Kupo Language Specification

Version 1.3.1
Justin David Smith
August, 2002

Abstract

This is the main language specification for Kupo, a term-rewriting language designed to simplify backend implementation for the Mojave Compiler Collection (MCC). Kupo supports architecture abstraction and instruction-stream rewriting, and is designed to significantly simplify peephole optimization and dead instruction elimination. This paper documents the Kupo grammar and gives the semantics of the various statements supported by Kupo. This paper also provides examples and implementation notes to provide additional insight on the design of Kupo.

Contents

List of Figures

1 Introduction

Note: The current version of this document is a specification, and as such it may not be the best document to read for an overview of the system. I recommend reading the x86 architecture files in conjunction with this file, otherwise some things may be unclear. `arch/x86/type/x86_arch.kupo` contains the architecture definition for x86; other `arch/x86/opt/x86_peephole_*.kupo` files define properties and patterns for x86 peephole optimizations. The `arch/x86/util/x86_mir.kupo` and `arch/x86/util/x86_mir_atom.kupo` files are used to define the MIR-to-assembly transformation.

1.1 What is Kupo?

Kupo is a term-rewriting language designed to make it easier to rewrite blocks of assembly code in an instruction stream. The language is designed specifically to simplify peephole optimization, however it can be used for other types of instruction-stream rewriting, including dead instruction elimination. It can also be used to generate new instruction streams, for example during MIR-to-assembly transformation. Kupo features live-register analysis, sophisticated pattern match expressions, wildcard matching of instructions, and transformation functions that operate on individual instructions. It is designed to be fairly architecture-independent, and includes a mechanism for defining instructions for a specific architecture.

Kupo files have a `.kupo` extension, and are compiled with the `moogleize` compiler. `moogleize` produces ML files that can be compiled by OCaml 3.04 as either native code or bytecode.

1.2 Quickstart: Using `moogleize`

To invoke the `moogleize` compiler, simply run `arch/x86/moogle/moogleize`. Only the basic options are documented here; for information on all available options, run `moogleize` with the `-help` option. You can specify the name of the emitted files with `-o` and `-oi`:

- `-o <filename>` sets the name of the emitted ML file. If this option is not specified, the ML file will be emitted to `stdout`.
- `-oi <filename>` sets the name of the emitted MLI interface file. If this option is not specified, no interface file will be emitted. In inline Kupo code, you will usually want to write your own interface file; for architecture definition and rewrite rules, you should let `moogleize` generate the interface file.

You must also specify what tables should be output; the output options are:

- `-insts` builds the instruction tables; these are usually generated off the main architecture file (e.g. `arch/x86/type/x86_arch.kupo`). The instruction tables include all tables used to classify instructions, and the transformation functions that operate on instructions. The following Kupo statements will emit code with this option:

  instructions (section 4.2.2)
  trivial (section 4.2.2)
  transform (section 4.2.3)

- `-rules` builds the pattern-match rules; these are generated off of the other Kupo files. The pattern-match rules contain the tables for actually rewriting an assembly instruction stream, used by peephole optimization and dead instruction elimination (among other things). The following Kupo statements will emit code with this option:

  rewrite (sections 4.3, 4.4)

- `-inline` emits all inlined Kupo code. Inline Kupo code is OCaml code with inlined Kupo expressions which are used to construct assembly instructions. This feature is particularly useful for the MIR-to-assembly conversion. The following Kupo statements will emit code with this option:

  inline_kupo (section 4.5)

You must specify at least one of the above three options. You usually want to emit the instruction tables only once, so it is recommended that you compile the architecture definition with `-insts` once, and build all other files without the `-insts` option.

An example use of `moogleize` for the x86 backend is given in figure 1.1. In the example, the architecture definition file is built with the `-insts` flags, and all other files are built with the `-rules` and `-inline` flags. The architecture definition file must be given as an input on every `moogleize` command, and is always specified as the first `.kupo` file[1]. All ML files generated by `moogleize` must be linked against the `Moogle_lib` and `Moogle_offset` modules. If you build the ML files using `cons`, this linking will be done automatically; otherwise, you should give the include flag for the directory containing `Moogle_lib` and `Moogle_offset` on the `ocamlc` command line.

```
# Build the architecture definition
cd arch/x86/type
moogleize -insts -o x86_arch.ml -oi x86_arch.mli x86_arch.kupo
ocamlopt.opt -w Au [includes...] -c -o x86_arch.cmi x86_arch.mli
ocamlopt.opt -w Au [includes...] -c -o x86_arch.o   x86_arch.ml

# Build some files containing rewrite rules
cd ../opt
moogleize -rules -o x86_peephole_basic.ml -oi x86_peephole_basic.mli \
   ../type/x86_arch.kupo x86_peephole_basic.kupo
moogleize -rules -o x86_peephole_memop.ml -oi x86_peephole_memop.mli \
   ../type/x86_arch.kupo x86_peephole_memop.kupo
ocamlopt.opt -w Au [includes...] -c -o x86_peephole_basic.cmi x86_peephole_basic.mli
ocamlopt.opt -w Au [includes...] -c -o x86_peephole_basic.o   x86_peephole_basic.ml
[...]

# Build some files containing inline Kupo code
# Note we do not need the -w Au flag to OCaml here...
# We do not have moogleize build a MLI usually...
cd ../util
moogleize -inline -o x86_mir.ml      ../type/x86_arch.kupo x86_mir.kupo
moogleize -inline -o x86_mir_atom.ml ../type/x86_arch.kupo x86_mir_atom.kupo
ocamlopt.opt [includes...] -c -o x86_mir.o x86_mir.ml
[...]
```

Figure 1.1: Example use of `moogleize`

2 Basic Concepts

The basic structure of the system allows users to specify architecture for the assembly language, as well as properties, transform functions, and rewrite rules. Properties are attached to specific instructions, and may even be attached to instructions restricted to taking certain operand types.

[1] Although the instruction tables will not be emitted without the `-insts` flag, the architecture definition file must be specified so its instruction and variable definitions will be in scope for other Kupo files.

Properties are primarily used in rewrite rules to classify instructions in the pattern match, and may also be called by external ML code to classify instructions. Transforms are used to rewrite single instructions, and rewrite rules are used to rewrite blocks of instructions from one form to another.

2.1 Architecture definition

The architecture definition is used to import the ML type definition for the assembly instructions, and define the instructions in Kupo using the instructions statement (section 4.2.2). Instruction definitions indicate the arity of the instruction, and may also restrict the types of the operands for instructions. The architecture definition is also used to define hardware and context registers on the platform using the declarations statement (section 4.2.1), and mark which instructions are "trivial" `NOP` instructions using the trivial statement (section 4.2.2). Finally, the architecture definition declares the type for the instruction set using the instruction_type statement (section 4.2.2).

2.2 Properties — classifying instructions

Properties (sometimes called tags) are used to classify the instructions. Properties may be attached to an opcode, but may also be associated with instructions which take specific operand types[2]. Properties may either be true or false for a particular instruction. Properties may be applied to any instruction in an instructions statements.

2.3 Transforms — rewriting single instructions

Transform functions are also defined in the architecture definition, using the transform statement (section 4.2.3). Transforms are functions that operate on a single instruction, and rewrite the instruction to another instruction. These are primarily used by rewrite rules that need to make slight alterations to a particular instruction[3]. They are implicit classification functions; the transform is usually only defined for a few instructions, therefore a rewrite rule using the transform can only be applied if the instruction matches a case in the transform.

2.4 Rewrite rules — rewriting instruction blocks

Rewrite rules are a primary feature of Kupo. Rewrite rules take a pattern block that matches a block of assembly instructions, and a replacement block which transforms the matched instructions into new instructions. Rewrite rules are specified with the rewrite statement; individual pattern and replacement blocks are specified with the input_block and output_block statements. The pattern/replacement scheme is very well suited for transformations of assembly code, and for complicated patterns it is much easier to use than ML's match statement[4].

2.5 Inline Kupo — generating instruction streams in ML code

Kupo code can be embedded in OCaml text to simplify generation of new instruction streams. inline Kupo code is used to construct instruction streams inside normal OCaml code, including the instruction streams for the MIR-to-assembly transformation. Kupo is more concise and expressive

[2] For example, a property may be applied to an instruction that specifically takes an integer register as its operand.

[3] For example, x86 floating-point optimizations use transforms frequently because of their tendency to reorder the stack, which requires revisions to the floating-point opcode; revisions to the opcode are not trivial to express with rewrite rules alone.

[4] Compare, for example, the emitted ML files sometime to the rules in the `.kupo` source file. It is easy to see that when conditions are required on the pattern beyond what structural patterns allow, the ML code becomes very complicated.

than OCaml[5] for instruction stream generation, and is often simpler to maintain than the equivalent OCaml code for building instructions. Inline Kupo code is defined by the inline_kupo statement.

2.6 Organization in `.kupo` files

The architecture, properties, transforms, patterns and inline code are all specified in `.kupo` files. The architecture definition (including properties and transforms) is usually specified in a separate `.kupo` file[6]. `moogleize` uses the architecture definition to build boolean ML functions that classify instructions, and it uses the rewrite rules to build ML code for rewriting an instruction stream. All of this information may be split across several `.kupo` files; to concatenate definitions across several `.kupo` files, simply name all the files, in order, on the `moogleize` command line. Note that the architecture definition must be given before any other `.kupo` file.

3 Syntax of Kupo Primitives

The syntax of a Kupo file is described in this section. See appendix D for a brief overview of the notation used here. This section covers the syntax of various Kupo primitives only; the syntax of Kupo top-level statements and programs is described in section 4. The topmost production is the program production, which is specified in section 4.6.

3.1 Comments

Kupo files support both OCaml-style comments `(* *)`, and C-style comments `/* */`. Comments may be used anywhere where whitespace is accepted; comments of the same type (OCaml-style or C-style) may be nested.

3.2 Basic entities

3.2.1 Primitive tokens

The following primitive tokens are defined:

cap_ident	::=	[‘A’-‘Z’] ident_tail	Uppercase identifiers
ident	::=	[‘a’-‘z’] ident_tail	Lowercase identifiers
	|	“_” [‘A’-‘Z’, ‘a’-‘z’, ‘0’-‘9’, ‘’’] ident_tail	
ident_tail	::=	[‘A’-‘Z’, ‘a’-‘z’, ‘0’-‘9’, ‘_’, ‘’’]*	
integer	::=	decimal | hexadecimal | octal	32-bit integer values
decimal	::=	(“-”)$^?$ [‘1’-‘9’] [‘0’-‘9’]*	
hexadecimal	::=	“0x” [‘0’-‘9’, ‘a’-‘f’, ‘A’-‘F’]$^+$	
octal	::=	“0” [‘0’-‘7’]*	
string	::=	“"” ([¬ ‘"’, ‘\’] | “\” [*any*])* “"”	String constants

Strings can contain standard C-style escape sequences using “\”, which will be expanded appropriately. Note that underscores may occur in ident, however the text “_” is *not* considered a

[5] This wasn’t always the case; however, with the introduction of 32-bit immediate numbers and offsets in the operands, building assembly code using ML constructors has become tedious in the backend. A number of helper functions were defined to simplify construction, but then it became difficult to determine what instruction stream was constructed by a particular helper function.

[6] This isn’t a strict requirement. However, it is recommended that the architecture definition reside in a separate file for clarity, and the remainder of this document will assume the architecture definition is in a separate file.

valid ident by the grammar. This is considered a special token, and is treated in most contexts as a wildcard token that matches anything and discards the matched value. Note also that identifiers beginning with “__” are prohibited by the grammar; all identifiers beginning with “__” are reserved by Kupo for ML code generation.

3.2.2 Typed and untyped registers

In most contexts, registers may be specified with or without their type. Typed registers tend to follow an AT&T-style syntax by preceding the register name with a special symbol or keyword to indicate the type.

untyped_reg	::=	ident	Untyped register name
typed_reg	::=	type? untyped_reg	Typed register name
type	::=	“%” | “@” | “$” | “int31” | “opr” | “lbl”	Types in the system
register	::=	untyped_reg | typed_reg	Either typed or untyped register

The types are documented below. Where possible, the AT&T symbols were used. If a type qualifier is omitted on a typed register, then the register is assumed to have type `opr`.

`%`	Integer register
`@`	Floating-point register
`$`	32-bit integer value
`int31`	31-bit (ML) integer value
`opr`	Arbitrary (complete) operand
`lbl`	Code/immediate label name

3.2.3 ML constructors

In a few locations, it is necessary to specify an ML union constructor[7]. Kupo supports a limited form of ML constructors, specified below. The ml_constructor statement specifies a full ML constructor, while ml_const_const is a special case for constant ML constructors.

ml_constructor	::=	cap_ident (ml_const_args)?	ML union constructor
ml_const_args	::=	“(” ml_const_arg (“,” ml_const_arg)* “)”	Constructor argument list
ml_const_arg	::=	cap_ident | ident	Constructor argument
ml_const_const	::=	cap_ident	ML constant constructor

3.3 OCaml code blocks

Destination operands and inline Kupo code may contain escapes to OCaml code for cases where Kupo is not sufficiently expressive to construct an instruction. Blocks of OCaml code are opaque to Kupo and must be marked with special delimiters. Escapes to OCaml code should be avoided when a reasonable alternative exists in Kupo, since they limit the ability for the Kupo compiler to manipulate the program.

There are two types of OCaml blocks, bracketed and unbracketed. Either form of block may be used where an OCaml block is expected, however by convention bracketed blocks are used for OCaml blocks embedded in Kupo code, and unbracketed blocks are used to delimit OCaml code from inline Kupo code.

Bracketed OCaml blocks are delimited with “‘{” (backquote, open-brace) and “‘}” (backquote, close-brace). Occurrences of “‘‘” (double-backquote) in the OCaml block will be emitted as a single backquote “‘”; any other occurrence of “‘” in the OCaml block is not allowed.

[7] Full ML constructors are only required for the architecture definition, but constant constructors may be used in operands.

Unbracketed OCaml blocks are delimited with "`" (backquote) on both ends of the block. Occurrences of "``" (double-backquote) in the OCaml block will be emitted as a single backquote "`"; any other occurrence of "`" in the OCaml block is not allowed.

ocaml_block	::=	ocaml_brack_block | ocaml_unbrack_block	OCaml block data
ocaml_brack_block	::=	"`{" ([¬ '`'] | "``")* "`}"	Bracketed OCaml block
ocaml_unbrack_block	::=	"`" ([¬ '`'] | "``")* "`"	Unbracketed OCaml block

3.4 Operand expressions

Kupo supports several notations for operands, depending on the context. The two major forms of operands are source and destination operands, which are respectively used in pattern match expressions and replacement expressions. Source operands are designed to pattern-match against operands in an instruction and bind matched values to temporary variables; they appear where an operand is required in a input_block statement and on the left-hand-side of a transform case. Destination operands are designed to emit a new operand, possibly using variables that were bound in a pattern-match; they appear where an operand is required in a output_block statement, and on the right-hand-side of a transform case. Destination operands also occur in inlined Kupo code.

Kupo also supports instruction operands, which are used exclusively by the architecture definition, and discussed in section 4.2.2.

3.4.1 Source operands

Source operands are used in input_block and transform statements to match an operand and optionally bind the operand to a variable name. Some of the operand patterns deconstruct an operand; in these cases, any symbols are bound to the deconstructed components, not the full operand. The wildcard token "_" matches any expression and always discards the value.

Source operands are defined by src_operand:

src_operand	::=	src_name src_modifier*	Match any operand
	|	ml_const_const	Match an ML constant constructor
	|	"%" src_name src_modifier*	Match an integer register. If src_name is a hardware register, this will match that specific register only.
	|	"@" src_name src_modifier*	Match a floating-point register. If src_name is a hardware register, this will match that specific register only.
	|	"@st"	Match top of FP stack (x86)
	|	"@st(" integer ")"	Match FP stack register integer (x86)
	|	"$" src_value	Match a numerical constant (immediate)
	|	"lbl" src_value	Match a named label (immediate)
	|	string	Match the string constant (exact)
	|	...	

Source names may either be identifiers, or the wildcard token.

src_name	::=	ident	Symbol name to bind to/match hardware register
	|	"_"	Wildcard — match anything and discard value

For source operands that expect integer values, you may specify a numerical constant to match an exact value, or you may match any integer constant and bind it to a symbol name[8]. The src_value

[8]This can be used to save the value for use in a replacement expression, or if you want to perform more sophisticated checks on the integer, such as an interval test. Note that hardware registers cannot occur where integer values are expected.

statement allows values to be specified as integers or symbols. Note that the wildcard token "_" will match any integer, as usual:

src_value	::=	src_name	Match any integer value, bind to src_name
	|	integer	Match a specific integer constant

Some basic source operands and the operands they will match are listed in figure 3.1. This figure does not include use of the source modifiers src_modifier, which are discussed below.

src_operand	Operands matched	Bindings
`_`	all operands	*none*
`name1`	all operands	`name1` := full operand
`Const1`	ML constant constructor `Const1`	*none*
`%_`	all integer registers	*none*
`%reg`	hardware register `reg` only	*none*
`%name2`	all integer registers	`name2` := integer register name
`@name3`	all float registers	`name3` := float register name
`@st`	top of floating-point stack only	*none*
`@st(3)`	register 3 on floating-point stack	*none*
`$_`	all integer constants	*none*
`$name4`	all integer constants	`name4` := integer value
`lbl name5`	any immediate label	`name5` := label name
`lbl label`	label named `label` only	*none*
`"foobar"`	exact string match	*none*

Assume `reg` is a hardware register and `label` is a defined label; other symbols are not previously defined.

Figure 3.1: Basic examples of src_operand

There are a number of custom forms of src_operand that deal with the various possible addressing modes on certain architectures (such as x86)[9]. Examples of source operands matching memory operands are given in figure 3.2.

src_operand	::=	...	
	|	"`*(`" src_name$_{base}$ "`,`" src_name$_v$ "`,`" src_value$_c$ "`,`" src_value$_{mul}$ "`)`" src_modifier*	Match `MemRegRegOffMul`, with base pointer src_name$_{base}$, variable offset src_name$_v$, constant offset src_value$_c$, multiplier src_value$_{mul}$.
	|	"`*(`" src_name$_{base}$ "`,`" src_value$_c$ "`)`" src_modifier*	Match `MemRegOff`, with base pointer src_name$_{base}$, constant offset src_value$_c$.
	|	"`*(`" src_name$_{base}$ "`)`" src_modifier*	Match `MemReg`, with base pointer only
	|	"`*`" src_name src_modifier*	Match any memory operand form, and bind the entire operand to src_name.

A non-empty list of source operands may be specified with src_operands:

src_operands	::=	src_operand ("`,`" src_operand)*	Source operands (pattern match)

Modifers may be associated with each source operand, and apply additional restrictions to the operand. These are specified using the src_modifier expression. In actual operands, these are a

[9] The exact memory operand forms are discussed in the source code (e.g. in `arch/x86/type/x86_inst_type.ml`), and their description is omitted here.

src_operand	Operands matched	Bindings
`*_`	all memory operands	*none*
`*name1`	all memory operands	`name1` := full operand
`*(_, _)`	all `MemRegOff` operands	*none*
`*(reg, _)`	all `MemRegOff` operands with base `reg`	*none*
`*(name2, _)`	all `MemRegOff` operands	`name2` := base register
`*(_, name3)`	all `MemRegOff` operands	`name3` := offset integer
`*(_, 32)`	all `MemRegOff` operands with offset 32	*none*

Assume `reg` is a hardware register; symbols with the `name` prefix are not hardware registers.

Figure 3.2: Examples of src_operand for memory operands

whitespace-separated list (possibly empty) that follow the operand pattern:

src_modifier	::=	"`!`" ident	Operand contains no occurrences of the register named in ident
	|	"`!st`"	Operand is not an FP stack register (x86)
	|	"`small`"	Operand can be read as a small (8- or 16-bit) integer value

Small registers are 32-bit (or longer) registers which can be referenced using an 8- or 16-bit register name. On the IA32 architecture, these registers are eax, ebx, ecx, and edx, all of which have 8- and 16-bit forms. Notably, the esi, edi, ebp and esp registers cannot be included here because they must always be read as a 16- or 32-bit value (they cannot be read as an 8-bit value).

3.4.2 Destination operands

The operands which appear in a replacement expression (a output_block statement, the right-hand-side of a transform case, or inline Kupo code) are oriented toward evaluation, not pattern matching. As a result, destination operands include arithmetic expressions, but cannot include the wildcard token — all elements of an operand must be fully specified here. When a symbol name is used in a destination operand, it must either be bound in a corresponding pattern-match expression, or it must be a hardware register named with the declarations statement.

Destination operands are defined by the dst_operand expression. The dst_operand expression mirrors src_operand, with the addition of OCaml code escapes:

dst_operand	::=	ident	Emit the value stored in symbol.
	|	ml_const_const	Emit an ML constant constructor.
	|	"`%`" ident	Emit the integer register named in ident. The `%` qualifier may usually be omitted, since the type system can infer this automatically.
	|	"`@`" ident	Emit the floating-point register named in ident. See above.
	|	"`@st`"	Emit the top of FP stack (x86)
	|	"`@st(`" dst_arith "`)`"	Emit the FP stack register indicated (x86)
	|	"`$`" dst_arith	Emit a numerical constant (immediate)
	|	"`lbl`" ident	Emit a program label (immediate)
	|	string	Emit a string constant
	|	("`opr`")$^?$ ocaml_block	Emit a block of OCaml code
	|	...	

When an OCaml block is embedded in an operand, it has access to all variables which are bound for the destination operand. This includes variables bound in a corresponding input pattern, and variables which are interned in an output pattern. The OCaml expression will be passed no arguments, and must return an expression which is a valid, fully-specified operand for the instruction. OCaml blocks which compute operands are generally only used by inline Kupo code, but are available in other contexts. Escapes to OCaml should not be used when a suitable alternative exists in Kupo, since OCaml blocks are opaque to Kupo and suboptimal code may result. Conventionally, bracketed OCaml blocks are used here.

As with source operands, destination operands also support a variety of exciting memory addressing operands:

dst_operand	::=	...	
	|	"*(" ident_{base} "," ident_v "," dst_arith_c "," dst_arith_{mul} ")"	Emit a `MemRegRegOffMul` operand
	|	"*(" ident_{base} "," dst_arith_c ")"	Emit a `MemRegOff` operand
	|	"*(" ident_{base} ")"	Emit a `MemReg` operand

A non-empty list of destination operands may be specified with the dst_operands expression:

dst_operands	::=	dst_operand ("," dst_operand)*	Destination operands (replacement)

Arithmetic expressions must always produce integer constants (immediate values). They can be specified with the dst_arith expression:

dst_arith	::=	ident	Read integer value bound to ident
	|	integer	Arbitrary integer constant
	|	"(" dst_arith ")"	Sub-expressions
	|	prefix_unop dst_arith	Unary operation
	|	dst_arith infix_binop dst_arith	Binary operation
prefix_unop	::=	"-" | "~" | "abs" | "log2" | "pow2"	Unary operators
infix_binop	::=	"&" | "|" | "<<" | ">>" | ">>a"	Binary bitwise operators
	|	"+" | "-" | "*" | "/"	Binary arithmetic operators

Some of the operators are described below:

"~"	Compute the bitwise negation of an integer
"&"	Compute the bitwise *and* of two integers
"|"	Compute the bitwise *or* of two integers
"abs"	Compute the absolute value of an integer
"log2"	Compute the logarithm, base 2, rounded down to the nearest integer
"pow2"	Compute the nth power of 2
"<<"	Shifts a value left by a constant
">>"	Shifts a value right by a constant, logically (unsigned)
">>a"	Shifts a value right by a constant, arithmetically (signed)

A few basic examples illustrating the features of destination operands are given in figure 3.3. Of particular note: if a symbol is bound to a register name or integer value, it may be used as an operand directly without using any of the type qualifiers (e.g. `%`, `@`, `$`). Figure 3.3 is not an exhaustive list of the possible cases where this occurs. Examples demonstrating embedded OCaml blocks are deferred to the section 4.5, which covers inline Kupo code.

3.5 Clause expressions

You can further restrict a rewrite by using a clause expression, a boolean expression which must be true in order for the rule to be applicable. The clause expression defines simple boolean expressions.

dst_operand	Variable types	Operand generated
`name1`	`name1`: full operand	any operand
`name2`	`name2`: integer register name	any integer register
`name3`	`name3`: float register name	any float register
`name4`	`name4`: integer value	any integer constant
`Const1`	*none*	ML constant constructor named Const1
`%name5`	`name5`: integer register name	any integer register
`@st`	*none*	top of floating-point stack
`$name6`	`name6`: integer value	any integer constant
`$abs name7`	`name7`: integer value	any nonnegative integer constant
`lbl name8`	`name8`: program label	any program label
`"foobar"`	*none*	exact string constant

Figure 3.3: Basic examples of dst_operand

The syntax of a clause expression is:

clause	::=	"(" clause ")"	Sub-expression
	|	dst_arith relop dst_arith	Relational operator applied to integers
	|	"`is_power2`" dst_arith	True if the integer is a power of 2
	|	clause boolop clause	Logical combinator
relop	::=	"`=`" | "`<>`" | "`<`" | "`<=`" | "`>`" | "`>=`"	Relational operator
	|	"`<u`" | "`<=u`" | "`>u`" | "`>=u`"	Unsigned relational operator
boolop	::=	"`&&`" | "`||`"	Logical operator

The comparison operators "<", "<=", ">", and ">=" are all signed comparisons, that is, they assume their operands are signed 32-bit integers. Use the operators "<u", "<=u", ">u", and ">=u" for unsigned comparisons.

4 Syntax of Kupo Statements and Programs

This section covers the syntax of top-level Kupo statements and a complete Kupo program. The syntax of Kupo primitives is described in section 3. The topmost production is the program production, which is specified in section 4.6.

4.1 Importing ML modules

Kupo files support the open statement for importing ML modules. The architecture definition file must open any ML modules required to obtain the type definition for the instruction set. For example, `arch/x86/type/x86_arch.kupo` must contain:

```
open X86_inst_type
open Frame_type
```

The other Kupo files generally only need to open the module that will be compiled from the architecture definition file. As an example, `arch/x86/opt/x86_peephole_basic.kupo` simply contains:

```
open X86_arch
```

The syntax for an open statement is:

open	::=	"`open`" cap_ident	Open an ML module

4.2 Architecture definition

The following statements usually occur only in the architecture definition file.

4.2.1 Declaring hardware registers and context variables

The hardware register set and context variables are specified by the declarations statement. Typically, only one declarations block should be defined for the architecture definition; it must be defined before any register is referenced. Individual Kupo files may use additional declarations to define global or common variables which are in the scope of the Kupo file; in particular, inline Kupo code will occasionally use a declarations statement to define commonly used operand names to avoid having to declare the variables in each block of Kupo code.

The declarations statement may be used to declare variables of any type. Each variable declared is declared in OCaml code (often in another module), and should be in scope to avoid OCaml compile errors.

declarations ::= "`declarations`" "`:`" (typed_reg "`;`")$^+$ Set of variable declarations

Any variable in a pattern which does not match a declared name is assumed to be a wildcard match, that binds any structurally compatible value to the variable name given. Variable names declared in declarations can never be used for wildcard matches.

A sample declarations statement is given in figure 4.1.

```
declarations:
   (* Integer hardware registers *)
   %reg1;
   %reg2;
   %reg3;
   %reg4;
   %flags;

   (* Floating-point hardware registers *)
   @float1;
   @float2;

   (* Useful constants defined by runtime *)
   int31 sizeof_int32;
   int31 sizeof_int64;
   int31 sizeof_pointer;
   int31 callee_header_stack;
```

Figure 4.1: Example use of declarations

4.2.2 Instruction definition

To define instructions, you should already have an ML type file for the assembly instructions that is reasonably compliant with AT&T[10]. Kupo requires you to define the instruction set in the architecture definition file to maintain independence from any specific architecture[11].

[10] In particular, the instructions should have *dest-source*-form operands, and operands should be compatible with the operands used x86, even if not all addressing modes are available on the particular archicture. This is an area of Kupo that needs to be cleaned up.

[11] At present, there are still internal ties to the x86 architecture in Kupo; in particular, the operands are tied to the x86-form operands.

To define instructions, use the instructions statement. More than one of these blocks may be defined in the architecture definition. You may include a whitespace-delimited list of properties that apply to all instructions listed in the block; each instruction in the block may append additional properties to this list, that apply only to that instruction. See `arch/x86/type/x86_arch.kupo` for an example.

instructions	::=	"`instructions`" inst_prop* "`:`" inst_line$^+$	Block of assembly instructions
inst_prop	::=	ident	Property name
inst_line	::=	ident$_{name}$ inst_operands$^?$ "`->`" ml_constructor $\left(\text{":" inst\_prop}^+\right)^?$ "`;`"	Instruction definition
inst_operands	::=	inst_operand ("`,`" inst_operand)*	List of operands to instruction
inst_operand	::=	register	Operand to the instruction

Recall that register may include an optional type qualifier. For each line in the instruction block, a new Kupo instruction will be defined with the name ident$_{name}$ and with the indicated arguments; the corresponding ML constructor is given to the right of the arrow. An instruction name can only be bound once, and should not conflict with the names of any hardware registers. Kupo arguments for an instruction definition may contain a type restriction:

- An untyped argument matches any operand
- `%` restricts the operand to an integer type (register or constant value)
- `@` restricts the operand to a float type (register or constant value)
- `$` restricts the operand to be a 32-bit integer constant (immediate)
- `int31` restricts the operand to be an ML (31-bit) integer constant (immediate)
- `lbl` restricts the operand to be a program label (immediate)
- `opr` matches any operand

Properties are used to build classification functions for the instructions. The classification function for property inst_prop reports true if the instruction is either listed with the property, or is in a instructions block which declares inst_prop as a block property. These classification functions are invoked by rewrite rules, and may also be used by external ML code to help classify instructions.

Instructions which are trivial[12] should be tagged with a special property name, so Kupo knows which instructions can be ignored in rewrite rules. Typically, this property is called "`trivial_inst`", however you can use any valid property name. To identify which property corresponds to the trivial instructions, you should use the trivial statement:

trivial	::=	"`trivial`" inst_prop	Define the trivial property name

Finally, the instruction definition should declare the ML type for the instruction set. This is done with the instruction_type statement. Exactly one declaration of the instruction type should occur in the architecture definition. If the ML type for instructions is not declared, then Kupo will not be able to build interface files.

instruction_type	::=	"`instruction_type`" ident	Declare the type of instructions

An example using instructions, trivial, and instruction_type is given in figure 4.2. In this figure, all instructions defined have the "`integer_inst`" property, but only `nop` has the "`trivial_inst`" property.

Note: the implementation is sensitive to the order instructions are defined in, when more than one Kupo instruction use the same ML constructor. See section 7.1 for details.

[12] Trivial instructions are instructions with absolutely no evaluation content. These include `NOP` and any "comment" instructions.

```
instructions integer_inst:
   nop             -> NOP  :  trivial_inst (* instruction taking no arguments *)
   movb  dst, src  -> MOV (Byte, dst, src) (* move byte, using ML constructor *)
   movw  dst, src  -> MOV (Word, dst, src) (* move word, using ML constructor *)
   add   dst, src  -> ADD (dst, src)       (* add two ordinary operands *)
   shl   dst, $src -> SHL (dst, src)       (* shift left by a constant *)
   and   dst, src  -> AND (dst, src)       (* compute bitwise and of values *)
   cmp   dst, src  -> CMP (dst, src)       (* compare two values; set flags *)
   jcc   cc, label -> JCC (cc, label)      (* branch; cc matches ML const. *)

(* Identify nop as a trivial instruction to Kupo *)
trivial trivial_inst

(* Identify the ML type containing instructions (e.g. NOP, MOV, ...) *)
instruction_type inst
```

Figure 4.2: Example use of instructions, trivial, and instruction_type

4.2.3 Transform functions

Transforms are functions that allow you to take an instruction that matches a particular pattern and rewrite it into another instruction, possibly using the same operands. Transforms are a convenient shorthand for manipulating a class of related instructions[13] for rewrite rules which would otherwise have to be expressed using many similar input/output patterns[14].

The definition of a transform function consists of pattern matches on single instructions, and are expressed using a notation similar to the instruction patterns found in rewrite rules. When a transform is used in a rule, `moogleize` will attempt to apply all operand qualifiers that appear on the left-hand-side of the transform rewrite as restrictions on the input pattern; however, certain qualifiers that are legal for instructions in a rewrite rule cannot be used in transform functions.

Transform functions are identified by a unique name, and may take 0 or more integer arguments in addition to the instruction they are manipulating[15]. When the transform name is used in a rewrite rule, each instruction rewrite listed in the transform is applied to the rewrite rule, in order. If the transform function contains N cases, then $O(N)$ new rewrite rules will be generated internally by `moogleize`.

A new transform function is declared using the transform statement:

transform	::=	"`transform`" trans_name trans_args$^?$ ":" trans_line$^+$	Define a new transform function
trans_name	::=	ident$_{name}$	Name of the transform function
trans_args	::=	ident ("," ident)*	Extra arguments to transform function
trans_line	::=	ident src_operands$^?$ "->" ident dst_operands$^?$ ";"	Match case (one instruction)

The exact forms for src_operands and dst_operands are documented in sections 3.4.1 and 3.4.2. An example definition of a transform function is given in figure 4.3.

[13] For example, x86 floating-point operations, many of which have a uniform operand syntax.

[14] For example, a rewrite rule that wishes to invert the condition code of a branch instruction can use a transform function to perform the inversion; otherwise we would have to list 10 separate rules to invert each possible case on the x86 architecture.

[15] The instruction being manipulated is an implicit argument. Explicit arguments to transform functions are rarely used, but come up for the x86 architecture in floating-point stack manipulation.

```
(* Inverts the condition code on a branch instruction *)
transform cond_invert:
   jcc EQ,  label -> jcc NEQ, label;
   jcc NEQ, label -> jcc EQ,  label;
   jcc LT,  label -> jcc GE,  label;
   jcc LE,  label -> jcc GT,  label;
   jcc GT,  label -> jcc LE,  label;
   jcc GE,  label -> jcc LT,  label;
```

Figure 4.3: Example use of transform

Note: the implementation is sensitive to the order that transform cases are defined in, when more than one case reduces to the same ML constructor. See section 7.2 for details.

4.3 Rewrite rule features

Rewrite rules are a core feature of Kupo. They allow you to rewrite a block of instructions matching a pattern (the input block), into a new block of instructions using a replacement block (the output block). The features of a rewrite rule are briefly summarized in the following subsections; the syntax is discussed in section 4.4.

4.3.1 Input block features

An input block supports:

- A destructuring pattern match on individual instructions (possibly subject to instruction constraints)
- Matching of more than one instruction in the stream
- Wildcards that match an arbitrary number of instructions (possibly subject to instruction constraints)
- Matching a class of instructions defined by a particular property
- Matching a class of instructions that are accepted by a transform function
- Restrictions based on liveness analysis of the program registers
- Ignoring trivial statements (defined by the property specified in the trivial statement). If the rewrite is applied, then any trivial statements are stripped out of the original instruction stream and placed to the top of the modified instruction stream, before the rewritten instructions.

4.3.2 Output block features

An output block can reference variables bound by the destructuring pattern match from the input block(s), and can apply transformations from a transform statement. Output blocks are also able to refer to entire instructions or lists of instructions from the input, and can copy instructions from the input stream verbatim. This feature is particularly useful with the wildcard instruction match offered by input blocks. Finally, output blocks may intern new temporary register names to store intermediate results.

4.3.3 Rewrite rule features

Rewrite rules consist of:

- One or more input blocks, each possibly restricted by a "`when`" clause that restricts when the particular input block can be used
- One or more output blocks, each possibly restricted by a "`when`" clause that restricts when the particular output block can be applied

For each rewrite rule, `moogleize` will process the input blocks in order, attempting to find one that matches. If it finds a matching input block, then it will process the output blocks in order and apply the first one whose conditions are satisfied. Therefore, a rewrite rule will be applied if `moogleize` can find one input block and one output block in the rule that can be applied.

For debugging purposes, rewrite rules may be named. A global name may be defined for a block of rewrites, and names may also be applied to specific rewrite rules. Names can be emitted in the instruction stream to identify the sequence of rewrite rules were applied.

A few basic examples of rewrite rules are given in figure 4.4.

```
(* Deletes NOP from the output stream *)
input "Remove NOP":
   nop;
output:
   (* No instructions listed here *)

(* Using multiple input rules with a single output... *)
name "Removing trivial arithmetic"
input:
   add   _, $0;
input:
   shl   _, $0;
output:
   (* No instructions listed here *)

(* A simple rewrite, note how v is bound in the input rule *)
input "ADD/SHL optimization":
   shl   %v,   $1;
output:
   add   v,    v;
```

Figure 4.4: Basic examples of rewrite rules using rewrite

4.4 Rewrite rule syntax

Rewrite rules are specified by the rewrite statement. Rewrite rules may specify a name explicitly; if no name is specified, they inherit the default name. The name statement will change the default name used by rewrite rules following it, until the next name statement.

name_value	::=	“"” string “"”	Valid value for a rule name
name	::=	“`name`” name_value	Change default name for rewrite rules
rewrite	::=	input_block$^+$ output_block$^+$	Define a new rewrite rule
input_block	::=	“`input`” name_value$^?$ “`:`” input_pattern (“`when`” clause)$^?$	Define a new input block
output_block	::=	“`output`” (“`when`” clause)$^?$ “`:`” output_pattern	Define a new output block

If the “`when`” clause is specified for an input_block or output_block, then it adds further restrictions to that block *only*, based on the clause expression given. Clause expressions are defined in section 3.5.

An output_block can only refer to symbols that are bound in *all* input_block statements for the rewrite rule, or to symbols that were declared in an earlier declarations statement. This is because the term rewriter may select any of the input_block statements to apply with the output_block.

4.4.1 Input patterns

Each input pattern must match at least one instruction. The syntax of input patterns is defined by the input_pattern statement:

input_pattern	::=	(input_line “`;`”)$^+$	Input pattern; must match ≥ 1 instructions
input_line	::=	opcode_name src_operands$^?$ (“`:`” inst_modifier$^+$)$^?$	Match an instruction named opcode_name
	|	“_” (“`:`” inst_modifier$^+$)$^?$	Match exactly one instruction of any type
	|	“`*`” (“`:`” inst_modifier$^+$)$^?$	Match 0 or more instructions of any type
opcode_name	::=	ident	Name of an opcode bound in instructions
	|	cap_ident	Name of any defined ML constructor

The inst_modifier statement is defined in section 4.4.3, and provides access to instruction properties and the liveness information computed by Kupo. In order for an input_line to match a particular instruction, all instruction modifiers must hold for that instruction.

Each line of an input pattern is one of three cases:

- The most common case is to match a specific instruction. In this case, you may match against an instruction defined by an instructions statement, or you may match a specific ML constructor. In either case, opcode_name must be bound, and the number of operands specified here must agree with the definition for the instruction. See section 3.4.1 for details on src_operands. Any symbols in the operands that are not hardware registers will be bound to their corresponding values.
- The “_” token matches a single instruction, regardless of its form. Note that no variables can be bound here; however, you may still use instruction modifiers here.
- The “`*`” token matches 0 or more instructions, regardless of their form[16]. Note that no variables can be bound here; however, you may still use instruction modifiers here. Any

[16] There are two technicalities for “`*`”. First, while it can match zero instructions, input rules must match at least one instruction; therefore if “`*`” is used as the only line of an input block, it will always match at least one instruction. Also, due to limitations in the current implementation, “`*`” will match at most `Moogle_state.rule_max_star` instructions. I intend to remove this limitation in the next major rewrite of Kupo.

instruction modifiers specified must hold for *all* instructions that are matched by "`*`"[17]. "`*`" will match the fewest number of instructions possible such that the entire pattern still holds. Simple examples using "`*`" are given in figure 4.5.

Most restrictions that will be applied to an input block are specified in the input pattern, or the "`when`" clause of the input block. The output block can impose a few additional restrictions — for example, an output block may specify its own "`when`" clause, or invoke a transform function that restricts the form of certain instructions.

4.4.2 Output patterns

Each output pattern can print zero or more instructions. The syntax of output patterns is defined by the output_pattern statement:

output_pattern	::=	(output_line "`;`")*	Output pattern; print ≥ 0 instructions
output_line	::=	opcode_name dst_operands$^?$	Emit instruction with given operands
	|	"`copy`" integer ("`,`" integer)*	Copy instructions from input pattern
	|	trans_name integer ("`,`" dst_arith)*	Invoke a transformation function
	|	"`new`" new_reg ("`,`" new_reg)*	Create a new register name
new_reg	::=	("`%`" | "`@`") untyped_reg	Register to be interned by "`new`"

Each line of an output pattern is one of four cases:

- The common case is to print a specific instruction. In this case, you may print an instruction defined by an instructions statement, or you may print a specific ML constructor. In either case, opcode_name must be bound, and the number of operands specified here must agree with the definition for the instruction. See section 3.4.2 for details on dst_operands. Any symbols that occur in the operands must be hardware registers, bound by all of the input blocks for the rewrite rule, or bound by a "`new`" line in this output block (see below).
- The "`copy`" token copies instructions from the input block that matched. The most common use of this is to copy the instructions matched by a "`_`" or "`*`" token; however, it may be used to copy any instruction. When given one integer argument n, "`copy`" copies the instruction(s)[18] matched by the n'th line of the input pattern[19]. When given a list of integers, instructions are copied from the input pattern using the order the integers are listed in. An example of using "`copy`" in conjunction with "`*`" is given in figure 4.5.
- A transform function may be invoked by name. The first argument to the transform function must be an integer n, specifying which line of the input pattern to operate on. The n'th line of the input pattern must not be a "`*`" in this case. Additional arithmetic expressions are passed as arguments to the transform function. See section 4.2.3 for more information on transform functions. If the instruction does not match any case in the transform function, then this output rule will not be applied. An example using transforms is given in figure 4.6.
- The "`new`" token creates a new integer or floating-point register name. This is useful for generating temporary registers to store intermediate results in. The generated register will always be a dead register. After register allocation, it may not be possible to find a register which is dead; if that is the case, then this output rule will not be applied.

[17] It is worth noting that peephole optimizations using "`*`" are responsible for the very long compile times of the generated ML files in MCC. The current implementation of "`*`" is to generate input rules that match specific numbers of instructions; while this is straightforward for `moogleize` to generate, OCaml does not deal gracefully with the massive pattern match expressions that are produced. In the next major rewrite, "`*`" will be processed by a recursive function, greatly simplifying the ML patterns that are generated and hopefully improving compile times.

[18] Usually "`copy`" copies a single instruction; when the n'th line of the input block is "`*`" then all instructions matched by the wildcard are copied.

[19] n here counts from 1 — the first line of the input pattern is line 1, the second line is line 2, etc.

```
(* This simple rewrite catches an arithmetic operation that is undone,
   even when there are other instructions between the two instructions.
   If add influences the flags on your architecture, then you need the
   dead flags modifier, since we are going to delete the addition. *)
input "Removing redundant addition":
   add   %v,   $value1     :  dead flags;
   *                       :  ignores v;
   add   %v,   $value2     :  dead flags;
(* Note: this when binds to the input block, not the output block. *)
when value1 = -value2
output:
   (* Copy all the instructions matched by the wildcard *)
   copy  2;

(* Here's a better version, since it folds together the two add
   statements, regardless of the values. *)
input "Removing redundant addition, try 2":
   add   %v,   $value1     :  dead flags;
   *                       :  ignores v;
   add   %v,   $value2;    (* No need to check flags here *)
output:
   copy  2;
   add   v,    $value1 + value2;
```

Figure 4.5: Using "*" and "copy" in a rewrite statement

```
(* Simple transform that negates a test for equality. *)
transform cond_invert_equality:
   jcc   EQ,  label  -> jcc   NEQ, label;
   jcc   NEQ, label  -> jcc   EQ,  label;

(* This optimization corrects an expression common in runtime-generated code.
   AND on IA32 sets ZF, therefore the comparison statement is redundant.  The
   transform function imposes restrictions on the wildcard match.  *)
input "Performing AND flags optimization":
   and   v,    $1;            (* Sets zero flag if LSB of v is clear *)
   cmpl  v,    $1;            (* Sets zero flag if LSB of v NOT clear *)
   _;
output:
   copy  1;                   (* Ignore the redundant comparison *)
   cond_invert_equality 3;    (* Need to invert the condition code *)
```

Figure 4.6: Using transform functions in a rewrite statement

4.4.3 Instruction modifiers

Instruction modifiers are used in input patterns, and allow you to require that matched instructions have a particular property, or follow certain liveness characteristics. Instruction modifiers are given as a whitespace-separated list (possibly empty) of inst_modifier statements:

inst_modifier	::=	"`preserves`" preserves_item	Instruction preserves named item
	|	"`preserves`" "`[`" preserves_item ("`;`" preserves_item)* "`]`"	Instruction preserves listed items
	|	"`ignores`" ignores_item	Instruction ignores named item
	|	"`ignores`" "`[`" ignores_item ("`;`" ignores_item)* "`]`"	Instruction ignores listed items
	|	"`dead`" dead_item	Named item is dead after instruction
	|	"`dead`" "`[`" dead_item ("`;`" dead_item)* "`]`"	Named items are dead after instruction
	|	"`endscope`" ident	End of the live range for source ident
	|	"`irrelevant`"	Instruction has no computational effect
	|	inst_prop	Instruction has the named property
preserves_item	::=	"`memory`" | ident	
ignores_item	::=	"`memory`" | ident	
dead_item	::=	ident	

Instruction modifiers allow you to place the following types of restrictions on the instruction:

- The "`preserves`" modifier holds when a specified item is preserved by the instruction. If the item is "`memory`", then the modifier is true when memory is not modified by the instruction; if the item is a register name, then the modifier is true when the register is not in the instruction's defs list. This modifier can also accept a list of items, using typical ML list notation.
- The "`ignores`" modifier holds when a specified item is ignored by the instruction. If the item is "`memory`", then the modifier is true when memory is not used at all by the instruction (the instruction only operates on registers and internal state). If the item is a register name, then the modifier is true when the register is not in the instruction's uses list, and the register is not in the instruction's defs list. This modifier can also accept a list of items, using typical ML list notation. An example of the "`ignores`" modifier is given in figure 4.5.
- The "`dead`" modifier holds when a specified register is not live out of the instruction. The modifier is true when the register does not appear in the instruction's live-out list. This modifier can also accept a list of registers, using typical ML list notation. An example of the "`dead`" modifier is given in figure 4.5.
- The "`endscope`" modifier holds when a specified register's scope ends at this instruction. The modifier is true when the register is not in the instruction's live-out list, *or* the register is in the instruction's defs list[20].
- The "`irrelevant`" modifier holds when the instruction has no computational content; that is, when none of the instruction's defs appear in the instruction's live-out list.
- A property that was defined in a instructions statement may also be specified. The modifier holds if the instruction has the specified property.

[20] In other words, if the register is redefined at this instruction, then it counts as the end of scope for the previous definition of the register.

4.5 Inline Kupo expressions

4.5.1 Overview of inline Kupo

Inline Kupo code is Kupo code embedded in OCaml code which is used to generate instruction streams. A special escape sequence is used to mark OCaml code which will be emitted verbatim. Each sequence of Kupo code builds a list of instructions which is returned to the enclosing OCaml code. Instruction streams are usually more clear and concise when written as Kupo code, compared to using standard ML constructors. As a result, inline Kupo can increase the readability and maintainability of OCaml code which transforms instruction streams where rewrite rules may not be suitable.

Inline Kupo code consists of alternating blocks of OCaml code and Kupo code. Each sequence of Kupo code is contained in a private scope; variables defined in one Kupo block are not visible in other Kupo block. The Kupo block is allowed to reference variables that are in scope in the surrounding OCaml code. A Kupo block may operate on an existing instruction stream, or it may construct a new stream. Each Kupo block will return the modified instruction stream, and optionally a list of operands which are defined in the Kupo block.

If the instruction set has ML type `inst`, then the corresponding instruction stream has ML type `inst Listbuf.t`.

4.5.2 Syntax of inline Kupo statements

A new inline Kupo statement is defined using the inline_kupo statement. The inline_kupo statement consists of a sequence of alternating ocaml_block and kupo_block blocks. By convention, the OCaml blocks are usually unbracketed blocks, therefore each Kupo block is delimited from the surrounding OCaml text by "‘". Two kupo_block blocks may not be immediately adjacent[21].

inline_kupo	::=	"`inline_kupo`" "`:`" "`begin_ocaml_code`" inline_kupo_expr "`end_ocaml_code`"	Inline Kupo statement
inline_kupo_expr	::=	kupo_block$^?$ (ocaml_block$^+$ kupo_block$^?$)*	Alternating blocks

If multiple inline_kupo statements are specified, then they will be concatenated in order in the final output file. Similarly, adjacent ocaml_block blocks are concatenated together. An example of the block structure is given in figure 4.7.

4.5.3 Syntax of Kupo blocks

A Kupo block consists of a prelude which indicates how we are manipulating the instruction stream, followed by a pattern which resembles output patterns. Kupo patterns are more expressive than rewrite output patterns, and may contain escapes to OCaml code which compute single instructions or manipulate the instruction stream directly.

A Kupo block is specified using the kupo_block statement:

kupo_block	::=	kupo_prelude "`:`" kupo_pattern	Block to manipulate inst. streams
kupo_prelude	::=	kupo_operator ("`->`" dst_operands)$^?$	Stream to manipulate/what to return
kupo_operator	::=	"`+`" ident	Append to existing instruction stream
	|	"`new`" ident$^?$	Create a new instruction stream

The prelude determines what operation is performed on the instruction stream. The following operations are defined:

[21] It is not reasonable to have two adjacent Kupo blocks, since the first block would be unable to return its value. A logical interpretation could be introduced here, such as returning a tuple of all adjacent Kupo blocks, however at this time no such interpretation has been applied.

```
inline_kupo:
begin_ocaml_code`
(* Note the backquote above: this comment is in an OCaml block! *)

(* ... *)
let insts =
   `new:
      (* Now we are in a Kupo block; in this case, we are building a new
         instruction stream.  All names are hardware integer registers;
         there are no dependencies to variables in surrounding OCaml code
         (other than the hardware registers which are declared in frame).  *)
      add   reg1,    reg2;
      add   reg1,    reg3;`
(* This comment is back in OCaml code...  The newly constructed stream is
   returned to the variable insts, defined in OCaml code.  *)

(* ... *)
`end_ocaml_code
(* Again, note the backquote: this is after the end of the statement.  *)
```

Figure 4.7: Example block structure for inline Kupo

- "`+name:`" append the instructions listed in kupo_pattern to an existing instruction stream, named "`name`". The identifier must be bound in the scope of the surrounding OCaml code, and must have ML type `inst Listbuf.t`. The modified instruction stream is returned, also with type `inst Listbuf.t`.
- "`new:`" create a new, unnamed instruction stream. OCaml escapes within kupo_patterncannot refer to the instruction stream being constructed. The instruction stream is returned with type `inst Listbuf.t`.
- "`new name:`" create a new instruction stream named "`name`". By naming the instruction stream, OCaml escapes in kupo_pattern can refer to the instruction stream and manipulate it directly. Both "`name`" and the returned instruction stream will have type `inst Listbuf.t`.

In addition, the prelude may return other values in addition to the instruction stream. For example, the prelude "`new -> u, v:`" will return a triple consisting of the instruction stream, the operand bound to "`u`", and the operand bound to "`v`" within kupo_pattern. This is particularly useful for returning complex operands which are computed in kupo_pattern which are nontrivial to express in OCaml, and for returning operands representing registers which are interned within kupo_pattern.

4.5.4 Syntax of Kupo patterns

Kupo patterns resemble the output patterns in rewrite rules, defined by output_pattern[22]. Kupo patterns do not include "`copy`" and transform functions since these operations are undefined in the absence of an input pattern. Instead, Kupo patterns contain variable declarations and definition, and OCaml escapes to compute new instructions or manipulate the instruction stream.

[22] In fact, kupo_pattern and output_pattern use exactly the same rules in the parser; validity of lines for each case is determined by a later stage in `moogleize`.

Kupo patterns are defined by the kupo_pattern statement:

kupo_pattern	::=	(kupo_line "`;`")*	Kupo pattern; print ≥ 0 instructions
kupo_line	::=	opcode_name dst_operands$^?$	Emit instruction with given operands
	|	"`new`" new_reg ("`,`" new_reg)*	Create a new register name
	|	"`dcl`" typed_reg ("`,`" typed_reg)*	Declare variables bound in OCaml code
	|	"`def`" var_expr ("`,`" var_expr)*	Define a 32-bit integer expression
	|	kupo_ocaml_op ocaml_block	Escape to OCaml code
	|	kupo_ocaml_op ident	Escape to OCaml value
var_expr	::=	ident "=" dst_arith	Bind a variable to a computed value
kupo_ocaml_op	::=	"`add_inst`" | "`add_listbuf`"	Methods used to escape to OCaml
	|	"`add_list`" | "`modify_listbuf`"	

The "`new`" operator is the same as in output_pattern, described in section 4.4.2. The operations specific to kupo_pattern are described below:

- The "`dcl`" token is used to declare variables which are defined in the surrounding OCaml code. This is similar to the declarations statement, except that the scope of variables declared using "`dcl`" is limited to the current Kupo block.
- The "`def`" token is used to bind variables to 32-bit computed value. This is useful for performing evaluation of common subexpressions based on values in the surrounding OCaml code. Currently, the types of value must always be 32-bit integers.

In addition, kupo_pattern patterns may escape to OCaml code to compute instructions or manipulate the instruction stream. By convention, the OCaml blocks are bracketed blocks. OCaml blocks have access to variables defined in the OCaml code surrounding the Kupo block, as well as variables which are defined in the Kupo block. If the instruction stream is named, then the OCaml block can refer directly to the instruction stream. Such references will return the instruction stream after it has been modified by all previous lines in the Kupo block. The interpretation of the OCaml block is determined by the operator that prefixes the block, described below:

- For "`add_inst`", the OCaml block is an expression which computes exactly one instruction, of type `inst`. This instruction is added to the end of the current instruction stream.
- For "`add_list`", the OCaml block is an expression which computes a list of instructions, of type `inst list`. The list is appended to the end of the current instruction stream.
- For "`add_listbuf`", the OCaml block is an expression which computes a new instruction stream of type `inst Listbuf.t`. The instruction stream's contents will be appended to the end of the current instruction stream.
- For "`modify_listbuf`", the OCaml block is an expression which computes an instruction stream of type `inst Listbuf.t`, which *replaces* the current instruction stream for the remainder of the Kupo pattern. Since it is very likely the OCaml code will want to manipulate the current instruction stream, the instruction stream must be named explicitly in the prelude[23].

Examples of using OCaml escapes are given in figure 4.8.

Note: normally, when a variable is interned within a Kupo block and returned, it is always returned as an operand. For example, a register name interned by "`new %r`" is returned as the operand `Register r`. However, OCaml escapes within the Kupo block will see the symbol name, not the operand. This is discussed in section 7.4.

Note: variable definitions via "`def`" propagate to the top of a Kupo block automatically; as a result it is unwise to attempt to shadow "`def`" statements in a Kupo block. See section 7.4 for more information.

[23] The current instruction stream is *not* passed as an argument to the OCaml code.

```
inline_kupo:
begin_ocaml_code`

let op = Register reg1
let n  = 32
let insts1, r_op =
   `new insts -> r:
      (* insts is bound as an instruction stream in this block. *)
      (* Declare an operand named op, and a 31-bit value named n.
         While we're at it, let's also create a new int register r. *)
      dcl   op, int31 n;
      new   %r;

      (* A typical instruction in a kupo block. *)
      add   op,      n;

      (* Escape to ML code to build the same instruction.  Typically,
         the OCaml block given here will be a bracketed block.  Note
         that it must explicitly construct the immediate operand.  *)
      add_inst `{ ADD (op, ImmediateNumber (OffNumber (Int32.of_int n))) `};

      (* Escape to ML code again; this modifies the instruction stream
         in-place; after this call, we will have *three* ADD insts.  *)
      modify_listbuf `{ Listbuf.add insts (ADD (op,
                           ImmediateNumber (OffNumber (Int32.of_int n)))) `};

      (* We can escape to ML code for a single operand as well, such
         as the following.  Granted, you'd never want to write this
         particular expression, but it's common to do this for string
         operands...  *)
      add   `{ op `},  n;

      (* Move the result to the register we created. *)
      movw  r,       op;`
(* At the end of the Kupo block, we return an instruction stream with
   five instructions, and we also return an operand which contains the
   new symbol that the Kupo code generated, stored in r_op.  r_op has
   an operand type, *not* a symbol type.  *)

`end_ocaml_code
```

Figure 4.8: Using OCaml escapes in inline Kupo code

4.6 Syntax of a complete program

A Kupo file is a program, which in turn is a sequence of zero or more statements. The productions for a complete program are given below:

program	::=	statement*	Top production for Kupo files
statement	::=	open	Import ML modules (section 4.1)
	|	declarations	Declarations (section 4.2.1)
	|	instructions	Instruction definition (section 4.2.2)
	|	trivial	Trivial property (section 4.2.2)
	|	transform	Transform function (section 4.2.3)
	|	rewrite	Rewrite rule (sections 4.3-4.4)
	|	name	Set default name (sections 4.3-4.4)
	|	inline_kupo	inline Kupo code (section 4.5)

5 Interfacing to Kupo from OCaml Files

This section describes how to invoke the exported Kupo functions from OCaml code.

5.1 Invoking property classifiers and transform functions

When compiling with `moogleize -insts`, the property classifiers and transform functions are emitted. These functions are called internally by other `.kupo` files, and can also be invoked from other OCaml files. For each property `prop` declared in the architecture definition, a function is declared with the following signature (assuming the ML type for instructions is `inst`), which returns true if the specified instruction has property `prop`:

```
val prop : inst -> bool
```

For each transform function `trans` taking integer arguments $\mathtt{n}_1 \ldots \mathtt{n}_k$ which is declared in the architecture definition, a function is declared with the following signature, which takes an instruction and returns a modified instruction. Note that the transform function raises a `Failure` exception if the given instruction does not match any of the cases defined for the transform.

```
val trans : inst -> int32 -> ... -> int32 -> inst
                    (repeated k times...)
```

5.2 Invoking rewrite rules

When compiling with `moogleize -rules`, an interface to access all rewrite rules in the `.kupo` file is declared. The following functions are declared:

```
val rule_data_create   : inst Listbuf.t -> inst live list -> Moogle_lib.rule_data
val rule_build_default : Moogle_lib.rule_data -> Moogle_lib.rule_data
val process_rules      : bool -> Moogle_lib.rule_data ->
                         (bool -> Moogle_lib.rule_data -> Moogle_lib.rule_data) ->
                         Moogle_lib.rule_data
```

The functions are briefly described below:

- `rule_data_create out_buf in_insts` creates a new set of rule data. `in_insts` are the instructions to apply rewriting to, and `out_buf` is the instruction stream to append the revised instructions onto.

- `process_rules nreg rd fallthrough` applies the rewrite rules defined in the module to the rule data in `rd`. If no rewrite rule can be applied, then the result of `(fallthrough nreg rd)` is returned. A new rule data structure `rd'` is returned. `rd'.Moogle_lib.rd_insts` contains a list of instructions which have not yet been rewritten, and `rd'.Moogle_lib.rd_buf` contains the current instruction stream.
- `rule_build_default rd` returns a rule data structure `rd'` where one instruction in the input list is copied to the output instruction stream. This should be called only if none of the other rewrite rules can be applied; in effect, it should be the last `fallthrough` function for `process_rules`.

A simple example of invoking rewrite rules to transform an entire instruction stream is given in figure 5.1. The modules which contain rewrite rules are `Rewrite_A` and `Rewrite_B`; in the figure, rules in `Rewrite_A` will be applied before rules in `Rewrite_B`.

```
(* rewrite_insts
   Run rewrites on a block of instructions.  *)
let rec rewrite_insts nreg ph =
   match ph.Moogle_lib.rd_insts with
      inst :: insts ->
         let try_default   _    ph = Rewrite_A.rule_build_default ph in
         let try_rewrite_b nreg ph = Rewrite_B.process_rules nreg ph try_default in
         let try_rewrite_a nreg ph = Rewrite_A.process_rules nreg ph try_rewrite_b in
            rewrite_insts nreg (try_rewrite_a nreg ph)
    | [] ->
         ph

(* rewrite_insts
   Interface which accepts an input stream and builds an output stream.  If nreg
   is true, then the rewrite rules are allowed to intern new register names. The
   insts is a list of liveness information for the input instruction stream.  *)
let rewrite_insts nreg insts =
   let buf = Listbuf.empty in
   let ph = Rewrite_A.rule_data_create buf insts in
   let ph = rewrite_insts nreg ph in
      ph.Moogle_lib.rd_buf
```

Figure 5.1: Using rewrite rules from ML code

5.3 Invoking functions in an inline Kupo file

In inline Kupo code, the OCaml blocks which contain Kupo blocks are emitted verbatim. Therefore, you can access functions declared in the OCaml blocks like any other ML function. It is recommended that you write the signature file yourself for any Kupo file which contains inline Kupo code.

6 Examples of Using Kupo

Currently, there are no advanced examples here. The best examples to review are the `.kupo` files under `arch/x86`; these files are fairly well-documented, and stress many of the features of Kupo.

7 Kupo Implementation and Caveats

Kupo is implemented with the `moogleize` compiler, which resides in the `arch/x86/moogle/` directory. `moogleize` is a two-stage compiler; the first stage is an AST expression which is produced by the parser, and defined in `moogle_ast_type.ml`. The expression is converted into a program IR which is easier to produce ML code from; the program IR is defined in `moogle_prog_type.ml`. The program IR separates the various types of statements into separate fields of the `prog` structure. Once we have the program in this IR representation, we print the resulting ML code directly.

In addition to the two stages, `moogleize` contains a small library module that the generated ML files must be linked against; this library is in `moogle_lib.ml`. In theory, the library is architecture-independent, however there is currently a dependence on the x86 architecture due to the operand forms. Most of the architecture-dependent code is generated by `moogleize` automatically, based on the architecture definition file.

7.1 Generation of classifier functions

When `moogleize -insts` is called, a classification function is generated for each inst_prop defined in the architecture definition. The classification function uses the name of the property; it returns true for instructions which have the property, and false for all other instructions. If a property was applied to an instruction that restricts its operands, then the classification function will only return true when the operands have the proper type.

Caution should be used when multiple instructions map to the same ML constructor, but have different restrictions on their arguments. The instructions are emitted in the same order as they were given in the instructions statements, using a standard ML pattern-match; therefore, if you list the less-restrictive instruction forms before the more restrictive forms, you may get unexpected results with the current implementation. For example, with the following instruction definition:

```
instructions:
   shl   dst, src   ->  SHL (dst, src);
   shlc  dst, $src  ->  SHL (dst, src)  :  pairable;
```

No `SHL` instruction will be considered `pairable`, because in ML the pattern for `shl` covers the pattern for `shlc`. To achieve the desired result, you should list the instructions in the other order:

```
instructions:
   shlc  dst, $src  ->  SHL (dst, src)  :  pairable;
   shl   dst, src   ->  SHL (dst, src);
```

7.2 Generation of transform functions

When `moogleize -insts` is called, transformation functions are also generated for each transform defined in the architecture definition. The transform functions are similar to classification functions, except they return a new instruction instead of a boolean value. Emitted transform functions use the same name they were defined with in the transform statement.

Note that transform functions use an ML match statement, and emit the cases in the same order they were listed in. Therefore it is possible for you to write cases that cannot be reached, by listing a case that has fewer restrictions on its operands before listing a case that is more restrictive; the first case will always be applied, ignoring the second case.

7.3 Generation of rewrite rules

When `moogleize -rules` is called, the functions that build the rewrite rules will be generated. Details regarding how rules are generated are omitted in this version of the specification, but will be added to a future version of the specification. A handful of top-level functions are generated to process *all* rules defined in the named input files.

`process_rules nreg ruledata fallthrough` processes all rules that were generated. `nreg` is a boolean which should be `true` before register allocation, and `false` after register allocation. This controls behaviour of output blocks containing "`new`" — if `true`, then `moogleize` is allowed to generate new register names. `ruledata` is information on the block of instructions being processed by `moogleize`, and is generated using `rule_data_create` (below). `fallthrough` is a function that is called if none of the rules in this file could be applied; it will be called with the `nreg` and `ruledata` arguments.

`rule_data_create buf insts` creates the `ruledata` needed by `process_rules`. `buf` is the `Listbuf.t` to write the output instructions to, and `insts` is the list of instructions to process.

7.4 Generation of inline Kupo code

This will be documented in a subrevision, once inline Kupo code stabilizes.

7.5 Compiling generated files with `ocamlc`

`moogleize` will emit redundant match cases, so when compiling the generated ML files, you should give the `-w Au` flag to `ocamlc`[24]. This is a known issue with the current version of `moogleize` which may be resolved in a later release.

A Revision History

The revision history of this document is listed below, with most recent revisions appearing first.

1.3.1 Minor updates to the language.

1.3 Updated specification to include new 1.3 features, including inline Kupo code.

1.2 OCaml code expressions in clauses were dropped; added unsigned comparison.

1.1 Minor corrections, added some examples.

1.0 Initial revision.

B Undocumented features

The following changes to the compiler are not documented:

- Interface to peephole/rewrite rules has changed.
- Wildcard may either match longest or shortest stream; see command-line option.

C Unimplemented Features

The following changes to moogle/kupo are pending:

- Introduction of support for SpillRegister.
- Rewrite match rules for `*` to use a recursive function.
- Introduction of language support for MIR to assembly conversion.

[24]If you compile with `-warn-error`, you will also want to compile with `-warn-error Au`.

D BNF Notation

The following BNF notation is used in this paper:

e	$::=$	$e_1\ e_2$	Match the concatenation of two expressions
	$\mid$	$e_1 \mid e_2$	Match either of two expressions
	$\mid$	(e)	Match the subexpression e
	$\mid$	"`text`"	Match the indicated verbatim text
	$\mid$	$[\cdots]$	Match a single character listed in the class
	$\mid$	$[\mathit{any}]$	Match any single character
	$\mid$	$[\neg\ \cdots]$	Match a single character which is not listed in the class
	$\mid$	e^*	Match zero or more occurrences of e
	$\mid$	$e^?$	Match zero or one occurrences of e
	$\mid$	e^+	Match one or more occurrences of e

Index