionalities:

- Edge frequencies – execution frequencies are maintained on all control flow edges.
- Extensible annotations – semantics information can be represented as annotations on the graph.
- Multiple facets – Facets are high-level views that automatically keep themselves up-to-date. Computed facets are cached and out-of-date facets are recomputed by demand. The IR defines a mechanism to attach multiple facets to the IR.

The signature of the IR is listed below

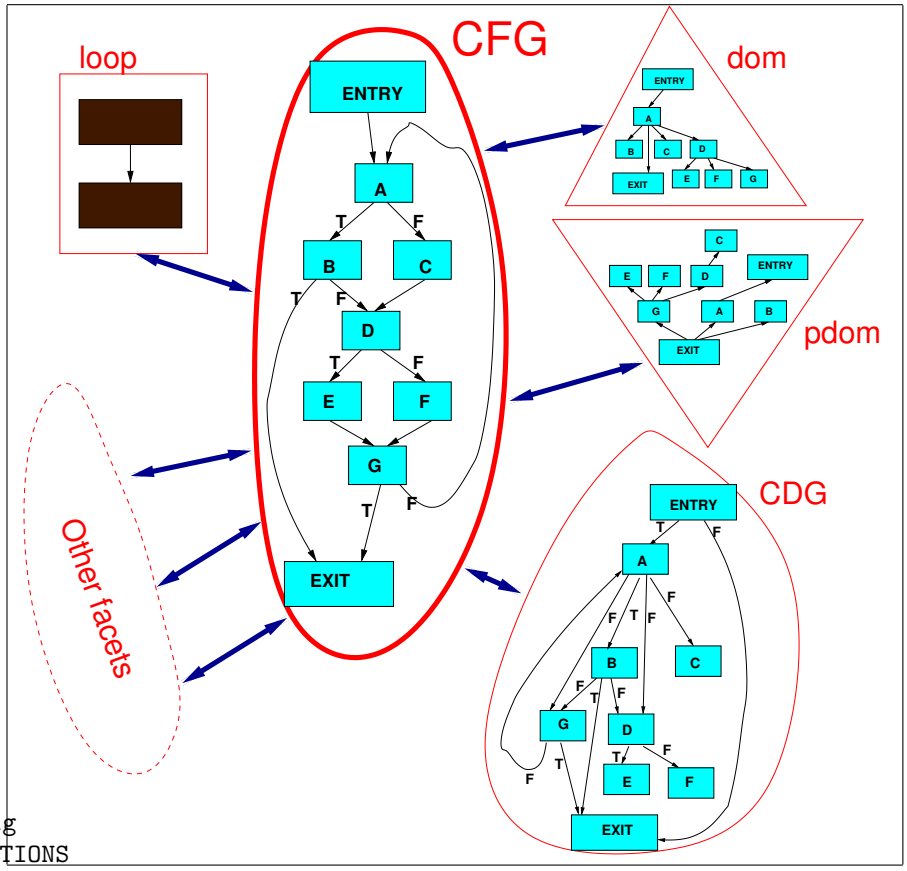


Figure 4: The MLRISC IR

```
signature MLRISC_IR[238] = sig
  structure I    : INSTRUCTIONS
  structure CFG  : CONTROL_FLOW_GRAPH
  structure Dom  : DOMINATOR_TREE
  structure CDG  : CONTROL_DEPENDENCE_GRAPH
  structure Loop : LOOP_STRUCTURE
  structure Util : CFG_UTIL
     sharing Util.CFG = CFG
     sharing CFG.I = I
     sharing Loop.Dom = CDG.Dom = Dom
```

[238] **file:** IR/mlrisc-ir.sig

```
  type cfg  = CFG.cfg
  type IR   = CFG.cfg
  type dom  = (CFG.block,CFG.edge_info,CFG.info) Dom.dominator_tree
  type pdom = (CFG.block,CFG.edge_info,CFG.info) Dom.postdominator_tree
  type cdg  = (CFG.block,CFG.edge_info,CFG.info) CDG.cdg
  type loop = (CFG.block,CFG.edge_info,CFG.info) Loop.loop_structure

  val dom   : IR -> dom
  val pdom  : IR -> pdom
  val cdg   : IR -> cdg
  val loop  : IR -> loop

  val changed : IR -> unit
  val memo : (IR -> 'facet) -> IR -> 'facet
  val addLayout : string -> (IR -> GraphLayout.layout) -> unit
  val view : string -> IR -> unit
  val views : string list -> IR -> unit
  val viewSubgraph : IR -> cfg -> unit
end
```

The following facets are predefined: dominator, post-dominator tree, control dependence graph and loop nesting structure. The functions `dom`, `pdom`, `cdg`, `loop` are *facet extraction* methods that compute up-to-date views of these facets.

The following protocol is used for facets:

- When the IR is changed, the function `changed` should be called to signal that all facets attached to the IR should be updated.
- To add a new facet of type `F` that is computed by demand, the programmer has to provide a facet construction function `f : IR -> F`. Call the function `mem` to register the new facet. For example, let `val g = memo f`. Then the function `g` can be used to as a new facet extraction function for facet `F`.
- To register a graph viewing function, call the function `addLayout` and provide an appropriate graph layout function. For example, we can say `addLayout "F" layoutF` to register a graph layout function for a facet called "F".

To view an IR, the functions `view`, `views` or `viewSubgraph` can be used. They have the following interpretation:

- `view` computes a layout for one facet of the IR and displays it. The predefined facets are called "dom", "pdom", "cdg", "loop." The IR can be viewed as the facet "cfg." In addition, there is a layout named "doms" which displays the dominator tree and the post-dominator tree together, with the post-dominator inverted.
- `views` computes a set of facets and displays it together in one single picture.

- `viewSubgraph` layouts a subgraph of the IR. This creates a picture with the subgraph highlighted and embedded in the whole IR.

33.1.4 Building a CFG

There are two basic methods of building a CFG:

- convert a sequence of machine instructions into a CFG through the emitter interface, described below, and
- convert it from a *cluster*, which is the basic linearized representation used in the MLRISC system.

The first method requires you to perform instruction selection from a compiler front-end, but allows you to bypass all other MLRISC phases if desired. The second method allows you to take advantage of various MLRISC's instruction selection modules currently available. We describe these methods in this section.

Directly from Instructions Signature `CODE_EMITTER` below describes an abstract emitter interface for accepting a linear stream of instructions from a source and perform a sequence of actions based on this stream[239].

```
signature CODE_EMITTER[240] = sig
  structure I : INSTRUCTIONS
  structure C : CELLS
  structure P : PSEUDO_OPS
     sharing I.C = C

  type emitter =
  {  defineLabel : Label.label -> unit,
     entryLabel  : Label.label -> unit,
     exitBlock   : C.cellset -> unit,
     pseudoOp    : P.pseudo_op -> unit,
     emitInstr   : I.instruction -> unit,
     comment     : string -> unit,
     init        : int -> unit,
     finish      : unit -> unit
  }
end
```

The code emitter interface has the following informal protocol.

[239] Unlike the signature `EMITTER_NEW` or `FLOWGRAPH_GEN`, it has the advantage that it is not tied into any form of specific flowgraph representation.

[240] **file:** `extensions/code-emitter.sig`

`init(`n`)`	Initializes the emitter and signals that the back-end should allocate space for n bytes of machine code.
`defineLabel(`l`)`	Defines a new label l at the current position.
`entryLabel(`l`)`	Defines a new entry label l at the current position. An entry label defines an entry point into the curren
`exitBlock(`c`)`	Defines an exit at the current position. The cellset c represents the live-out information
`pseudOp(`p`)`	Emits an pseudo op p at the current position
`emitInstr(`i`)`	Emits an instruction i at the current position
`blockName(`b`)`	Changes the block name to b
`comment(`msg`)`	Emits a comment msg at the current position
`finish`	Signals that the use of the emitter is finished. The emitter is free to perform its post-processing functio

The functor `ControlFlowGraphGen` below can be used to create a CFG builder that uses the `CODE_EMITTER` interface.

```
signature CONTROL_FLOW_GRAPH_GEN[241] = sig
  structure CFG     : CONTROL_FLOW_GRAPH
  structure Emitter : CODE_EMITTER
      sharing Emitter.I = CFG.I
      sharing Emitter.P = CFG.P
  val emitter : CFG.cfg -> Emitter.emitter
end
functor ControlFlowGraphGen[242]
   (structure CFG     : CONTROL_FLOW_GRAPH
    structure Emitter : CODE_EMITTER
    structure P       : INSN_PROPERTIES
        sharing CFG.I = Emitter.I = P.I
        sharing CFG.P = Emitter.P
        sharing CFG.B = Emitter.B
   ) : CONTROL_FLOW_GRAPH_GEN
```

Cluster to CFG The core MLRISC system implements many instruction selection front-ends. The result of an instruction selection module is a linear code layout block called a cluster. The functor `Cluster2CFG` below generates a translator that translates a cluster into a CFG:

```
signature CLUSTER2CFG[243] = sig
  structure CFG : CONTROL_FLOW_GRAPH
  structure F   : FLOWGRAPH
     sharing CFG.I = F.I
     sharing CFG.P = F.P
     sharing CFG.B = F.B
  val cluster2cfg : F.cluster -> CFG.cfg
end
functor Cluster2CFG[244]
  (structure CFG : CONTROL_FLOW_GRAPH
   structure F   : FLOWGRAPH
   structure P   : INSN_PROPERTIES
```

[241] **file:** `IR/mlrisc-cfg-gen.sig`
[242] **file:** `IR/mlrisc-cfg-gen.sml`
[243] **file:** `IR/mlrisc-cluster2cfg.sig`
[244] **file:** `IR/mlrisc-cluster2cfg.sml`

```
      sharing CFG.I = F.I = P.I
      sharing CFG.P = F.P
      sharing CFG.B = F.B
  ) : CLUSTER2CFG
```

CFG to Cluster The basic MLRISC system also implements many back-end functions such as register allocation, assembly output and machine code output. These modules all utilize the cluster representation. The functor `CFG2Cluster`[245] below generates a translator that converts a CFG into a cluster. With the previous functor, the CFG and the cluster presentation can be freely inter-converted.

```
signature CFG2CLUSTER[246] = sig
  structure CFG : CONTROL_FLOW_GRAPH
  structure F   : FLOWGRAPH
     sharing CFG.I = F.I
     sharing CFG.P = F.P
     sharing CFG.B = F.B
  val cfg2cluster :  cfg : CFG.cfg, relayout : bool  -> F.cluster
end
functor CFG2Cluster[247]
  (structure CFG  : CONTROL_FLOW_GRAPH
   structure F    : FLOWGRAPH
      sharing CFG.I = F.I
      sharing CFG.P = F.P
      sharing CFG.B = F.B
   val patchBranch : instr:CFG.I.instruction, backwards:bool ->
                        CFG.I.instruction list
  ) : CFG2CLUSTER
```

When a CFG originates from a cluster, we try to preserve the same code layout through out all optimizations when possible. The function `cfg2cluster` takes an optional flag that specifies we should force the recomputation of the code layout of a control flow graph when translating a CFG back into a cluster.

33.1.5 Basic CFG Transformations

Basic CFG transformations are implemented in the functor `CFGUtil`. These transformations include splitting edges, merging edges, removing unreachable code and tail duplication.

```
functor CFGUtil[248]
   (structure CFG : CONTROL_FLOW_GRAPH
    structure P   : INSN_PROPERTIES
       sharing P.I = CFG.I
   ) : CFG_UTIL
```

The signature of `CFGUtil` is defined below:

[245] **file:** `IR/mlrisc-cfg2cluster.sml`
[246] **file:** `IR/mlrisc-cfg2cluster.sig`
[247] **file:** `IR/mlrisc-cfg2cluster.sml`
[248] **file:** `IR/mlrisc-cfg-util.sml`

```
signature CFG_UTIL[249] = sig
   structure CFG : CONTROL_FLOW_GRAPH
   val updateJumpLabel : CFG.cfg -> node_id -> unit
   val mergeEdge       : CFG.cfg -> CFG.edge -> bool
   val eliminateJump   : CFG.cfg -> node_id -> bool
   val insertJump      : CFG.cfg -> node_id -> bool
   val splitEdge  : CFG.cfg ->  edge : CFG.edge, jump : bool
                    ->  edge : CFG.edge, node : CFG.node
   val isMerge        : CFG.cfg -> node_id -> bool
   val isSplit        : CFG.cfg -> node_id -> bool
   val hasSideExits   : CFG.cfg -> node_id -> bool
   val isCriticalEdge : CFG.cfg -> CFG.edge -> bool
   val splitAllCriticalEdges : CFG.cfg -> unit
   val ceed : CFG.cfg -> node_id * node_id -> bool
   val tailDuplicate : CFG.cfg -> { subgraph : CFG.cfg, root : node_id }
                               -> { nodes : CFG.node list,
                                    edges : CFG.edge list }
   val removeUnreachableCode : CFG.cfg -> unit
   val mergeAllEdges : CFG.cfg -> unit
end
```

These functions have the following meanings:

- `updateJumpLabel` Gu. This function updates the label of the branch instruction in a block u to be consistent with the control flow edges with source u. This is an nop if the CFG is already consistent.
- `mergeEdge` Ge. This function merges edge $e \equiv u \rightarrow v$ in the graph G if possible. This is successful only if there are no other edges flowing into v and no other edges flowing out from u. It returns true if the merge operation is successful. If successful, the nodes u and v will be coalesced into the block u. The jump instruction (if any) in the node u will also be elided.
- `eliminateJump` Gu. This function eliminate the jump instruction at the end of block u if it is feasible.
- `insertJump` Gu. This function inserts a jump instruction in block u if it is feasible.
- `splitEdge` Ge. This function split the control flow edge e, and return a new edge e' and the new block u as return values. It addition, it takes as argument a flag `jump`. If this flag is true, then a jump instruction is always placed in the split; otherwise, we try to eliminate the jump when feasible.
- `isMerge` Gu. This function tests whether block u is a *merge* node. A merge node is a node that has two or more incoming flow edges.
- `isSplit` Gu. This function tests whether block u is a *split* node. A split node is a node that has two or more outgoing flow edges.
- `hasSideExits` Gu. This function tests whether a block has side exits G. This assumes that u is a hyperblock.
- `isCriticalEdge` Ge. This function tests whether the edge e is a *critical* edge. The edge $e \equiv u \rightarrow v$ is critical iff there are u is merge node and v is a split node.

[249] **file:** `IR/mlrisc-cfg-util.sig`

- `splitAllCriticalEdges` G. This function goes through the CFG G and splits all critical edges in the CFG. This can introduce extra jumps and basic blocks in the program.
- `mustPreceed` $G(u,v)$. This function checks whether two blocks u and v are necessarily adjacent. Blocks u and v must be adjacent iff u must preceed v in any feasible code layout.
- `tailDuplicate`.

```
val tailDuplicate : CFG.cfg -> { subgraph : CFG.cfg, root : node_id }
                            -> { nodes : CFG.node list,
                                 edges : CFG.edge list }
```

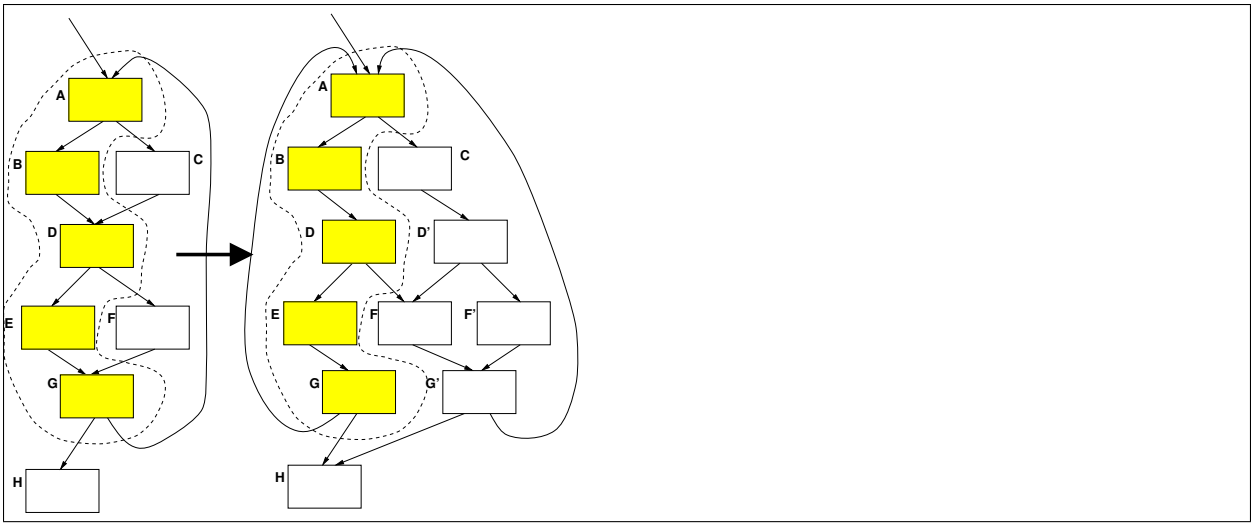


Figure 5: Tail-duplication

This function tail-duplicates the region `subgraph` until it only has a single entry `root`. Return the set of new nodes and new edges. The region is represented as a subgraph view of the CFG. Figure 5 illustrates this transformation.

- `removeUnreachableCode` G. This function removes all unreachable code from the graph.
- `mergeAllEdges` G. This function tries to merge all the edges in the flowgraph G. Merging is performed in the non-increasing order of edge frequencies.

33.1.6 Dataflow Analysis

MLRISC provides a simple customizable module for performing iterative dataflow analysis. A dataflow analyzer has the following signature:

```
signature DATAFLOW_ANALYZER[250] = sig
  structure CFG : CONTROL_FLOW_GRAPH
  type dataflow_info
  val analyze : CFG.cfg * dataflow_info -> dataflow_info
end
```

A dataflow problem is described by the signature `DATAFLOW_PROBLEM`, described below:

```
signature DATAFLOW_PROBLEM[251] = sig
  structure CFG : CONTROL_FLOW_GRAPH
  type domain
  type dataflow_info
  val forward   : bool
  val bot       : domain
  val ==        : domain * domain -> bool
  val join      : domain list -> domain
  val prologue  : CFG.cfg * dataflow_info ->
                      CFG.block node ->
                          { input    : domain,
                            output   : domain,
                            transfer : domain -> domain
                          }
  val epilogue  : CFG.cfg * dataflow_info ->
                      { node   : CFG.block node,
                        input  : domain,
                        output : domain
                      } -> unit
end
```

This description contains the following items

- `type domain` is the abstract lattice domain D.
- `type dataflow_info` is where the dataflow information is stored.
- `forward` is true iff the dataflow problem is in the forward direction
- `bot` is the bottom element of D.
- `==` is the equality function on D.
- `join` is the least-upper-bound function on D.
- `prologue` is a user-supplied function that performs pre-processing and setup. For each CFG node X, this function computes
  - `input` – which is the initial input value of X
  - `output` – which is the initial output value of X
  - `transfer` – which is the transfer function on X.

[250] **file:** `IR/dataflow.sig`
[251] **file:** `IR/dataflow.sig`

- `epilogue` is a function that performs post-processing. It visits each node X in the flowgraph and return the resulting `input` and `output` value for X.

To generate a new dataflow analyzer from a dataflow problem, the functor `Dataflow` can be used:

```
functor Dataflow[252](P : DATAFLOW_PROBLEM) : DATAFLOW_ANALYZER =
```

33.1.7 Static Branch Prediction

33.1.8 Branch Optimizations

[252] **file:** `IR/dataflow.sml`

34 SSA Optimizations

All SSA optimization modules satisfy the signature SSA_OPTIMIZATION[253], which is defined as:

```
signature SSA_OPTIMIZATION = sig
   structure SSA : SSA

   val optimize : SSA.ssa -> SSA.ssa
end
```

The following SSA based scalar optimizations have been implemented in MLRISC.

- Dead code elimination[254]
- Global value numbering, constant folding, algebraic simplication[255]
- Global code motion[256]
- Conditional constant propagation[257]
- Strength reduction[258]

[253] **file:** SSA/ssa-optimization.sig
[254] **file:** SSA/ssa-dead-code-elim.sml
[255] **file:** SSA/ssa-gvn.sml
[256] **file:** SSA/ssa-gcm.sml
[257] **file:** SSA/ssa-cond-const-prop.sml
[258] **file:** SSA/ssa-op-str-red.sml

35 ILP Optimizations

35.1 Introduction

This section is under construction. A new scheduler framework for superscalars that ties into the machine description language is currently being developed.

35.2 The ILP ToolBox

35.2.1 List Scheduler

35.2.2 Ranking Algorithms

Some more complex ranking algorithms (than say critical path) have been implemented. These are:

- The algorithm of `Palem and Simons`[259] which appeared in TOPLAS '93. This algorithm computes the modified deadlines of a set instructions, with precedence, latency, and deadlines constraints.
- The algorithm of `Leung, Palem, and Pnueli`[260] which appeared in PACT '98. This algorithm computes the modified deadlines of a set of instructions, with precedence, latency, release-times and deadline constraints.

[259] **file:** `scheduling/PalemSimons.sig`
[260] **file:** `scheduling/LeungPalemPnueli.sig`

36 Optimizations for VLIW/EPIC Architectures

36.1 Overview

Many newer architectures such as the upcoming IA-64 and the DSPs such as the C6 are VLIW or so called EPIC machines. These architectures depends on the compiler to extract instruction level parallelism (*ILP*) and data level parallelism (*DLP*).

Optimizations for these architectures include:

- Hyperblock construction
- Predication and predicate analysis
- Hyperblock scheduling
- Modulo scheduling

36.2 Hyperblocks

36.3 Predicate Analysis

36.4 Hyperblock Scheduling

36.5 Modulo Scheduling

37 Register Allocator

The MLRISC register allocator implements the iterated-coalescing algorithm described in POPL '96 [George, Appel]. The details are described in these papers

1. A New MLRISC Register Allocator[261]

[261] url: `http://cm.bell-labs.com/cm/cs/what/smlnj/compiler-notes/new-ra.ps`

Part IV
Back Ends

38 The Alpha Back End

38.1 Trap Shadows, Floating Exceptions, and Denormalized Numbers on the DEC Alpha

By Andrew W. Appel and Lal George, Nov 28, 1995

See section 4.7.5.1 of the *Alpha Architecture Reference Manual*.

The Alpha has imprecise exceptions, meaning that if a floating point instruction raises an IEEE exception, the exception may not interrupt the processor until several successive instructions have completed. ML, on the other hand, may want a "precise" model of floating point exceptions.

Furthermore, the Alpha hardware does not support denormalized numbers (for "gradual underflow"). Instead, underflow always rounds to zero. However, each floating operation (add, mult, etc.) has a trapping variant that will raise an exception (imprecisely, of course) on underflow; in that case, the instruction will produce a zero result AND an exception will occur. In fact, there are several variants of each instruction; three variants of MULT are:

MULT s1,s2,d truncate denormalized result to zero; no exception

MULT/U s1,s2,d truncate denormalized result to zero; raise UNDERFLOW

MULT/SU s1,s2,d software completion, producing denormalized result

The hardware treats the `MULT/U` and `MULT/SU` instructions identically, truncating a denormalized result to zero and raising the UNDERFLOW exception. But the operating system, on an UNDERFLOW exception, examines the faulting instruction to see if it's an `/SU` form, and if so, recalculates `s1*s2`, puts the right answer in `d`, and continues, all without invoking the user's signal handler.

Because most machines compute with denormalized numbers in hardware, to maximize portability of SML programs, we use the `MULT/SU` form. (and `ADD/SU`, `SUB/SU`, etc.) But to use this form successfully, certain rules have to be followed. Basically, d cannot be the same register as s1 or s2, because the opsys needs to be able to recalculate the operation using the original contents of s1 and s2, and the MULT/SU instruction will overwrite d even if it traps.

More generally, we may want to have a sequence of floating-point instructions. The rules for such a sequence are:

1. The sequence should end with a `TRAPB` (trap barrier) instruction. (This could be relaxed somewhat, but certainly a `TRAPB` would be a good idea sometime before the next branch instruction or update of an ML reference variable, or any other ML side effect.) 2. No instruction in the sequence should destroy any operand of itself or of any previous instruction in the sequence. 3. No two instructions in the sequence should write the same destination register.

We can achieve these conditions by the following trick in the Alpha code generator. Each instruction in the sequence will write to a different temporary; this is guaranteed by the translation from ML-RISC. At the beginning of the sequence, we will put a special pseudo-instruction (we call it `DEFFREG`) that "defines" the destination register of the arithmetic instruction. If there are K arithmetic instructions in the sequence, then we'll insert K `DEFFREG` instructions all at the beginning of the sequence. Then, each arithop will not only "define" its destination temporary but will "use" it as well. When all these instructions are fed to the liveness analyzer, the resulting interference graph will then have inteference edges satisfying conditions 2 and 3 above.

Of course, `DEFFREG` doesn't actually generate any code. In our model of the Alpha, every instruction generates exactly 4 bytes of code except the "span-dependent" ones. Therefore, we'll specify `DEFFREG` as a span-dependent instruction whose minimum and maximum sizes are zero.

At the moment, we do not group arithmetic operations into sequences; that is, each arithop will be preceded by a single `DEFFREG` and followed by a `TRAPB`. To avoid the cost of all those `TRAPB`'s, we should improve this when we have time. Warning: Don't put more than 31 instructions in the sequence, because they're all required to write to different destination registers!

What about multiple traps? For example, suppose a sequence of instructions produces an Overflow and a Divide-by-Zero exception? ML would like to know only about the earliest trap, but the hardware will report *BOTH* traps to the operating system. However, as long as the rules above are followed (and the software-completion versions of the arithmetic instructions are used), the operating system will have enough information to know which instruction produced the trap. It is very probable that the operating system will report *ONLY* the earlier trap to the user process, but I'm not sure.

For a hint about what the operating system is doing in its own trap-handler (with software completion), see section 6.3.2 of "*OpenVMS Alpha Software*" (Part II of the Alpha Architecture Manual). This stuff should apply to Unix (OSF1) as well as VMS.

39 The PA RISC Back End

No documentation yet.

40 The Sparc Back End

The Sparc back end can function in two different modes:

Sparc V8 This is V8 instruction set is used. In this mode the processor behaves like a 32-bit processor. In this mode we assume we have 16 floating point registers numbered `%f0, %f2, %f4, ..., %f30`. These are all in IEEE double precision.

Sparc V9 This generates code assuming the V9 instruction set is used. In this mode the processor functions at 64-bit. In this mode the floating point processors can number from `%f0, %f2, %f4, ..., %f62`. These are all in IEEE double precision.

New V9 instructions include the 64-bit extended version of multiplications, divisions, shifts, and load and store.

```
MULX SMULX DIVX SLLX SRLX SRAX LDX STX
```

Also, V9 includes conditional moves and more general form of branches.

MOVcc conditional moves on condition code

FMOVcc conditional moves on condition code

MOVR conditional moves on integer condition

BR branch on integer register with prediction

BP branch on integer condition with prediction

40.1 General Setup for V8

The SPARC architecture has 32 general purpose registers (`%g0` is always 0) and 32 single precision floating point registers.

Some Ugliness: double precision floating point registers are register pairs. There are no double precision moves, negation and absolute values. These require two single precision operations. I've created composite instructions `FMOVd`, `FNEGd` and `FABSd` to stand for these.

All integer arithmetic instructions can optionally set the condition code register. We use this to simplify certain comparisons with zero in the instruction selection process.

Integer multiplication, division and conversion from integer to floating go thru the pseudo instruction interface, since older sparcs do not implement these instructions in hardware.

In addition, the trap instruction for detecting overflow is a parameter. This allows different trap vectors to be used.

40.2 General Setup for V9

40.3 Specializing the Sparc Back End

41 The Intel x86 Back End

No documentation yet.

42 The PowerPC Back End

No documentation yet.

43 The MIPS Back End

No documentation yet.

44 The TI C6x Back End

No documentation yet.

Part V
Basic Types

45 Annotations

45.1 Overview

A compiler front-end has to be propagate information to the back-end. An optimization phase may have to leave behind information at various places of the IR so that other phases can reuse such information. MLRISC uses the *annotations* mechanism for these functions. Individual instructions, basic blocks, and flow graph edges, can be attached one or more annotations.

The basic MLRISC system understands many annotations. Some examples are:

COMMENT these can be used to attach comments. If attached to an instruction, the assemblers will output them as part of their assembly output.

BRANCH_PROB these can be attached to a branch instruction to indicate the probability in which is it taken.

EXECUTION_FREQ these can be attached to a basic block to indicate its expected execution frequency

45.2 Details

The primitive annotations datatype is defined to have this `signature`[262]. In addition, MLRISC predefined a few primitive annotations that are recognized by the core system. This signature is `MLRISC_ANNOTATIONS`[263]. More detailed documentation can be found in this paper[264].

[262] **file:** `library/annotations.sig`
[263] **file:** `instructions/mlriscAnnotations.sig`
[264] url: `http://cm.bell-labs.com/cm/cs/what/smlnj/compiler-notes/annotations.ps`

46 Cells

MLRISC uses the CELLS[265] interface to define all readable/writable resources in a machine architecture, or *cells* The types defined herein are:

- cellkind – different classes of cells are assigned difference cellkinds. The following cellkinds should be present
  - GP – general purpose registers.
  - FP – floating point registers.
  - CC – condition code registers.

  In addition, the cellkinds MEM and CTRL should also be defined. These are used for representing memory based data dependence and control dependence.
  - MEM – memory
  - CTRL – control dependence
- regmap – register map[266]
- cellset – a cellset represent a set of cells. This type can be used to denote live-in/live-out information. Cellsets are implemented as immutable abstract types.

These core definitions are defined in the following signature

```
signature CELLS_BASIS[267] =
sig
   eqtype cellkind
   type cell = int
   type regmap = cell Intmap.intmap
   exception Cells

   val cellkinds : cellkind list
   val cellkindToString : cellkind -> string
   val firstPseudo : cell
   val Reg   : cellkind -> int -> cell
   val GPReg : int -> cell
   val FPReg : int -> cell
   val cellRange : cellkind -> low:int, high:int
   val newCell   : cellkind -> 'a -> cell
   val cellKind : cell -> cellkind
   val updateCellKind : cell * cellkind -> unit
   val numCell   : cellkind -> unit -> int
   val maxCell   : unit -> cell
   val newReg    : 'a -> cell
   val newFreg   : 'a -> cell
   val newVar    : cell -> cell
```

[265] **file:** instructions/cells.sig
[266] url: regmap.html
[267] **file:** instructions/cells.sig

```
   val regmap    : unit -> regmap
   val lookup    : regmap -> cell -> cell
   val reset     : unit -> unit
end
```

- `cellkinds` – this is a list of all the cellkinds defined in the architecture
- `cellkindToString` – this function maps a cellkind into its name
- `firstPseudo` – MLRISC numbered physical resources in the architecture from 0 to firstPseudo-1. This is the first usable virtual register number.
- `Reg` – This function maps the ith physical resource of a particular cellkind to its internal encoding used by MLRISC. Note that all resources in MLRISC are named uniquely.
- `GPReg` – abbreviation for `Reg GP`
- `FPReg` – abbreviation for `Reg FP`
- `cellRange` – this returns a range `low, high` when given a cellkind, with denotes the range of physical resources
- `newCell` – This function returns a new virtual register of a particular cellkind.
- `newReg` – abbreviation as `newCell GP`
- `newFreg` – abbreviation as `newCell FP`
- `cellKind` – When given a cell number, this returns its cellkind. Note that this feature is not enabled by default.
- `updateCellKind` – updates the cellkind of a cell.
- `numCell` – returns the number of virtual cells allocated for one cellkind.
- `maxCell` – returns the next virtual cell id.
- `newVar` – given a cell id, return a new cell id of the same cellkind.
- `regmap` – This function returns a new empty regmap
- `lookup` – This converts a regmap into a lookup function.
- `reset` – This function resets all counters associated with all virtual cells.

```
signature CELLS = sig
   include CELLS_BASIS
   val GP   : cellkind
   val FP   : cellkind
   val CC   : cellkind
   val MEM  : cellkind
   val CTRL : cellkind
   val toString : cellkind -> cell -> string
   val stackptrR : cell
```

```
   val asmTmpR : cell
   val fasmTmp : cell
   val zeroReg : cellkind -> cell option

   type cellset

   val empty      : cellset
   val addCell    : cellkind -> cell * cellset -> cellset
   val rmvCell    : cellkind -> cell * cellset -> cellset
   val addReg     : cell * cellset -> cellset
   val rmvReg     : cell * cellset -> cellset
   val addFreg    : cell * cellset -> cellset
   val rmvFreg    : cell * cellset -> cellset
   val getCell    : cellkind -> cellset -> cell list
   val updateCell : cellkind -> cellset * cell list -> cellset

   val cellsetToString : cellset -> string
   val cellsetToString' : (cell -> cell) -> cellset -> string

   val cellsetToCells : cellset -> cell list
end
```

- `toString` – convert a cell id of a certain cellkind into its assembly name.
- `stackptrR` – the cell id of the stack pointer register.
- `asmTmpR` – the cell id of the assembly temporary
- `fasmTmp` – the cell id of the floating point temporary
- `zeroReg` – given the cellkind, returns the cell id of the source that always hold the value of zero, if there is any.
- `empty` – an empty cellset
- `addCell` – inserts a cell into a cellset
- `rmvCell` – remove a cell from a cellset
- `addReg` – abbreviation for `addCell GP`
- `rmvReg` – abbreviation for `rmvCell GP`
- `addFreg` – abbreviation for `addCell FP`
- `rmvFreg` – abbreviation for `rmvCell FP`
- `getCell` – lookup all cells of a particular cellkind from the cellset
- `updateCell` – replace all cells of a particular cellkind from the cellset.
- `cellsetToString` – pretty print a cellset
- `cellsetToString'` – pretty print a cellset, but first apply a regmap function.
- `cellsetToCells` – convert a cellset into list form.

47 Cluster

A *cluster* represents a compilation unit in linearized form, and contains information about the control flow, global annotations, block and edge execution frequencies, and live-in/live-out information.

Its signature is:

```
signature FLOWGRAPH = sig
  structure C : CELLS[268]
  structure I : INSTRUCTIONS[269]
  structure P : PSEUDO_OPS[270]
  structure W : FREQ[271]
     sharing I.C = C

  datatype block =
      PSEUDO of P.pseudo_op
    | LABEL of Label.label
    | BBLOCK of
        { blknum      : int,
          freq        : W.freq ref,
          annotations : Annotations.annotations ref,
  liveIn      : C.cellset ref,
  liveOut     : C.cellset ref,
  succ        : edge list ref,
  pred        : edge list ref,
  insns       : I.instruction list ref
        }
    | ENTRY of
        {blknum : int, freq : W.freq ref, succ : edge list ref}
    | EXIT of
        {blknum : int, freq : W.freq ref, pred : edge list ref}
  withtype edge = block * W.freq ref

  datatype cluster =
      CLUSTER of {
        blocks: block list,
        entry : block,
        exit  : block,
        regmap: C.regmap,
        blkCounter : int ref,
        annotations : Annotations.annotations ref
      }
end
```

[268] url: cells.html
[269] url: instructions.html
[270] url: pseudo-ops.html
[271] url: freq.html

Clusters are used in span dependency resolution[272], delay slot filling[273], assembly[274], and machine code[275] output, since these phases require the code laid out in linearized form.

[272]url: `span-dep.html`
[273]url: `delayslots.html`
[274]url: `asm.html`
[275]url: `mc.html`

48 Client Defined Constants

48.0.1 Introduction

MLRISC allows the client to inject abstract *constants* that are resolved only at the end of the compilation phase into the instruction stream. These constants can be used whereever an integer literal is expected. Typical usage are stack frame offsets for spill locations which are only known after register allocation, and garbage collection and exception map which are resolved only when all address calculation are performed.

48.0.2 The Details

Client defined constants should satsify the following signature:

```
signature CONSTANT[276] = sig
   type const

   val toString : const -> string
   val valueOf  : const -> int
   val hash     : const -> word
   val ==       : const * const -> bool
end
```

The methods are:

`toString`	a pretty printing function
`valueOf`	returns the value of the constant
`hash`	returns the hash value of the constant
`==`	compare two constants for identity

The method `toString` should be implemented in all cases. The method `valueOf` is necessary only if machine code generation is used. The last two methods, `hash` and `==` are necessary only if SSA optimizations are used.

[276] **file:** `instructions/constant.sig`

49 Client Defined Pseudo Ops

49.1 Introduction

Pseudo ops are client defined instruction stream markers. They can be used to represent assembly directives. Pseudo ops should satisfy the following signature:

```
signature PSEUDO_OPS[277] = sig
  type pseudo_op
  val toString : pseudo_op -> string
  val emitValue : pOp:pseudo_op, loc:int, emit:Word8.word -> unit -> unit
  val sizeOf : pseudo_op * int -> int
  val adjustLabels : pseudo_op * int -> bool
end
```

The method that is required is:

- `toString` – pretty printing the pseudo in assembly format.

When machine code generation is used, we also have to implement the following methods:

- `emitValue` – emit value of pseudo op give current location counter and output stream. The value emitted should respect the endianness of the target machine.
- `sizeOf` – Size of the pseudo op in bytes given the current location counter The location counter is provided in case some pseudo ops are dependent on alignment considerations.
- `adjustLabels` – adjust the value of labels in the pseudo op given the current location counter.

These methods are involved during the span dependence resolution[278] phase to determine the size and layout of the pseudo ops.

[277] **file:** `instructions/pseudoOps.sig`
[278] url: `span-dep.html`

50 Instructions

Instructions in MLRISC are implemented as abstract datatypes and must satisfy the signature INSTRUCTIONS[279], defined as follows:

```
signature INSTRUCTIONS =
sig
   structure C        : CELLS[280]
   structure Constant : CONSTANT[281]
   structure LabelExp : LABELEXP[282]
      sharing LabelExp.Constant = Constant

   type operand
   type ea
   type addressing_mode
   type instruction
end
```

Type operand is used to represent ioperands, ea is used to represent effective addresses, type addressing_mode is used to represent the internal addressing mode used by the architecture. Note that these are all abstract according to the signature, so the client has complete freedom in choosing the most convenient representation for these things.

50.1 Predication

For architectures that have full *predication* built-in, such as the C6xx or IA-64, the instruction set should be extended to satisfy the signature:

```
signature PREDICATED_INSTRUCTIONS[283] =
sig
   include INSTRUCTIONS

   type predicate
end
```

This basically says that the type that is used to represent a predicate can be implemented however the client wants. This flexibility is quite important since the predication model may differ substantially from architecture to architecture.

For example, in the TI C6, there are no seperate predicate register files and integer registers double as predicate registers, and the predicate true is any non-zero value. Each instruction can be predicated under a predicate register or its negation. In contrasts, architectures such as IA-64 and HP's Playdoh incorporate separate predicate registers into their architectures. In Playdoh, *predicate defining* instructions actually set a pair of complementary predicate registers, and instructions can only be predicated under the value of a predicate register, not its negation.

[279] **file:** instructions/instructions.sig
[280] url: cells.html
[281] url: constants.html
[282] url: labelexp.html
[283] **file:** instructions/pred-instructions.sig

50.2 VLIW

VLIW architectures differ from superscalars in that resource assignments are statically determined at compile time. We distinguish between two different types of resources, namely *functional units* and *data paths*. The latter type is particularly important for clustered architectures. The following signature is used to describe VLIW instructions:

```
signature VLIW_INSTRUCTIONS[284] =
sig

   include INSTRUCTIONS
   structure FU : FUNITS[285]
   structure DP : DATAPATHS[286]
end
```

The signature `FUNITS` is used to describe functional unit resources, while the signature `DATAPATHS` is used to describe data paths.

50.3 Predicated VLIW

Finally, instructions sets for predicated VLIW/EPIC machines should match the signature

```
signature PREDICATED_VLIW_INSTRUCTIONS[287] =
sig
   include VLIW_INSTRUCTIONS
   type predicate
end
```

[284] **file:** `instructions/vliw-instructions.sig`
[285] **file:** `instructions/funits.sig`
[286] **file:** `instructions/datapaths.sig`
[287] **file:** `instructions/pred-vliw-instructions.sig`

51 Instruction Streams

51.0.1 Overview

An *instruction stream* is an abstraction used by MLRISC to describe linearized instructions. This abstraction turns out to fit the function of many MLRISC modules. For example, a phase such as Instruction Selection[288] can be viewed as taking an stream of MLTREE[289] statements and return a stream of instructions[290]. Similarly, phases such as assembly output[291] and machine code generation[292] can be seen as taking a stream of instructions and returning a stream of characters and a stream of bytes.

51.0.2 The Details

An instruction stream satisfy the following abstract signature:

```
signature INSTRUCTION_STREAM[293] =
sig
   structure P : PSEUDO_OPS[294]

   datatype ('a,'b,'c,'d,'e,'f) stream =
      STREAM of
      { beginCluster: int -> 'b,
        endCluster  : 'c -> unit,
        emit        : 'a,
        pseudoOp    : P.pseudo_op -> unit,
        defineLabel : Label.label -> unit,
        entryLabel  : Label.label -> unit,
        comment     : string -> unit,
        annotation  : Annotations.annotation -> unit,
        exitBlock   : 'd -> unit,
        alias       : 'e -> unit,
        phi         : 'f -> unit
      }
end
```

This type is specialized in other modules as such the assembler[295], the machine code emitter[296], and the instruction selection modules[297].

51.0.3 The protocol

All instruction streams, irrespective of their actual types, follow the following protocol:

[288] url: instrsel.html
[289] url: mltree.html
[290] url: instructions.html
[291] url: asm.html
[292] url: mc.html
[293] **file:** instructions/stream.sig
[294] url: pseudo-ops.html
[295] url: asm.html
[296] url: mc.html
[297] url: instrsel.html

- The method `beginCluster` should be called at the beginning of the stream to mark the start of a new compilation unit. The integer passed to this method is the number of bytes in the stream. This integer is only used for machine code emitter, which uses it to allocate space for the code string.
- The method `endCluster` should be called when the entire compilation unit has been sent.
- In between these calls, the following methods can be called in any order:
  - `emit` – this method emits an instruction. It takes a regmap[298] as argument.
  - `pseudoOp` – this method emits a pseudo op.
  - `defineLabel` – this method defines a *local* label, i.e. a label that is only referenced within the same compilation unit.
  - `entryLabel` – this method defines an *enternal* label that marks an procedure entry, and may be referenced from other compilation units.
  - `comment` – this emits a comment string
  - `annotation` – this function attaches an annotation to the current basic block.
  - `exitBlock` – this marks the current block as an procedure exit.

[298] url: `regmap.html`

52 Label Expressions

A *label expression* is a constant expression defined in terms of labels, or user defined constants[299]. ML-RISC uses the type `labexp` to represent label expressions. Label expressions are defined in the structure `LabelExp`[300].

The datatype `labexp` has the following definition:

```
datatype labexp =
    LABEL of Label.label
  | CONST of Constant.const
  | INT of int
  | PLUS of labexp * labexp
  | MINUS of labexp * labexp
  | MULT of labexp * labexp
  | DIV of labexp * labexp
  | LSHIFT of labexp * word
  | RSHIFT of labexp * word
  | AND of labexp * word
  | OR of labexp * word
```

In addition, the following functions are defined in `labexp`:

- `valueOf : labexp -> int` – Returns the value associated with a label expression
- `toString : labexp -> string` – Return the pretty printed representation of an expression
- `hash : labexp -> word` – Returns the hash value of an expression
- `== : labexp * labexp -> bool` – Tests whether two label expression are lexically identical

The type `labexp` is depends on client defined constants[301] typed. The functor `LabelExp` is parameterized as follows.

`functor LabelExp`[302]`(Constant : CONSTANT`[303]`)`

[299] url: `constants.html`
[300] **file:** `instructions/labelExp.sml`
[301] url: `constants.html`
[302] **file:** `instructions/labelExp.sml`
[303] **file:** `instructions/constant.sig`

53 Labels

Labels are used as symbolic names for address. The structure `Label`[304] defines the label datatype. The following operations are defined on labels:

- `newLabel : string -> label` – Generate a new label with a given name. If the name is `""`, a new name is generated.
- `nameOf : label -> string` – Returns the name of a label
- `id : label -> int` – Return the unique id of a label
- `reset : unit -> unit` – Return the label id counter to 0.

For machine code generation, the following two additional methods are defined.

- `addrOf : label -> int` – Return the address associated with a label
- `setAddr : label * int -> unit` – Set the address associated with a label

See also Label Expressions[305].

[304] **file:** `instructions/labels.sml`
[305] url: `labelexp.html`

54 Regions

54.0.1 Overview

The MLRISC system uses user defined type called *regions* to propagate aliasing information to the back-end. This type is abstract and no constraint is imposed on how it is implemented. The advantage of this is that the client can optimize the representation of the region information according to the semantics of the source language. The downside of this freedom is that the client has to implement various modules to extract information from the regions datatype required by various optimization phases.

For clients that do not want to implement their own regions datatype, there is now a new generic mechanism, called *MLRiscRegions*, built on top of the regions concept, for propagating both:

- Aliasing information, and
- Control dependence/anti-control dependence information

Both kinds of information are crucial for extracting parallelism from the target code, and are used in all optimizations that perform code motion, such as SSA optimizations and all scheduling optimizations.

54.0.2 MLRisc Regions

55 Regmap

A *regmap* is a mapping from virtual register to virtual or physical register, and is used by MLRISC register allocators to represent the current binding of virtual registers. Regmaps are implemented as `Intmap`[306] in MLRISC, and are defined in the CELLS[307] interface.

Regmaps are used in phases such as assembly generation[308] and machine code[309]. MLRISC program representations such clusters[310] and IR[311] each contains a global regmap per compilation unit. Representations such as hyperblock[312] may contain its own regmap, which overrides the global regmap.

[306] **file:** `library/intmap.sml`
[307] url: `cells.html`
[308] url: `asm.html`
[309] url: `mc.html`
[310] url: `cluster.html`
[311] url: `mlrisc-ir.html`
[312] url: `hyperblock.html`

References

[BCHS88] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practial improvements to the construction and destructon of static single assignment form. *Software–Practice and Experience*, 1(1):1–28, January 1988.

[BR91] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method for computing static single assignment form. In *ACM SIGPLAN POPL*, pages 25–35, 1989.

[Ell85] John Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1985.

[Fis81] J. Fisher. Trace scheduling: A general technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, 1981.

[SG95] V. C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. *ACM SIGPLAN POPL*, pages 62–73, 1995.