Logical Representation using **if** Expressions*

Brian Monahan

Department of Computer Science
University of Manchester†
Oxford Road, Manchester, UK.

May 11th 1992

†The University of Manchester acknowledges support from the SERC via research grant GR/F 38174.

1 Introduction

Based on Shannon's law[Sha38], Boolean Decision Diagrams have been developed by Bryant for the compact representation of boolean predicates[Bry86]. They are being used by Clarke et al. to represent the finitary relational systems that arise when verifying hardware systems[BCM+90]. In this paper, an approach is described which tries to lessen one of the drawbacks of BDDs – finding a suitable variable ordering upon which to build a canonical representation.

The idea is to avoid using *canonical* forms until they are needed, such as when checking two expressions for equivalence. We do this by relaxing the form in which a propositional expression is expressed during intermediate computations. However, it is still necessary to impose structural invariants to achieve normal and canonical forms – the approach simply allows a choice of *when* the ordering invariants are imposed.

The starting point is to take the well-known 'if-then-else' conditional operator (denoted by **if**):

$$\mathbf{if}(\mathit{Condition}, \mathit{Then}, \mathit{Else})$$

and to develop its algebra to give representations of boolean functions. For the implementation, we intend to adopt a DAG representation of **if** terms and other boolean terms so that potential sharing is exploited as far as possible, using standard implementation techniques (cf. [BRB90]). In the next few sections, we discuss BDDs and their variable ordering problem and also the intuition behind our approach to lessening its influence upon predicate representations.

1.1 Binary Decision Diagrams

In essence, BDDs and their variants[MB88] represent boolean propositional predicates upon free variables. A BDD is a kind of decision graph with shared sub-expressions, having a guaranteed ordering of variables throughout the resulting graph. More precisely, it is a directed acyclic digraph (i.e. each node has at most 2 descendents), with at most two terminal nodes (i.e. true and false respectively). Internal arcs issuing from internal nodes corresponds to choosing a value for some propositional variable. Depending upon the (boolean) choice made for the variable, a descendent node is selected and so on, until a terminal node is reached, representing the value of the predicate for the specified assignment.

1.2 The Problem

To construct a BDD representation for a predicate, a variable ordering is needed to ensure that the representation is canonical. Typically, this ordering is determined prior to the building of a representing BDD. Different orderings would produce differently shaped representative BDDs which may involve different numbers of internal nodes.

We believe that this forces a premature choice of variable ordering precisely when least is known about the predicate's semantics. Moreover, it is upon this semantics that a reasonable choice of variable ordering should be based, unless heuristic techniques are employed. But this is immediately a double bind, since the associated analysis will itself need a semantical representation of the predicate e.g. a BDD representation or similar. From this we see that an inexpensive choice of variable orderings is only possible when relatively little analysis is done beforehand. As such, any choice of variable ordering can only be influenced by attributes that are directly and cheaply extracted from the syntactic presentation. The upshot of this is that it is hard to find variable orderings that give reasonable representations of arbitrarily given predicates.

In fact the situation is worse than this — there may be *no* reasonable BDD representation at all for particular predicates. For example, Bryant has shown that all BDD representations of the n-bit integer multiplication function are always of size exponential in n, no matter which variable ordering is used [Bry91]. The result is robust in the sense that minor tinkering with the definition of BDDs is unlikely to produce much of an improvement, if any. Furthermore, an argument due to Shannon suggests that, since there are 2^{2^n} boolean function of arity n, only a vanishingly small percentage of the boolean functions of given arity could have a reasonably sized canonical form[Sha49]. Such results suggests to us that the problem lies not just with the use of BDDs, but with the management and usage of canonical forms in general.

Further evidence that canonical forms are hard to manage comes from the fact that the combination or processing of several BDD structures usually requires that all such structures have compatible variable orderings. To ensure that this is the case may then involve the rearrangement or at least the re-indexing of one or more structures, which can be an expensive process in the worst case. In any system giving verification support for non-trivial applications, semantic representations (such as BDDs) are very likely to come from a variety of independent sources and as such are therefore likely to use incompatible variable orderings. In short, the guaranteed compatibility of variable orderings is unlikely to be achievable in practice.

Our approach involves the investigation of a more general class of representation which has the following objectives and properties.

- To obtain a semantic representation of a given predicate without concern for structural invariants such as variable orderings and so on. In particular, it should be possible to calculate this representation without incurring excessive overheads due consistency or optimal selection requirements.
- To in principle combine arbitrary structures without consistency requirements on the constituents. Although it may be more efficient to exploit such requirements during processing, it should not be strictly necessary to do so.

  In short, input structures need not be in canonical form for operations to be generally applicable.
- To permit a degree of latitude in the representation of structures so that implementations are free to make expedient choices on-the-fly, during computation. This permits implementations to choose the representations of intermediate values appropriate to the context and the natural flow of computation.
- To ensure that BDD structures are included within the representation.

This last property says that a generalisation of BDDs is needed. Fortunately, a suitable candidate is easily found, in the form of the **if** expression. Given a BDD in the form of an acyclic graph, it easy to see how to construct, by bottom-up traversal, an equivalent **if** expression representing the predicate denoted by the BDD.

1.3 Anticipated Benefits and Drawbacks of the Approach

We anticipate the following benefits from adopting an approach based on the above requirements.

- Flexibility of structural representation during processing. This permits implementations to avoid putting the results into unnecessarily restrictive normal forms for later processing.
- On-the-fly selection of equivalent representatives is permitted, so allowing implementations to make choices according to internal criteria.
- Use of particular normal forms (see below) is not forced and can therefore be chosen by the implementation.
- Representatives of input expressions can be found that are of similar complexity to the given expression. Although such forms are likely to be far from being in any normal form, basic logical operations can be performed on them nonetheless.

The main disadvantages we anticipate are:

- computational effort is needed to place formulae into canonical forms when required, such as when testing for equality.
- non-optimal sharing of representations of equivalent sub-expression due to the use of a representation that is potentially non-canonical which, in the worst case, can lead to much larger structures than for conventional BDDs.

We anticipate that the first disadvantage is acceptable because tests for equality are relatively rare in comparison to other logical operations. The second disadvantage is potentially far more serious. We suspect that the flexibility gained by not fixing an ordering early on in the lifetime of a representation will have to be paid for by introducing *ordering constraints* later on dynamically. Further investigation of these and other factors therefore need to be undertaken.

We briefly remark on the manner in which results will be presented in this paper. Our purpose is to state a number of conceptual relationships, within the propositional and relational calculus, that we expect will have a bearing on implementation considerations. Individually, many of the results are rather straightforward and (very probably) known. What is of interest here is the combination and collection of results as a whole, to provide motivation for the implementation. Accordingly, we do not formally present proofs, unless the result seems unusual or otherwise interesting. In such cases, an informal indication of how a proof could be provided is given. In other cases, most of the results yield to a combination of equational deduction, case analysis and structural induction over syntactic structures such as expressions.

2 The Algebra of Conditionals

The humble conditional form, **if**, is well-known within computer science, it being the basic logical building block used within several major programming languages (cf. Pascal, Ada, Lisp, ...).

$$\mathbf{if}(A,B,C) \quad \underline{\triangle} \quad (A \wedge B) \vee (\neg A \wedge C)$$

However, when used in these programming languages, it is essential that the conditional is *non-strict* to permit the dynamic selection of the alternative to be evaluate. In the work presented here, **if** is treated more as an active switching unit where the alternatives are always computed, regardless of the value taken by the control expression. In other words, the control input acts to *select* which of the inputs to pass to the output. As such, this version of **if** can be viewed as a strict, total function upon booleans.

We use the following terminology for parts of a conditional expression:

$$\mathbf{if}(Cntrl, Pos, Neg)$$

where:

Cntrl
: the *control expression*, whose value governs which arm is selected for output.

Pos
: the *positive arm*, selected when *Cntrl* takes the value tt.

Neg
: the *negative arm*, selected when *Cntrl* takes the value ff.

Occasionally we use the term *logical arm* to refer to either of these 'arms'.

2.1 Alternative Forms of Conditional and their Duals

We should note that there are many equivalent ways in which the **if** function could have been defined, such as by the following:

$$\mathbf{if}(A,B,C) \equiv (A \Rightarrow B) \wedge (\neg A \Rightarrow C) \qquad \boxed{\text{cond-impl-def}}$$

The equivalence to the original definition may easily be demonstrated in several ways, such as by case analysis or by algebraic manipulation.

Since **if** is a propositional operator, we may consider what function its formal 'dual', **dual–if**, corresponds to. This satisfies:

$$\neg\,\mathbf{if}(A,B,C) \equiv \mathbf{dual\text{–}if}(\neg A, \neg B, \neg C)$$

By applying negation to the arguments of **dual–if**, we see that:

$$\begin{aligned} \mathbf{dual\text{–}if}(A,B,C) &\equiv \neg\,\mathbf{if}(\neg A, \neg B, \neg C) \\ &\equiv \mathbf{if}(\neg A, B, C) \\ &\equiv \mathbf{if}(A, C, B) \end{aligned}$$

So, in this sense, we can say that **if** is *self-dual*.

In a similar vein, other, less obvious, forms of 'conditional' may also be defined, such as by :

$$\mathbf{dual\text{–}if}_1(A,B,C) \quad \underline{\triangle} \quad (B \Rightarrow A) \wedge (C \Rightarrow \neg A)$$

It turns out that this, too, is expressable in terms of **if** :

$$\begin{aligned}\textbf{dual–if}_1(A,B,C) &\equiv \textbf{if}(\neg A, \neg B, \neg C)\\ &\equiv \textbf{if}(A, \neg C, \neg B)\end{aligned}$$

By extending this analysis, it can be seen that there are essentially eight different conditional functions, each one corresponding to an entry in the following table of expressions:

$\textbf{if}(A,B,C)$	$\textbf{if}(A,B,\neg C)$	$\textbf{if}(A,\neg B,C)$	$\textbf{if}(A,\neg B,\neg C)$
$\textbf{if}(\neg A,B,C)$	$\textbf{if}(\neg A,B,\neg C)$	$\textbf{if}(\neg A,\neg B,C)$	$\textbf{if}(\neg A,\neg B,\neg C)$

Moreover, it may be seen that each form can express each of the others. To see this, observe that it is sufficient for the form to express the original **if** function. However this is easily shown, by negating appropriate arguments and exploiting the fact that negation is an involution.

From this, we conclude that there is no essential advantage to using any particular form over conditional over any other and so continue using the original **if** operator with impunity.

2.2 Preliminaries and Notational Conventions

In this section, we shall make a number of preliminary remarks concerning the technical framework used.

2.2.1 Boolean Constants

The collection of boolean values is denoted by $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$, where tt stands for 'true', ff stands for 'false', as usual.

2.2.2 Signed literals

We shall be using *signed literals*. Let *Sym* stand for an unbounded supply of objects whose internal structure is irrelevent, apart from being able to distinguish one from another. Such objects will be called *symbols*. A *signed literal* is a pair consisting of a *sign* (one of '+' and '-') and a symbol. Particular literals can be written in the obvious way as +*s* or -*s*, for symbol *s*. A literal of the form +*s* is said to be a *positive literal* and literals of the form -*s* are said to be *negative*.

For convenience, we may omit the plus sign when writing positive literals, so *s* stands for the literal +*s*. However, by the same token, negative literals will *always* be written as -*s*. Note that sign combinations such as '- +*s*', '+ +*s*' and '- -*s*' are not well-formed.

Formally, we also introduce the following sets:

$$\begin{aligned} Lit_+ &= \{+s \mid s \in Sym\}\\ Lit_- &= \{-s \mid s \in Sym\}\\ Lit &= Lit_+ \cup Lit_- \end{aligned}$$

where Lit_+ is the set of positive literals, Lit_- is the set of negative literals, and Lit is the set of all literals.

It will be useful to introduce a couple of operators on literals, defined by the following equations, where $s \in Sym$:

$$\begin{array}{rcl} \overline{(\_)} & : & Lit \rightarrow Lit \\ \\ \overline{+s} & = & \text{-}s \\ \overline{\text{-}s} & = & +s \end{array} \qquad \begin{array}{rcl} pos(\_) & : & Lit \rightarrow Lit \\ \\ pos(+s) & = & +s \\ pos(\text{-}s) & = & +s \end{array}$$

Clearly, the $\overline{(\_)}$ operator is an *involution*, whereas the $pos(\_)$ operator is *idempotent*. Let $p \in Lit$:

$$\overline{\overline{p}} = p \qquad \boxed{\text{involution}}$$

$$pos(pos(p)) = pos(p) \qquad \boxed{\text{idempotence}}$$

The sign switching operator $\overline{(\_)}$ can be used to implement the effect of negation upon literals (e.g. $\neg p = \overline{p}$).

In addition, it is useful to introduce some set operators which use the operators above. Let $S \subseteq Lit$:

$$\begin{array}{rcl} \overline{S} & \triangleq & \{\overline{x} \mid x \in S\} \\ pos(S) & \triangleq & \{pos(x) \mid x \in S\} \\ closure(S) & \triangleq & \overline{S} \cup S \\ & \equiv & \overline{pos(S)} \cup pos(S) \end{array}$$

2.2.3 Variables and Meta Variables

For the sake of clarity in this paper, it is convenient to have several classes of lexically distinguishable meta variables. In general, we shall let lower case letters range over literals and upper case letters range over boolean expressions and propositional functions (e.g. predicates).

In addition, we may use the following conventions:

s to range over Sym, the set of symbols

$v, v_1, v_2, \cdots$ denote positive literals (i.e. elements of Lit_+)

2.2.4 Free variables and literals

As we use signed literals, we need to clarify what exactly constitutes a free variable and a free literal, especially since we shall be using constructs that bind variables, thereby introducing restricted scopes.

Due to signed literals, it is possible to discriminate between positive and negative occurrences. In our view, a variable is said to freely occur within an expression whenever it freely occurs with either sign. That is, whether or not a variable occurs freely should be independent of the sign of any of its occurrences.

To say this more formally, we shall introduce a number of 'operators' which pin these notions down further. This usually involves defining the structure of expressions explicitly and then introducing these operators by induction. In our presentation, we shall take a more generic approach which avoids introducing unnecessary details.

For this purpose, assume that there is a general structured set of expressions, called *Expr*. This will contain representations of all the term structures that we need to use. Next, we introduce names and types of the operators needed:

$$
\begin{aligned}
PL &: Expr \to \mathbb{P}Lit \text{ — the set of free positive literals}\\
NL &: Expr \to \mathbb{P}Lit \text{ — the set of free negative literals}\\
FL &: Expr \to \mathbb{P}Lit \text{ — the set of free signed literals}\\
UL &: Expr \to \mathbb{P}Lit \text{ — the set of free literals (disregarding sign)}\\
FV &: Expr \to \mathbb{P}Lit \text{ — the set of free variables}
\end{aligned}
$$

Let the set of all these operators be $LitOP = \{PL, NL, FL, UL, FV\}$ and let $BLitOP = \{PL, NL\}$, the set of basic literal operators. Our definitions are generic since the laws we give often apply to more than one of the operators. In addition, the operators can be defined generally over classes of expressions, the structure of which need not be made fully explicit, except to say that the generic equations cover all possibilities.

The ground cases for the *BLitOP* operators are:

$$
\begin{aligned}
PL(+s) &\equiv \{+s\} && \boxed{\text{PL+lit}}\\
PL(-s) &\equiv \{\} && \boxed{\text{PL-lit}}\\
NL(+s) &\equiv \{\} && \boxed{\text{NL+lit}}\\
NL(-s) &\equiv \{-s\} && \boxed{\text{NL-lit}}
\end{aligned}
$$

The remainder of the definitions for *BLitOP* operators are common to both - and may be given genericlly in terms of the following laws, where $LL \in BLitOP$, F is a unary operator symbol, Op is a binary operator symbol and Q is a variable quantification symbol:

$$
\begin{aligned}
LL(C) &\equiv \{\} \qquad (\text{where } C \in \mathbb{B}) && \boxed{\text{LL-const}}\\
LL(F(A)) &\equiv LL(A) && \boxed{\text{LL-unary}}\\
LL(A\ Op\ B) &\equiv LL(A) \cup LL(B) && \boxed{\text{LL-binary}}\\
LL(\mathbf{if}(A,B,C)) &\equiv LL(A) \cup LL(B) \cup LL(C) && \boxed{\text{LL-cond}}\\
LL(Q\ v \bullet E) &\equiv LL(E) - \{v, \overline{v}\} && \boxed{\text{LL-quant}}
\end{aligned}
$$

The other operators in *LitOP* can be defined as follows:

$$
\begin{aligned}
FL(E) &\triangleq PL(E) \cup NL(E) && \boxed{\text{FL-defn}}\\
UL(E) &\triangleq \{pos(p) \mid p \in FL(E)\} && \boxed{\text{UL-defn}}\\
FV(E) &\triangleq FL(E) \cup \{\overline{p} \mid p \in FL(E)\} && \boxed{\text{FV-defn}}
\end{aligned}
$$

From the above, it is easy to read off a number of straightforward corollaries, such as:

$$FL(x) \equiv \{x\} \qquad \boxed{\text{FL-lit}}$$
$$UL(x) \equiv \{pos(x)\} \qquad \boxed{\text{UL-lit}}$$
$$FV(x) \equiv \{x, \overline{x}\} \qquad \boxed{\text{FV-lit}}$$

$$LOp(\mathbf{if}(A,B,C)) \equiv LOp(A) \cup LOp(B) \cup LOp(C) \qquad \boxed{\text{LOp-cond}}$$
$$(\text{where } LOp \in LitOP)$$

where the proof of LOp-cond needs the general law:

$$\{f(x) \mid x \in (S_1 \cup S_2)\} \equiv \{f(x) \mid x \in S_1\} \cup \{f(x) \mid x \in S_2\} \qquad \boxed{\text{apply-set-distr}}$$

In addition, other general properties are equally evident:

$$p \in FV(E) \Leftrightarrow pos(p) \in UL(E) \qquad \boxed{\text{FV-UL}}$$
$$p \in FV(E) \Leftrightarrow \overline{p} \in FV(E) \qquad \boxed{\text{neg-FV}}$$
$$x \in FV(y) \Leftrightarrow y \in FV(x) \qquad \boxed{\text{FV-lit sym}}$$

The law FV-UL is useful as it suggests a more space-efficient way of computing collections of free variables, instead of using the definition of FV directly.

Finally, we introduce a useful extension of the above operators so that they may be applied to sets of expressions and so on. For $LOp \in LitOp$, and $S \subseteq Expr$, put:

$$LOp(S) \triangleq \bigcup \{LOp(e) \mid e \in S\} \qquad \boxed{\text{LOp-set-defn}}$$

2.2.5 Equivalence

Various notions of equality are used, each generally related to the other. As we shall be dealing with several object languages, it will be useful to state these properties generically.

Let *Lang* be an object language, considered as a collection of structured phrases. Without loss of generality, we can assume that a denotational semantics is specified by function $M{:}Lang \to Den$, for some appropriate space of semantic denotations, *Den*. This may always be done, by perhaps rearranging and currying of arguments.

To simplify the discussion, we further assume that our object language *Lang* carries an *internal logic*[1], based upon classical propositional calculus. This means that we have a semantic notion of *validity*[2], $\models P$ which satisfies the following (somewhat meta circular) condition:

[1] A more general discussion would involve entering the field of *generic logical frameworks*.

[2] The notion of validity is usually defined relative to a semantic notion of *satisfaction* between interpretations and formulae. However, the definition used here is sufficient for our purpose in this paper.

$$(\models P) \;\Leftrightarrow\; (M(P) = M(\mathsf{tt}))$$

We now describe the relationships between the notions of equivalence we use:

Identity : $a = b$

This means that the values denoted by a and b are absolutely identical. Moreover, this operator is genericlly applicable between arbitrary kinds of object. In particular, when a and b denote syntactical phrases, $a = b$ means that a and b are syntactically indistinguishable.

Semantic equivalence : $P_1 \equiv P_2$

This means that the linguistic phrases $P_1, P_2 \in \mathit{Lang}$ have identical meaning or semantics. More formally, this means that:

$$(P_1 \equiv P_2) \;\Leftrightarrow\; (M(P_1) = M(P_2))$$

Internal logical equivalence : $P_1 \;\Leftrightarrow\; P_2$

This is a syntactic operator with *Lang* that represents logical equality within Lang itself. As such, phrases of the form $P_1 \;\Leftrightarrow\; P_2$ are members of *Lang* (i.e. $(P_1 \;\Leftrightarrow\; P_2) \in \mathit{Lang}$). From a semantic point of view, the following statement relates the internal and external equivalences:

$$(\models (P_1 \;\Leftrightarrow\; P_2)) \;\Leftrightarrow\; (P_1 \equiv P_2)$$

2.2.6 General Meta Notation

It is often useful to introduce and use functions and relations that operate on the symbolic and syntactic data, such as the 'literal' operators described above. Generally speaking, these things are said to operate at the *meta level* to distinguish them from the syntactic operators used to form phrases at the *object level* : meta level operators are said to form part of the *meta language*, whereas object level operators are said to form part of the *object language*.

It is clearly reasonable to avoid unnecessary confusion of these object and meta languages and we shall attempt to do this by introducing various syntactical conventions. For example, we have already done so for the *signed literals* and the literal operators such as $FV(\_)$ and $\overline{(\_)}$ above. Other special syntax will be introduced when needed, such as the substitution notation: $A\,[v{:=}\,B]$ in section 6.

However, it is not always desireable to introduce special syntax, especially if it would introduce too much overloading of meaning on the resulting notation. Therefore, it is also necessary to have a 'default' notation for meta level operators. We may do this by writing specific meta level terms in standard prefix notation, such as:

$$\mathsf{op}[A_1, \cdots, A_k]$$

or by using subscript notation:

$$\mathsf{op}_{A_1, \cdots, A_k}$$

We may use the above notations interchangeably whenever ambiguity does not result by doing so.

2.3 Equational rules and properties

We now turn to describing many of the equational properties of the **if** algebra. Before proceeding in detail, we note that it is not our intention to develop an equational theory for reasons other than appropriateness to the subject matter. In particular, there is no specific intention to map our equations into a form of term rewriting system. As will be seen, the form of some of the equations makes that objective difficult to achieve.

2.3.1 Propositional completeness

We can show how **if** can be used to code up arbitrary propositional functions, according to the following table of equivalence rules:

$$
\begin{array}{rcll}
(A \wedge B) & \equiv & \mathbf{if}(A, B, \mathsf{ff}) & \boxed{\text{and-cond}} \\
(A \vee B) & \equiv & \mathbf{if}(A, \mathsf{tt}, B) & \boxed{\text{or-cond}} \\
(A \Rightarrow B) & \equiv & \mathbf{if}(A, B, \mathsf{tt}) & \boxed{\text{imp-cond}} \\
(A \oplus B) & \equiv & \mathbf{if}(A, \neg B, B) & \boxed{\text{xor-cond}} \\
(A \Leftrightarrow B) & \equiv & \mathbf{if}(A, B, \neg B) & \boxed{\text{iff-cond}} \\
\neg A & \equiv & \mathbf{if}(A, \mathsf{ff}, \mathsf{tt}) & \boxed{\text{neg-cond}}
\end{array}
$$

The above rules show that **if** is complete for expressing propositional systems. However, the problem with applying the above rules naïvely is that the resulting **if** expressions will be severely skewed. In all the above rules, the first argument of each operation is used in the control position of the **if** expression. In practice, a richer set of transformation may be used which is often more direct and balanced. Even then, the above rules are still needed in the case that the sub-formulae A, B are atomic (i.e. literals or constants).

If it is necessary, the following rules could be used to map literals into a **if** expression form such that they only occur positively in the control position:

$$
\begin{array}{rcll}
+s & \equiv & \mathbf{if}(+s, \mathsf{tt}, \mathsf{ff}) & \boxed{\text{poslit-cond}} \\
\text{-}s & \equiv & \mathbf{if}(+s, \mathsf{ff}, \mathsf{tt}). & \boxed{\text{neglit-cond}}
\end{array}
$$

2.3.2 Basic Projection and Simplification rules

$$
\begin{array}{rcll}
\mathbf{if}(A, B, B) & \equiv & B & \boxed{\text{Hobsons choice}} \\
\mathbf{if}(\mathsf{tt}, A, B) & \equiv & A & \boxed{\text{cond-true}} \\
\mathbf{if}(\mathsf{ff}, A, B) & \equiv & B & \boxed{\text{cond-false}} \\
\mathbf{if}(A, \mathsf{tt}, \mathsf{ff}) & \equiv & A & \boxed{\text{identity}} \\
\mathbf{if}(A, A, \neg A) & \equiv & \mathsf{tt} & \boxed{\text{tt cond}} \\
\mathbf{if}(A, \neg A, A) & \equiv & \mathsf{ff} & \boxed{\text{ff cond}}
\end{array}
$$

The identity rule is a converse to the more specific 'poslit-cond' rule given above. Clearly, the identity rule is unlikely to be used to undo the effect of a prior application of 'poslit-cond'; its main use is to eliminate redundant **if** expressions.

The following simple cancellation rules are straightforward consequences of the above rules.

$$\mathbf{if}(A, \mathbf{if}(A, B, C), X) \equiv \mathbf{if}(A, B, X) \qquad \boxed{\text{nest-L}}$$
$$\mathbf{if}(A, X, \mathbf{if}(A, B, C)) \equiv \mathbf{if}(A, X, C) \qquad \boxed{\text{nest-R}}$$
$$\mathbf{if}(A, \mathbf{if}(A, B, C), \mathbf{if}(A, D, E)) \equiv \mathbf{if}(A, B, E) \qquad \boxed{\text{nest-LR}}$$

It turns out that the above 'nesting' rules are also consequences of the more general 'meta-eval' law (see Section 2.3.7).

2.3.3 Reorganisation rules

The following rules can be used to reorganise compound **if** expressions, by redistributing their immediate subcomponents.

$$\mathbf{if}(\mathbf{if}(A, B, C), D, E) \equiv \qquad \boxed{\text{push}}$$
$$\mathbf{if}(A, \mathbf{if}(B, D, E), \mathbf{if}(C, D, E))$$

$$\mathbf{if}(A, \mathbf{if}(B, C, D), \mathbf{if}(B, E, F)) \equiv \qquad \boxed{\text{swop}}$$
$$\mathbf{if}(B, \mathbf{if}(A, C, E), \mathbf{if}(A, D, F)).$$

The push rule moves **if** expressions out of the control position and redistributes them over the logical arms. The swop rule exchanges control expressions assuming that the major **if** expression is 'balanced' - i.e. each of the logical arms are **if** expressions containing the same control expression.

$$\mathbf{if}(A, X, \mathbf{if}(B, C, D)) \equiv \mathbf{if}(B, \mathbf{if}(A, X, C), \mathbf{if}(A, X, D)) \qquad \boxed{\text{rotate-L}}$$
$$\mathbf{if}(A, \mathbf{if}(B, C, D), X) \equiv \mathbf{if}(B, \mathbf{if}(A, C, X), \mathbf{if}(A, D, X)). \qquad \boxed{\text{rotate-R}}$$

$$\mathbf{if}(A, \mathbf{if}(B, X, C), \mathbf{if}(B, X, D)) \equiv \mathbf{if}(B, X, \mathbf{if}(A, C, D)) \qquad \boxed{\text{cancel-L}}$$
$$\mathbf{if}(A, \mathbf{if}(B, C, X), \mathbf{if}(B, D, X)) \equiv \mathbf{if}(B, \mathbf{if}(A, C, D), X). \qquad \boxed{\text{cancel-R}}$$

Note that the rotate-L and rotate-R rules are the converses of the cancel-L and cancel-R rules. So, if applied naïvely, these rules would lead to probable non-termination. However, in practice, the reasons for applying one or other of these rules would differ in each case and will not clash. These issues are considered later within Section **??** which deals with implementation matters.

2.3.4 Control Combination rules

$$
\begin{aligned}
\mathbf{if}(\neg A, B, C) &\equiv \mathbf{if}(A, C, B) && \boxed{\text{cntrl-neg}}\\
\mathbf{if}(A \wedge B, C, D) &\equiv \mathbf{if}(A, \mathbf{if}(B, C, D), D) && \boxed{\text{cntrl-and}}\\
\mathbf{if}(A \vee B, C, D) &\equiv \mathbf{if}(A, C, \mathbf{if}(B, C, D)) && \boxed{\text{cntrl-or}}\\
\mathbf{if}(A \;\Rightarrow\; B, C, D) &\equiv \mathbf{if}(A, \mathbf{if}(B, C, D), C) && \boxed{\text{cntrl-imp}}\\
\mathbf{if}(A \oplus B, C, D) &\equiv \mathbf{if}(A, \mathbf{if}(B, D, C), \mathbf{if}(B, C, D)) && \boxed{\text{cntrl-xor}}\\
\mathbf{if}(A \;\Leftrightarrow\; B, C, D) &\equiv \mathbf{if}(A, \mathbf{if}(B, C, D), \mathbf{if}(B, D, C)). && \boxed{\text{cntrl-iff}}
\end{aligned}
$$

2.3.5 External combination rules

We can also give a number of rules for combining **if** expressions via unary operations (denoted by $F(\_)$) and binary operations (denoted by $(\_\ Op\ \_)$). The interesting point about the laws below is that they are *generic* in the interpretation of the operators. This arises because **if** just ‘multiplexes’ its arguments to the output (c.f. the evaluation rules ‘cond-true’ and ‘cond-false’. The laws below could be proven by case analysis on the value of the control expression.

$$
\begin{aligned}
F(\mathbf{if}(A, B, C)) &\equiv \mathbf{if}(A, F(B), F(C)) && \boxed{\text{distr 1}}\\
\mathbf{if}(A, B, C)\ Op\ \ X &\equiv \mathbf{if}(A, (B\ Op\ \ X), (C\ Op\ \ X)) && \boxed{\text{distr 2 R}}\\
X\ Op\ \ \mathbf{if}(A, B, C) &\equiv \mathbf{if}(A, (X\ Op\ \ B), (X\ Op\ \ C)). && \boxed{\text{distr 2 L}}
\end{aligned}
$$

The following rules are useful when both arguments are **if** expressions:

$$
\mathbf{if}(A, B, C)\ Op\ \ \mathbf{if}(A, E, F) \equiv \mathbf{if}(A, (B\ Op\ \ E), (C\ Op\ \ F)) \qquad \boxed{\text{comb 2}}
$$

$$
\begin{aligned}
&\mathbf{if}(A, B, C)\ Op\ \ \mathbf{if}(D, E, F) \;\equiv && \boxed{\text{comb 2 LR}}\\
&\qquad \mathbf{if}(A, \mathbf{if}(D, (B\ Op\ \ E), (B\ Op\ \ F)),\\
&\qquad\qquad \mathbf{if}(D, (C\ Op\ \ E), (C\ Op\ \ F)))
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{if}(A, B, C)\ Op\ \ \mathbf{if}(D, E, F) \;\equiv && \boxed{\text{comb 2 RL}}\\
&\qquad \mathbf{if}(D, \mathbf{if}(A, (B\ Op\ \ E), (C\ Op\ \ E)),\\
&\qquad\qquad \mathbf{if}(A, (B\ Op\ \ F), (C\ Op\ \ F))).
\end{aligned}
$$

2.3.6 Multiple combination

The following rules show how a multiple combination formed using *associative-commutative* operators may be evaluated as a group. An *associative-commutative* binary operator, *aop*, is one satisfying:

$$(A\ aop\ B)\ aop\ C \equiv A\ aop\ (B\ aop\ C) \qquad \boxed{\text{associativity}}$$
$$A\ aop\ B \equiv B\ aop\ A \qquad \boxed{\text{commutativity}}$$

We shall proceed by defining a new operator, $aop(E_1, \cdots, E_k)$, for associative operator *aop* and expressions $E_1, \cdots, E_k$ (where $k > 1$). For our purposes, we need only consider the logical associative operators: $AOp = \wedge, \vee, \Leftrightarrow, \oplus\}$, and can assume that $aop \in AOp$.

The following provides some general laws:

$$aop(E_1, E_2) \equiv (E_1\ aop\ E_2) \qquad \boxed{\text{mcomb two}}$$
$$(E_1\ aop\ E_2)\ aop\ E_3 \equiv aop(E_1, E_2, E_3) \qquad \boxed{\text{mcomb intro L}}$$
$$E_1\ aop\ (E_2\ aop\ E_3) \equiv aop(E_1, E_2, E_3) \qquad \boxed{\text{mcomb intro R}}$$
$$A\ aop\ aop(E_1, \cdots, E_k) \equiv aop(A, E_1, \cdots, E_k) \qquad \boxed{\text{mcomb nest L}}$$
$$aop(E_1, \cdots, E_k)\ aop\ A \equiv aop(A, E_1, \cdots, E_k) \qquad \boxed{\text{mcomb nest R}}$$
$$aop((A\ aop\ B), E_1, \cdots, E_k) \equiv aop(A, B, E_1, \cdots, E_k) \qquad \boxed{\text{mcomb unpack 1}}$$
$$aop(aop(A_1, \cdots, A_m), B_1, \cdots, B_n) \equiv \qquad \boxed{\text{mcomb unpack many}}$$
$$aop(A_1, \cdots, A_m, B_1, \cdots, B_n)$$

$$aop(p_1, p_2, \cdots, p_n) \equiv p_1\ aop\ aop(p_2, \cdots, p_n) \qquad \boxed{\text{mcomb literal}}$$
$$aop(\mathbf{if}(A, B_1, C_1), \cdots, \mathbf{if}(A, B_k, C_k), E_1, \cdots, E_m) \equiv \qquad \boxed{\text{mcomb cond}}$$
$$\mathbf{if}(A, aop(B_1, \cdots, B_k, E_1, \cdots, E_m),$$
$$aop(C_1, \cdots, C_k, E_1, \cdots, E_m))$$
$$(\text{where } k \geq 1 \wedge m \geq 0)$$

The following rule is very useful in providing *licence* to apply multiple combination rules arbitrarily, where for permutation $\pi \in S_k$:

$$aop(E_1, \cdots, E_k) \equiv aop(E_{\pi(1)}, \cdots, E_{\pi(k)}) \qquad \boxed{\text{mcomb permute}}$$

Note that an implementation need not use permutations directly to apply such rules – it needs only to act as though it did. For example, this rule would justify 'sorting' the argument expressions under some appropriate criteria.

The remaining rules for multiple combination are concerned with specific properties inherited from particular binary operators. We introduce the following division of operators:

$AbsAOp = \{\wedge, \vee\}$ — absorptive operators

$BalAOp = \{\Leftrightarrow, \oplus\}$ — balanced operators

For the *absorptive operators*, $aop \in AbsAOp$, we have the following absorption rule:

$$aop(A, \cdots, A, E_1, \cdots, E_k) \equiv aop(A, E_1, \cdots, E_k) \qquad \boxed{\text{mcomb absorb}}$$

We note that operators $\wedge, \vee$ possess both zero and identity elements and can define the meta operators $\mathsf{zero\_}, \mathsf{unit\_}$ as follows:

$$\mathsf{zero}_{\wedge} = \mathsf{ff}$$
$$\mathsf{zero}_{\vee} = \mathsf{tt}$$

and, for $aop \in AbsAOp$:

$$\mathsf{unit}_{aop} = \neg\, \mathsf{zero}_{aop}$$

Using the above notation, we can easily sum up the zero and identity laws for operators $aop \in AbsAOp$:

$$A\ aop\ \mathsf{unit}_{aop} \equiv A \equiv \mathsf{unit}_{aop}\ aop\ A \qquad \boxed{\text{identity}}$$

$$A\ aop\ \mathsf{zero}_{aop} \equiv \mathsf{zero}_{aop} \equiv \mathsf{zero}_{aop}\ aop\ A \qquad \boxed{\text{zero}}$$

$$A\ aop\ \neg A \equiv \mathsf{zero}_{aop} \equiv \neg A\ aop\ A \qquad \boxed{\text{cancel}}$$

So, applying the above to multiple combination with $aop \in AbsAOp$:

$$aop(\mathsf{unit}_{aop}, E_1, \cdots, E_k) \equiv aop(E_1, \cdots, E_k) \qquad \boxed{\text{mcomb unit}}$$
$$aop(\mathsf{zero}_{aop}, E_1, \cdots, E_k) \equiv \mathsf{zero}_{aop} \qquad \boxed{\text{mcomb zero}}$$
$$aop(A, \neg A, E_1, \cdots, E_k) \equiv \mathsf{zero}_{aop} \qquad \boxed{\text{mcomb cancel}}$$

Concerning the balanced operators, $aop \in BalAOp$, we also have the following, for $k \geq 2$:

$$\Leftrightarrow(A, A, E_1, \cdots, E_k) \equiv \Leftrightarrow(E_1, \cdots, E_k) \qquad \boxed{\text{mcomb iff dup}}$$
$$\Leftrightarrow(\mathsf{tt}, E_1, \cdots, E_k) \equiv \Leftrightarrow(E_1, \cdots, E_k) \qquad \boxed{\text{mcomb iff tt}}$$
$$\Leftrightarrow(\mathsf{ff}, E_1, \cdots, E_k) \equiv \oplus(E_1, \cdots, E_k) \qquad \boxed{\text{mcomb iff ff}}$$

Now, from Boolean Algebra, we can see that the following properties of $\oplus$ and $\Leftrightarrow$ hold:

$$(A \Leftrightarrow B) \equiv (\neg A \Leftrightarrow \neg B) \qquad \boxed{\text{iff duo neg}}$$
$$\neg(A \Leftrightarrow B) \equiv (A \oplus B) \qquad \boxed{\text{iff xor}}$$
$$\neg(A \Leftrightarrow B) \equiv (\neg A \Leftrightarrow B) \qquad \boxed{\text{iff neg L}}$$
$$\neg(A \Leftrightarrow B) \equiv (A \Leftrightarrow \neg B) \qquad \boxed{\text{iff neg R}}$$

The following somewhat surprising law is a straightforward consequence of the above:

$$A \Leftrightarrow (B \Leftrightarrow C) \equiv A \oplus (B \oplus C) \qquad \boxed{\text{iff xor 3}}$$

Using this and a straightforward induction, we can then show that, for $k \geq 1$:

$$\oplus(E_1, \cdots, E_{2k+1}) \equiv \Leftrightarrow(E_1, \cdots, E_{2k+1}) \qquad \boxed{\text{mcomb xor odd}}$$
$$\oplus(E_1, \cdots, E_{2k}) \equiv \neg\Leftrightarrow(E_1, \cdots, E_{2k}) \qquad \boxed{\text{mcomb xor even}}$$

In addition, we note the following, for $k \geq 2$:

$$\oplus(A, A, E_1, \cdots, E_k) \equiv \Leftrightarrow(E_1, \cdots, E_k) \qquad \boxed{\text{mcomb xor dup}}$$
$$\oplus(\mathsf{tt}, E_1, \cdots, E_k) \equiv \Leftrightarrow(E_1, \cdots, E_k) \qquad \boxed{\text{mcomb xor tt}}$$
$$\oplus(\mathsf{ff}, E_1, \cdots, E_k) \equiv \oplus(E_1, \cdots, E_k) \qquad \boxed{\text{mcomb xor ff}}$$

Finally, using the above we can show that the following soundness property holds:

Theorem 2.1 (multiple combination theorem)
For $aop \in AOp$, we have, for $k \geq 2$:

$$E_1 \; aop \; \cdots \; aop \; E_k \equiv aop(E_1, \cdots, E_k)$$

2.3.7 Shannon's case analysis law and evaluation

The construction of BDDs rests on the following case analysis law, due to Shannon[Sha38], for boolean expressions, E, and a literal p:

$$E \equiv \mathbf{if}(p, (E/p), (E/\overline{p})) \qquad \boxed{\text{Shannon's Law}}$$

where the expression E/p stands for the result of *evaluating* under the interpretation specified by the literal p. Essentially, the $(\_/\_)$ operator is a *meta* operator, working at the level of boolean expressions and propositional functions. Not surprisingly, the evaluation process can be defined by the following inductive rules over the structure of expressions:

$$\mathbf{if}(A, B, C)/p \equiv \mathbf{if}((A/p), (B/p), (C/p)) \qquad \boxed{\text{eval-cond}}$$
$$\mathbf{if}(p, B, C)/p \equiv (B/p) \qquad \boxed{\text{eval-cond-pos}}$$
$$\mathbf{if}(p, B, C)/\overline{p} \equiv (C/\overline{p}) \qquad \boxed{\text{eval-cond-neg}}$$
$$(A \; Op \; B)/p \equiv (A/p) \; Op \; (B/p) \qquad \boxed{\text{eval-bop}}$$
$$(F(E))/p \equiv F(E/p) \qquad \boxed{\text{eval-uop}}$$

$$\begin{aligned} \mathcal{C}/p &\equiv \mathcal{C} \qquad (\text{where } \mathcal{C} \in \mathbb{B}) && \boxed{\text{eval-const}} \\ p/p &\equiv \mathsf{tt} && \boxed{\text{eval-var +}} \\ p/\overline{p} &\equiv \mathsf{ff} && \boxed{\text{eval-var -}} \\ x/y &\equiv x \qquad (\text{where } x \notin FV(y)) && \boxed{\text{eval-var xy}} \end{aligned}$$

Shannon's law may now be proven by structural induction over expressions.

We also have the following useful derived rule:

$$\textbf{if}(p,A,B) \equiv \textbf{if}(p,(A/p),(B/\overline{p})) \qquad \boxed{\text{meta-eval}}$$

This rule can be proven by using Shannon's law and the nesting laws. The importance of this rule is that it justifies optimisation at lower levels in the expression graph, within the context of appropriate case analyses nearer the root of the graph.

There are a number of other useful derived laws that involve evaluation:

$$\begin{aligned} (E/p) &\equiv E \qquad (\text{where } p \notin FV(E)) && \boxed{\text{independance}} \\ (E/p)/p &\equiv (E/p) && \boxed{\text{eval-eval +}} \\ (E/p)/\overline{p} &\equiv (E/p) && \boxed{\text{eval-eval -}} \\ (E/x)/y &\equiv (E/y)/x \qquad (\text{where } x \notin FV(y)) && \boxed{\text{eval-xy-yx}} \end{aligned}$$

and all of which may be proven via structural induction over expression structure. In addition, there is the following useful property:

$$p \notin (FV(E/p) \cup FV(E/\overline{p})) \qquad \boxed{\text{FV-eval}}$$

which amounts to the hiding property for quantified formulae. Note that 'eval-eval +' and 'eval-eval -' is an easy corollary of this law and 'independance'. However, they are useful properties and worth stating explicitly.

Significantly for BDD construction, we may also give an interpretation of the $(\_/\_)$ notation for propositional functions that is consistent with the interpretation given above for boolean expressions. If P is a propositional function, with *named* parameters : $v_1, \cdots, v_n$, then:

$$\begin{aligned} (P/v_k) &\equiv \lambda(v_1, \cdots, v_{k\text{-}1}, v_{k+1}, \cdots, v_n) \bullet P(v_1, \cdots, \mathsf{tt}, \cdots, v_n) \\ (P/\overline{v_k}) &\equiv \lambda(v_1, \cdots, v_{k\text{-}1}, v_{k+1}, \cdots, v_n) \bullet P(v_1, \cdots, \mathsf{ff}, \cdots, v_n) \end{aligned}$$

Hence, (P/l) produces a new function by essentially 'fixing' the value of one of its parameters.

3 Normal forms for if expressions

In this section, we shall define a number of normal forms for **if** expressions which can be useful during verification.

3.1 Terminology

First of all, we shall precisely define our use of the terminology used for discussing normal forms and such like. So, let *Expr* stand for some class of expressions and suppose that $Sem\colon Expr \to Val$ stand for a given semantic function.

Now, let $NF \subseteq Expr$. We say that *NF* is a *normal form set* for *Expr* (relative to *Sem*) whenever we have that:

$$\forall\, e \in Expr \bullet \exists\, n \in NF \bullet Sem(e) = Sem(n)$$

We define the notion of a *normal form function* as a function $Nf\colon Expr \to Expr$ such that:

(1) $(Sem\ o\ Nf) = Sem$

(2) $(Nf\ o\ Nf) = Nf$

A straightforward consequence of this is:

$\{Nf(e) \mid e \in Expr\}$ is a normal form set.

Clearly, the identity function is a normal form function, implying that *Expr* is a normal form set.

To discuss the notion of canonical form, we let $(\_ \simeq \_)$ stand for some equivalence relation over *Expr*, which preserves the given semantics for *Expr*:

$$\forall\, e_1, e_2 \in Expr \bullet Sem(e_1) = Sem(e_2) \;\Rightarrow\; e_1 \simeq e_2$$

Let $CF \subseteq Expr$ and we say that *CF* is a *canonical form set* for *Expr* with respect to equivalence relation $(\_ \simeq \_)$:

(1) *CF* is a normal form set for *Expr*

(2) $\forall\, e \in Expr \bullet \exists!\, c \in CF \bullet e \simeq c$

We define the notion of *canonical form function* with respect to equivalence relation $(\_ \simeq \_)$ as a function $Cf\colon Expr \to Expr$ such that:

(1) *Cf* is a normal form function for *Expr*

(2) $\forall\, e \in Expr \bullet e \simeq Cf(e)$

(3) $\forall\, e_1, e_2 \in Expr \bullet Cf(e_1) = Cf(e_2) \;\Leftrightarrow\; e_1 \simeq e_2$

The first property requires that Cf is a normal form function. The second property says that Cf preserves the given equivalence and the final property says that Cf *characterises* the equivalence up to identity.

3.2 Particular normal forms

In the remainder of this section, we shall define a number of normal forms for boolean expressions, paying particular attention to **if** expressions. Our first example is defined by presenting a particular normal form function such that the image corresponds to the normal form set. The remaining examples indirectly prescribe normal form sets by using predicates, leaving the construction of corresponding normal form functions open.

3.2.1 Negation Normal Form (NNF)

This normal form is one that applies generally to propositional logic expressions. The basic idea is to take the given formula and to 'push' all the explicit negations down towards the atomic formulae (using De Morgan's laws), and cancelling double negations as necessary. The result is a formula lacking any explicit occurrences of negation, although there may be occurrences of negative literals. Formally, the rules for performing this are:

$$\begin{aligned}
\neg\,\mathsf{tt} &\equiv \mathsf{ff} & \boxed{\text{Not tt}}\\
\neg\,\mathsf{ff} &\equiv \mathsf{tt} & \boxed{\text{Not ff}}\\
\neg(\neg E) &\equiv E & \boxed{\text{Not Not}}\\
\neg p &\equiv \overline{p} & \boxed{\text{Not lit}}\\
\neg(A \wedge B) &\equiv (\neg A \vee \neg B) & \boxed{\text{Not And}}\\
\neg(A \vee B) &\equiv (\neg A \wedge \neg B) & \boxed{\text{Not Or}}\\
\neg(A \Rightarrow B) &\equiv (A \wedge \neg B) & \boxed{\text{Not Imp}}\\
\neg(A \oplus B) &\equiv (A \Leftrightarrow B) & \boxed{\text{Not Xor}}\\
\neg(A \Leftrightarrow B) &\equiv (A \oplus B) & \boxed{\text{Not Iff}}\\
\neg\,\mathbf{if}(A,B,C) &\equiv \mathbf{if}(A,\neg B,\neg C) & \boxed{\text{Not cond}}
\end{aligned}$$

Using the above rules for negation as motivation, we define an appropriate normal form function $Nnf : Expr \to Expr$. Some of the rules for doing this are presented below:

$$
\begin{aligned}
Nnf(\mathsf{tt}) &\equiv \mathsf{tt} && \boxed{\text{Nnf tt}}\\
Nnf(\mathsf{ff}) &\equiv \mathsf{ff} && \boxed{\text{Nnf ff}}\\
Nnf(p) &\equiv p && \boxed{\text{Nnf lit}}\\
Nnf(A \wedge B) &\equiv (Nnf(A) \wedge Nnf(B)) && \boxed{\text{Nnf And}}\\
Nnf(\mathbf{if}(A,B,C)) &\equiv \mathbf{if}(Nnf(A),Nnf(B),Nnf(C)) && \boxed{\text{Nnf cond}}\\
Nnf(\neg\,\mathsf{tt}) &\equiv \mathsf{ff} && \boxed{\text{Nnf Not tt}}\\
Nnf(\neg\,\mathsf{ff}) &\equiv \mathsf{tt} && \boxed{\text{Nnf Not ff}}\\
Nnf(\neg\, p) &\equiv \overline{p} && \boxed{\text{Nnf Not lit}}\\
Nnf(\neg\,(\neg\, E)) &\equiv Nnf(E) && \boxed{\text{Nnf Not Not}}\\
Nnf(\neg\,(A \wedge B)) &\equiv (Nnf(\neg\, A) \vee Nnf(\neg\, B)) && \boxed{\text{Nnf Not And}}\\
Nnf(\neg\,\mathbf{if}(A,B,C)) &\equiv \mathbf{if}(Nnf(A),Nnf(\neg\, B),Nnf(\neg\, C)) && \boxed{\text{Nnf Not cond}}
\end{aligned}
$$

We define $NNF \subseteq Expr$ to be the fixpoint set of the normal form function Nnf defined above:

$$e \in NNF \;\Leftrightarrow\; Nnf(e) = e$$

Although NNF is, by the above definition, a normal form set for $Expr$, it is easily seen that NNF is not canonical, since:

$$
\begin{aligned}
&Nnf(\neg\,(x \wedge (y \vee \overline{z}))) = (\overline{x} \vee (\overline{y} \wedge z))\\
&Nnf(\neg\,((y \wedge x) \vee (x \wedge \overline{z}))) = ((\overline{y} \vee \overline{x}) \wedge (\overline{x} \vee z))
\end{aligned}
$$

even though $x \wedge (y \vee \overline{z}) \equiv (y \wedge x) \vee (x \wedge \overline{z})$.

3.2.2 Literal Normal Form (*LNF*)

We now consider a normal form set that is specified by predicates upon structure, rather than by some particular transformation function, as previously for *NNF*.

Let $E \in Expr$. We say that E is in *Literal Normal Form (or LNF)* whenever:

(1) $E \in \mathbb{B}$

or

(2) $E = \mathbf{if}(A,B,C)$, for some $A,B,C \in Expr$
such that :
$(A \in Lit) \wedge B, C \in LNF.$

This asserts that the set *LNF* consists of the ground boolean constants and **if** expressions whose control expression is a literal and whose logical arms are also in *LNF*.

Using many of the laws given earlier, particularly those that rearrange **if** expressions, we can show that there exists an element of *LNF* which is equivalent to any expression selected from *Expr*.

3.2.3 Reduced Normal Form (RNF)

Reduced Normal Form (RNF) is a refinement of *LNF*. Let $E \in Expr$ and we say that E is in *Reduced Normal Form (or RNF)* whenever:

$$
\begin{array}{ll}
(1) & E \in \mathbb{B} \\
\text{or} & \\
(2) & E = \textbf{if}(A,B,C), \text{ for some } A,B,C \in Expr \\
 & \text{such that :} \\
 & \qquad (e \in LNF) \\
 & \wedge \quad A \in Lit_+ \wedge A \notin (FV(B) \cup FV(C)) \\
 & \wedge \quad (B \neq C) \wedge B,C \in RNF.
\end{array}
$$

We can informally see that *RNF* is a normal form set by referring to several of the laws mentioned earlier. By appeal to Shannon's Law, we can require that once a variable occurs as control for a **if** expression, then it will not occur freely in either of the logical arms. By using the law 'Hobson's choice', we can require that the logical arms of every **if** expression are always distinct expressions. If they are the same, then this law permits the **if** expression to be replaced by the common value of the logical arms. Finally, we can require that only positive literals occur within control expressions, by using either one of the laws 'Not cond' or 'distr 1'.

It turns out to be more practical to permit signed literals to occur in places other than the control position of **if** expressions. For this reason, we accept the following slight extension of *RNF*: Let $E \in Expr$ and we say that E is in *Reduced Normal Form*$_1$ *(or RNF*$_1$*)* whenever:

$$
\begin{array}{ll}
(1) & E \in \mathbb{B} \\
\text{or} & \\
(2) & E \in Lit \\
\text{or} & \\
(3) & E = \textbf{if}(A,B,C), \text{ for some } A,B,C \in Expr \\
 & \text{such that :} \\
 & \qquad (E \in LNF) \\
 & \wedge \quad A \in Lit_+ \wedge A \notin (FV(B) \cup FV(C)) \\
 & \wedge \quad (B \neq C) \wedge \{B,C\} \not\subseteq \mathbb{B} \\
 & \wedge \quad B,C \in RNF_1.
\end{array}
$$

This permits literals to occur in the logical arms of **if** expressions, without semantic ambiguity. Note that in this case both positive and negative literals may be needed.

In what follows, the slight difference between *RNF* and RNF_1 is hardly ever significant. Unless otherwise indicated, we will use *RNF* to mean RNF_1 also.

Finally, we note that *RNF* is also not a canonical form for *Expr*, since the arrangement of literals in the expression graph is not fixed, so permitting differing decompositions due to the choice of variables on which to do case analysis. For instance, we have:

$$x \wedge y \equiv \textbf{if}(x, y, \mathsf{ff})$$
$$x \wedge y \equiv \textbf{if}(y, x, \mathsf{ff})$$

This aspect is considered in the next and final normal form, called *Reduced, Ordered Normal Form*.

3.2.4 Reduced, Ordered Normal Form (RONF)

As alluded to above, *RONF* is a refinement of *RNF* which adds in a constraint on the order in which variables are used in case analysis.

We now deploy the machinery introduced above for our definition of *RONF*. Let $E \in Expr$ and suppose that $\preceq$ is a given partial ordering upon $V \subseteq Lit_+$ such that $UL(e) \subseteq V$. We say that E is in *Reduced, Ordered Normal Form* (with respect to variable ordering $\preceq$) whenever:

(1) $E \in \mathbb{B}$

or

(2) $E = \textbf{if}(A, B, C)$, for some $A, B, C \in Expr$ such that :

$$(e \in RNF)$$
$$\wedge \quad UL(e) \subseteq \downarrow pos(A)$$
$$\wedge \quad B, C \in RONF.$$

The set $\downarrow pos(A)$ consists of any literal that is *less than or equal* to some value in the set $pos(A)$ (see Appendix **??** for more details):

$$\downarrow pos(A) = \{l \in Lit_+ \mid \exists\, p \in pos(A) \cdot l \preceq p\}$$

Therefore, the main difference between *RNF* and *RONF* is that the free variables of any **if** expression in *RONF* must lie amongst the lower bounds of $pos(A)$ with respect to $\preceq$. Notice that we have not yet required the ordering $\preceq$ to be total.

3.3 Theorems about Normal Forms

We have the following important theorems about the normal forms *RNF* and *RONF*. But before stating the first theorem, we need to define what is meant be a *contingent* formula: Let $A \in Expr$, and we say that A is *contingent* whenever it is neither a tautology nor a contradiction (i.e. neither $A \equiv \mathsf{tt}$ not $A \equiv \mathsf{ff}$ are valid). We say that A is *tautologous* when A is valid and A is *contradictory* when $\neg A$ is valid.

Theorem 3.1 (The Contingency Theorem for *RNF*)
Let $A \in RNF$. Then:

1. $A \notin \mathbb{B} \;\Leftrightarrow\; A$ is contingent
2. A is tautologous $\;\Leftrightarrow\; A = \mathsf{tt}$
3. A is contradictory $\;\Leftrightarrow\; A = \mathsf{ff}$
4. A is satisfiable $\;\Leftrightarrow\; A \neq \mathsf{ff}$
5. A is falsifiable $\;\Leftrightarrow\; A \neq \mathsf{tt}$

The fact that it is straightforward to determine tautology-hood for formulae in *RNF* suggests that the procedure for bringing arbitrary expressions into *RNF* is potentially expensive (i.e. exponential in cost) in the worst case. However, notice that satisfiability can always be deduced as soon as there is sufficient computational evidence that the expression is not equivalent to ff (and dually for falsifiability). Also, contingency implies both satisfiability and falsifiability. Contingency of formulae is further explored in Section 4 below.

The following theorem justifies a form of canonical form property for expressions in *RONF*:

Theorem 3.2 (The RONF canonical form theorem)
Let $V \subseteq Lit_+$ and further suppose that $\preceq\lfloor_V$ is a *total* ordering on V. Let $Expr_V = \{E \in Expr \mid UL(E) \subseteq V\}$. Then:

- *RONF* (relative to ordering $\preceq$) provides a canonical form set for $Expr_V$, with respect to semantic equivalence ($\_ \equiv \_$).

It is this theorem that justifies the construction of BDDs as canonical representations of formulae, since BDDs correspond to **if** expressions in *RONF* such that the ordering on positive literals, $\preceq$ is constrained to be total.

4 Contingency, effective variables and dimension

In an earlier section, we saw that an expression A was said to be *contingent* when it is neither tautologous nor contradictory. By De Morgan's laws, we have the following:

$$
\begin{aligned}
&A \text{ is tautologous} \;\Leftrightarrow\; \neg(A \text{ is falsifiable}) && \boxed{\text{tautologous}}\\
&A \text{ is contradictory} \;\Leftrightarrow\; \neg(A \text{ is satisfiable}) && \boxed{\text{contradictory}}\\
&A \text{ is contingent} \;\Leftrightarrow\; (A \text{ is satisfiable}) \wedge (A \text{ is falsifiable}) && \boxed{\text{contingence}}
\end{aligned}
$$

Furthermore, the following useful laws show that the above properties are semantically invariant, and hence significant:

$(A \equiv B) \;\Rightarrow\; A \text{ is satisfiable} \;\Leftrightarrow\; B \text{ is satisfiable}$ $\boxed{\equiv \text{ — contingence}}$

$(A \equiv B) \;\Rightarrow\; A \text{ is contingent} \;\Leftrightarrow\; B \text{ is contingent}$ $\boxed{\equiv \text{ — satisfiablity}}$

$(A \equiv B) \;\Rightarrow\; A \text{ is falsifiable} \;\Leftrightarrow\; B \text{ is falsifiable}$ $\boxed{\equiv \text{ — falsifiability}}$

We also have the following useful theorems concerning contingency, satisfiability, and falsifiability:

$A \text{ is contingent} \;\Leftrightarrow\; (\neg A) \text{ is contingent}$ $\boxed{\neg \text{ — contingence}}$

$(A \wedge B) \text{ is contingent} \;\Leftrightarrow\; (A \vee B) \text{ is contingent}$ $\boxed{\wedge\text{— } \vee \text{ contingence}}$

$(A \;\Leftrightarrow\; B) \text{ is contingent} \;\Leftrightarrow\; (A \oplus B) \text{ is contingent}$ $\boxed{\Leftrightarrow \text{ —}\oplus \text{ contingence}}$

$l \text{ is contingent}$ $\boxed{\text{literal contingence}}$

Note that since $(A \vee B) \;\Rightarrow\; A$ is contingent and $A \;\Rightarrow\; (A \vee B)$ is tautologous, we can say little about the preservation of contingency by implication. However, as $(A \;\Rightarrow\; B) \;\Leftrightarrow\; ((\neg B) \;\Rightarrow\; (\neg A))$, we have that:

$(A \;\Rightarrow\; B) \text{ is contingent} \;\Leftrightarrow\; ((\neg B) \;\Rightarrow\; (\neg A)) \text{ is contingent}$ $\boxed{\Rightarrow \text{ contingence}}$

Related to the above are the following satisfiability properties:

$(\neg A) \text{ is satisfiable} \;\Leftrightarrow\; A \text{ is falsifiable}$ $\boxed{\neg \text{ satisfiability}}$

$(A \wedge B) \text{ is satisfiable} \;\Rightarrow\; A \text{ is satisfiable} \wedge B \text{ is satisfiable}$ $\boxed{\wedge \text{ satisfiability}}$

$(A \vee B) \text{ is satisfiable} \;\Leftrightarrow\; A \text{ is satisfiable} \vee B \text{ is satisfiable}$ $\boxed{\vee \text{ satisfiability}}$

$(A \;\Rightarrow\; B) \text{ is satisfiable} \;\Leftrightarrow\; A \text{ is falsifiable} \vee B \text{ is satisfiable}$ $\boxed{\Rightarrow \text{ satisfiability}}$

$(A \;\Leftrightarrow\; B) \text{ is satisfiable} \;\Rightarrow$ $\boxed{\Leftrightarrow \text{ satisfiability}}$

$\qquad (A \text{ is satisfiable} \wedge B \text{ is satisfiable})$

$\quad \vee \quad (A \text{ is falsifiable} \wedge B \text{ is falsifiable})$

$(A \oplus B) \text{ is satisfiable} \;\Rightarrow$ $\boxed{\oplus \text{ satisfiability}}$

$\qquad (A \text{ is satisfiable} \wedge B \text{ is falsifiable})$

$\quad \vee \quad (A \text{ is falsifiable} \wedge B \text{ is satisfiable})$

$\mathtt{tt} \text{ is satisfiable}$ $\boxed{\text{tt satisfiability}}$

$l \text{ is satisfiable}$ $\boxed{\text{literal satisfiability}}$

which have their duals in terms of falsifiability:

$(\neg A)$ is falsifiable $\Leftrightarrow$ A is satisfiable $\boxed{\neg \text{ falsifiability}}$

$(A \wedge B)$ is falsifiable $\Leftrightarrow$ A is falsifiable $\vee$ B is falsifiable $\boxed{\wedge \text{ falsifiability}}$

$(A \vee B)$ is falsifiable $\Rightarrow$ A is falsifiable $\wedge$ B is falsifiable $\boxed{\vee \text{ falsifiability}}$

$(A \Rightarrow B)$ is falsifiable $\Rightarrow$ A is satisfiable $\wedge$ B is falsifiable $\boxed{\Rightarrow \text{ falsifiability}}$

$(A \Leftrightarrow B)$ is falsifiable $\Rightarrow$ $\boxed{\Leftrightarrow \text{ falsifiability}}$

$(A$ is satisfiable $\wedge$ B is falsifiable$)$

$\vee$ $(A$ is falsifiable $\wedge$ B is satisfiable$)$

$(A \oplus B)$ is falsifiable $\Rightarrow$ $\boxed{\oplus \text{ falsifiability}}$

$(A$ is satisfiable $\wedge$ B is satisfiable$)$

$\vee$ $(A$ is falsifiable $\wedge$ B is falsifiable$)$

ff is falsifiable $\boxed{\text{ff falsifiability}}$

l is falsifiable $\boxed{\text{literal falsifiability}}$

We can derive from these the following properties, for arbitrary $A \in Expr$:

A is satisfiable $\vee$ A is falsifiable $\boxed{\text{meta dichotomy}}$

A is tautologous $\oplus$ A is contingent $\oplus$ A is contradictory $\boxed{\text{meta trichotomy}}$

4.0.1 Effective variables and dimension

Intuitively, a boolean expression, E represents a propositional function, F, from its free variables (in some order) to the booleans. If the value of this expression is constant, no matter what the value of variable v is assigned to then we say that variable v is *ineffective* for expression E.

More formally, we can define the notion of an *effective* variable as follows. Let $E \in Expr$ and $v \in Lit$:

$$v \text{ is an } \textit{effective} \text{ variable of } E \Leftrightarrow \models (E/v \oplus E/\overline{v})$$

We denote the collection of all effective variables of an expression, A, by $EV(A)$, and defined by:

$$EV(A) \underline{\triangle} \{pos(v) \mid v \text{ is an effective variable of } A\}$$

We may further define the *dimension* of an expression A, denoted by $\dim A$ to be the cardinality of the set $EV(A)$. Formally we have that:

$$\dim A \underline{\triangle} \mathsf{card}\, EV(A)$$

5 QBF : Quantified Boolean Formulae

We can introduce *bounded quantification* over booleans by the following definitions:

$$
\begin{array}{lcl}
(\exists\, v \bullet E) & \triangleq & (E/v) \vee (E/\overline{v}) \\
(\forall\, v \bullet E) & \triangleq & (E/v) \wedge (E/\overline{v})
\end{array}
$$

The usual duality properties are easily shown to hold (from the above definitions and De Morgan's laws):

$$\neg(\forall\, p \bullet E) \;\equiv\; (\exists\, p \bullet \neg E) \qquad \boxed{\text{Not All}}$$

$$\neg(\exists\, p \bullet E) \;\equiv\; (\forall\, p \bullet \neg E) \qquad \boxed{\text{Not Some}}$$

The above may be stated a little more compactly by extending the $\overline{(\_)}$ notation to the quantifier symbols $\forall, \exists$:

$$
\begin{array}{lcl}
\overline{\forall} & = & \exists \\
\overline{\exists} & = & \forall
\end{array}
$$

and by introducing the following meta operator $\mathsf{qop}_{\_}$:

$$
\begin{array}{lcl}
\mathsf{qop}_{\forall} & = & \wedge \\
\mathsf{qop}_{\exists} & = & \vee
\end{array}
$$

Using this notation, we could have defined the quantifiers and exhibited their duality properties by the following couple of equations:

$$
\begin{array}{l}
(Q\, v \bullet E) \;\;\equiv (E/v\ \mathsf{qop}_Q E/\overline{v}) \\
\neg(Q\, v \bullet E) \equiv \overline{Q} v \bullet \neg E
\end{array}
$$

To be able to compute with quantified formulae, we need to explore the way that quantifiers interact with general expressions, especially **if** forms, where $Q \in \{\forall, \exists\}$:

$$(Q\, p \bullet E) \;\equiv\; E \qquad (\text{where } p \notin FV(E)) \qquad \boxed{\text{Q-null}}$$

$$(Q\, \overline{p} \bullet E) \;\equiv\; (Q\, p \bullet E) \qquad \boxed{\text{Q-neg}}$$

$$(Q\, p \bullet (Q\, p \bullet E)) \;\equiv\; (Q\, p \bullet E) \qquad \boxed{\text{Q-dup}}$$

$$(Q\, p \bullet (Q\, q \bullet E)) \;\equiv\; (Q\, q \bullet (Q\, p \bullet E)) \qquad \boxed{\text{Q-swop}}$$

$$
\begin{array}{l}
(Q\, p \bullet \mathbf{if}(A,B,C)) \;\;\equiv \\
\qquad \mathbf{if}((A/p),(B/p),(C/p))\ \mathsf{qop}_Q \mathbf{if}((A/\overline{p}),(B/\overline{p}),(C/\overline{p}))
\end{array}
\qquad \boxed{\text{Q-cond}}
$$

$$
\begin{array}{l}
(Q\, p \bullet \mathbf{if}(A,B,C)) \;\;\equiv \\
\qquad \mathbf{if}(A,(Q\, p \bullet B),(Q\, p \bullet C)) \qquad (\text{where } p \notin FV(A))
\end{array}
\qquad \boxed{\text{Q-non-cntrl}}
$$

$$(Q\, p \bullet \mathbf{if}(p,B,C)) \;\equiv\; (B/p)\ \mathsf{qop}_Q (C/\overline{p}) \qquad \boxed{\text{Q-cntrl-var}}$$

$$(Q\, p \bullet \mathbf{if}(\overline{p},B,C)) \;\equiv\; (B/\overline{p})\ \mathsf{qop}_Q (C/p) \qquad \boxed{\text{Q-cntrl-neg-var}}$$

The above rules show that quantifier operations can be efficient for **if** expressions with literals as control expressions. Note that, in the case that $\mathbf{if}(x,B,C) \in RNF$ (see Section 3.2.3), several rules can be simplified even further:

$$\begin{aligned}
&(Q\, p \bullet \mathbf{if}(x,B,C)) \;\equiv \\
&\quad \mathbf{if}(x,(Q\, p \bullet B),(Q\, p \bullet C)) \\
&\qquad (\text{where } p \notin FV(x))
\end{aligned} \qquad \boxed{\text{Q-lnf-non-cntrl}}$$

$$(Q\, x \bullet \mathbf{if}(x,B,C)) \;\equiv\; B \;\mathsf{qop}_Q C \qquad \boxed{\text{Q-lnf-var}}$$

$$(Q\, x \bullet \mathbf{if}(\overline{x},B,C)) \;\equiv\; B \;\mathsf{qop}_Q C \qquad \boxed{\text{Q-lnf-neg-var}}$$

5.1 Multiple quantification

The single-bit boolean quantifiers can be extended to cope with multiple quantification in a standard way. This is defined as follows, for $n \geq 1$ and where $Q \in \{\forall, \exists\}$:

$$Q\, v_1, \cdots, v_n, v_{n+1} \bullet E \quad \equiv \quad Q\, v_1, \cdots, v_n \bullet (Q\, v_{n+1} \bullet E)$$

Since we have the commutativity property, 'Q-swop', we can see that, for any permutation $\pi \in S_n$ (the *symmetric group* on n elements):

$$(Q\, v_1, \cdots, v_n \bullet E) \quad \equiv \quad (Q\, v_{\pi(1)}, \cdots, v_{\pi(n)} \bullet E)$$

In particular, we can reorder the sequence of quantified variables to try and reduce the expense of computing normal forms. For instance, useful heuristics might be based upon the heights that variables occur within the expression graph for E.

5.2 Efficient multiple quantification

The above equations suggest an iterative algorithm for computing multiple quantifications. This involves a pass over the expression for each variable in the quantification. A more efficient algorithm is suggested by the following formulation - where only a single pass is used. We shall proceed by defining the form:

$$\mathsf{mq}(Q, S, E)$$

where $Q \in \{\exists, \forall\}, E \in LNF \subseteq Expr$ and $S \subseteq Lit_+$.

We emphasise that $\mathsf{mq}(Q, S, E)$ is only defined in the case that expression E is in LNF.

$$\mathsf{mq}(Q\,,\emptyset,E) \;\equiv\; E \qquad \boxed{\text{mq-empty}}$$

$$\mathsf{mq}(Q\,,S,p) \;\equiv\; p \qquad (\text{where } pos(p) \notin S) \qquad \boxed{\text{mq-non-lit}}$$

$$\mathsf{mq}(\forall,S,p) \;\equiv\; \mathsf{ff} \qquad (\text{where } pos(p) \in S) \qquad \boxed{\text{mq-all-lit}}$$

$$\mathsf{mq}(\exists,S,p) \;\equiv\; \mathsf{tt} \qquad (\text{where } pos(p) \in S) \qquad \boxed{\text{mq-some-lit}}$$

$$\neg\,\mathsf{mq}(Q\,,S,E) \;\equiv\; \mathsf{mq}(\overline{Q},S,\neg\,E) \qquad \boxed{\text{mq-dual}}$$

$$\mathsf{mq}(Q\,,S,\mathbf{if}(a,B,C)) \;\equiv \qquad \boxed{\text{mq-cond 1}}$$
$$\mathbf{if}(a,\mathsf{mq}(Q\,,S,B),\mathsf{mq}(Q\,,S,C))$$
$$(\text{where } pos(a) \notin S)$$

$$\mathsf{mq}(Q\,,S,\mathbf{if}(p,B,C)) \;\equiv \qquad \boxed{\text{mq-cond 2}}$$
$$\mathsf{mq}(Q\,,S',(B/p\;\mathsf{qop}_Q C/\overline{p}))$$
$$(\text{where } pos(p) \in S \wedge S' = S - UL(p))$$

$$\mathsf{mq}(Q\,,S,(B\;\mathsf{qop}_Q C)) \;\equiv \qquad \boxed{\text{mq-distr qop}}$$
$$\mathsf{mq}(Q\,,S,B)\;\mathsf{qop}_Q \mathsf{mq}(Q\,,S,C))$$

When we can assume that $\mathbf{if}(p,B,C) \in RNF$, then we can simplify 'mq-cond 2' to give, for $pos(p) \in S \wedge S' = S - UL(p)$:

$$\mathsf{mq}(Q\,,S,\mathbf{if}(p,B,C)) \;\equiv\; \mathsf{mq}(Q\,,S',(B\;\mathsf{qop}_Q C)) \qquad \boxed{\text{mq-cond 2 rnf}}$$

Using the above equations, we can show that the following property holds:

Theorem 5.1 (multiple quantification theorem)
For $E \in LNF$, we have that:

$$(Q\;v_1,\;\cdots\;,v_k \bullet E) \equiv \mathsf{mq}(Q\,,pos(\{v_1,\;\cdots\;,v_k\}),E)$$

6 Substitution

In this section, we shall explore the notion of substitution and how it interacts with **if** expressions. Let A,B be expressions and suppose that p is a literal. We denote the result of substituting B for all free occurrences of p within A by the form $A\,[p{:}=B]$. This may be defined inductively over the structure of expressions according to the following equations:

$$p\,[p := B] \;\equiv\; B \qquad \boxed{\text{subst lit +}}$$

$$\overline{p}\,[p := B] \;\equiv\; \neg B \qquad \boxed{\text{subst lit -}}$$

$$x\,[p := B] \;\equiv\; x \qquad (\text{where } x \notin FV(p)) \qquad \boxed{\text{subst lit}}$$

$$C\,[p := B] \;\equiv\; C \qquad (\text{where } C \in \mathbb{B}) \qquad \boxed{\text{subst const}}$$

$$F(A)\,[p := B] \;\equiv\; F(A\,[p := B]\,) \qquad \boxed{\text{subst uop}}$$

$$(E_1\ Op\ \ E_2)\,[p := B] \;\equiv\; (E_1\,[p := B]\ \ Op\ \ E_2\,[p := B]\,) \qquad \boxed{\text{subst bop}}$$

$$\mathbf{if}(A, B, C)\,[p := X] \;\equiv \qquad \boxed{\text{subst cond}}$$
$$\qquad \mathbf{if}(A\,[p := X]\,, B\,[p := X]\,, C\,[p := X]\,)$$

$$(Q\ v \bullet A)\,[p := B] \;\equiv \qquad \boxed{\text{subst quant 1}}$$
$$\qquad (Q\ v \bullet A\,[p := B]\,)$$
$$\qquad\qquad (\text{where } p \notin FV(v))$$

$$(Q\ v \bullet A)\,[p := B] \;\equiv \qquad \boxed{\text{subst quant 2}}$$
$$\qquad (Q\ v \bullet A)$$
$$\qquad\qquad (\text{where } p \in FV(v))$$

Based upon the above, the following standard observations may be inferred, by structural induction over expressions:

$$A\,[x\!:=x] \;\equiv\; A \qquad \boxed{\text{subst identity}}$$

$$A\,[p\!:=B] \;\equiv\; A \qquad (\text{where } p \notin FV(A)) \qquad \boxed{\text{subst independance}}$$

$$A\,[p\!:=B]\;[q\!:=C] \;\equiv \qquad \boxed{\text{subst swop}}$$
$$A\,[q\!:=C]\;[p\!:=B]$$
$$(\text{where } q \notin FV(p) \wedge p \notin FV(C) \wedge q \notin FV(B))$$

$$A\,[p\!:=B]\;[q\!:=C] \;\equiv \qquad \boxed{\text{subst swop 1}}$$
$$(A\,[q\!:=C]\,)\,[p\!:=(B\,[q\!:=C]\,)]$$
$$(\text{where } q \notin FV(p) \wedge p \notin FV(C))$$

$$A\,[p\!:=B]\;[q\!:=C] \;\equiv \qquad \boxed{\text{subst swop 2}}$$
$$A\,[p\!:=(B\,[q\!:=C]\,)]$$
$$(\text{where } q \in FV(p) \wedge q \notin FV(A))$$

$$A\,[p\!:=q]\;[q\!:=B] \;\equiv \qquad \boxed{\text{subst rename}}$$
$$A\,[p\!:=B]$$
$$(\text{where } q \notin FV(p) \wedge q \notin FV(A))$$

The following derived rules concerning **if** expressions are also useful:

$$\mathbf{if}(p,B,C)\,[p\!:=X] \;\equiv\; \mathbf{if}(X,B\,[p\!:=X]\,,C\,[p\!:=X]\,) \qquad \boxed{\text{subst cond +}}$$

$$\mathbf{if}(\overline{p},B,C)\,[p\!:=X] \;\equiv\; \mathbf{if}(X,C\,[p\!:=X]\,,B\,[p\!:=X]\,) \qquad \boxed{\text{subst cond -}}$$

$$\mathbf{if}(x,B,C)\,[p\!:=X] \;\equiv \qquad \boxed{\text{subst cond lit}}$$
$$\mathbf{if}(x,B\,[p\!:=X]\,,C\,[p\!:=X]\,)$$
$$(\text{where } x \notin FV(p))$$

These may be further simplified, assuming that $\mathbf{if}(p,B,C) \in RNF$:

$$\mathbf{if}(p,B,C)\,[p\!:=X] \;\equiv\; \mathbf{if}(X,B,C) \qquad \boxed{\text{subst rnf cond +}}$$

$$\mathbf{if}(\overline{p},B,C)\,[p\!:=X] \;\equiv\; \mathbf{if}(X,C,B) \qquad \boxed{\text{subst rnf cond -}}$$

These hold because discrimination upon variable p implies that neither B nor C have p as a free variable where $\mathbf{if}(p,B,C) \in RNF$.

Now that we have formally introduced substitution, we can express a number of useful meta theorems involving renaming:

Theorem 6.1

1. (α-conversion theorem)

$$(Q\,x \bullet E) \equiv (Q\,y \bullet E\,[x := y]\,)$$
$$(\text{where } y \notin FV(E))$$

2. (Free variable renaming property)
Let $y \notin FV(x)$.

$$x \notin FV(E\,[x := y]\,)$$

(i.e. renaming x to y within E eliminates all free occurrences of x in E.)

Proof : Both by a structural induction over E, and using the above mentioned properties. □

There is a simple link between *evaluation* and *substitution*; in some sense, substitution generalises evaluation, as is shown by:

$$E/(+s) \equiv E\,[s := \mathsf{tt}\,] \qquad \boxed{\text{eval subst tt}}$$
$$E/(-s) \equiv E\,[s := \mathsf{ff}\,] \qquad \boxed{\text{eval subst ff}}$$
$$E/p \equiv E\,[p := \mathsf{tt}\,] \qquad \boxed{\text{eval subst}}$$

Note that p is an arbitrary *signed literal* in the law 'eval subst'.

We must also note the following result, that is essential for justifying the simplification within terms:

$$A \wedge (x \Leftrightarrow B) \equiv A\,[x := B\,] \wedge (x \Leftrightarrow B) \qquad \boxed{\text{iff subst}}$$

This may be proven by structural induction upon expression A.

The above law can be applied in showing the interesting association between existential quantification and substitution as revealed by the following propositions:

$$A\,[x := B\,] \equiv \exists\, x \bullet A \wedge (x \Leftrightarrow B) \qquad (\text{where } x \notin FV(B)) \qquad \boxed{\text{exist subst}}$$

$$A\,[x := B\,] \equiv \qquad \boxed{\text{gen exist subst 1}}$$
$$\exists\, y \bullet A\,[x := y\,] \wedge (y \Leftrightarrow B)$$
$$(\text{where } y \notin FV(\{A, B, x\}))$$

$$A\,[x := B\,] \equiv \qquad \boxed{\text{gen exist subst 2}}$$
$$\exists\, y \bullet (\exists\, x \bullet A \wedge (x \Leftrightarrow y)) \wedge (y \Leftrightarrow B)$$
$$(\text{where } y \notin FV(\{A, B, x\}))$$

The law 'exist subst' is the most fundamental of the above, and is proven by using the law 'iff subst'. The second law 'gen exist subst 1' is obtained by using 'exist subst' and properties of substitution such as 'subst rename' to change variable from x to y. Finally, 'gen exist subst 2' is obtained by replacing the embedded substitution using 'exist subst' again.

6.1 Simultaneous substitution

We can now introduce *simultaneous substitution* - a useful operation that is essential in defining the application of defined predicates. The first definition given below is naive, in that simultaneous substitution is expressed in terms of a sequence of simple substitutions. Later, a more efficient representation, in terms of mappings, is given that avoids the use of intermediate variables and multiple passes over the expression graph.

We briefly indicate what we mean by simultaneous substitution by recourse to the following example: Let E be equal to the expression:

$$((a \wedge \overline{b}) \vee (\exists\, \overline{v} \bullet a \wedge v \wedge c))$$

and consider the simultaneous substitution denoted by:

$$E\,[a{:}=(b \vee \overline{c}), \overline{b}{:}=(a \wedge x), c{:}=(a \vee v)]$$

Note that the assignments for a, b and c also involve the same names - a, b, and c. We interpret this by assuming that variables on the left hand side of assignments are acting as *placeholders* for the expressions on the right hand side. Consequently, the variables used in the expressions are not related to those occurring as placeholders.

Before computing the result, we note that there is a clash of free and bound variable in that v occurs free in the assignment for c and as a bound variable in E. Consequently, we use the 'α-conversion theorem' to change the quantified variable v to v', to produce:

$$((a \wedge \overline{b}) \vee (\exists\, \overline{v'} \bullet a \wedge v' \wedge c))$$

Using this form of E, we have the result:

$$(((b \vee \overline{c}) \wedge (a \wedge x)) \vee (\exists\, v' \bullet (b \vee \overline{c}) \wedge v' \wedge (a \vee v)))$$

This may be simplified as follows:

$$
\begin{aligned}
& ((b \vee \overline{c}) \wedge (a \wedge x)) \vee (\exists\, v' \bullet (b \vee \overline{c}) \wedge v' \wedge (a \vee v))) \\
\equiv\; & \\
& ((b \vee \overline{c}) \wedge (a \wedge x)) \vee ((b \vee \overline{c}) \wedge (a \vee v)) \\
\equiv\; & \\
& (b \vee \overline{c}) \wedge ((a \wedge x) \vee a \vee v) \\
\equiv\; & \\
& (b \vee \overline{c}) \wedge (a \vee v)
\end{aligned}
$$

Naively, we can operationally define simultaneous substitiution in terms of a sequence of simple substitutions as follows:

1. Rename all the placeholder variables to temporary, intermediate variables that are unused elsewhere (cf. the use of `gensym` in Lisp).

2. Perform the corresponding simple substitution for each of these new variables.

More formally, we have that:

$$
\begin{aligned}
&E\,[v_1 := E_1,\ \cdots\ , v_k := E_k] \equiv \\
&\qquad E\,[v_1 := t_1]\ \cdots\ [v_k := t_k]\,[t_1 := E_1]\ \cdots\ [t_k := E_k]
\end{aligned}
$$

where:

$$
\begin{aligned}
&t_i \notin (FV(E) \cup FV(E_1) \cup\ \cdots\ \cup FV(E_k)) \\
&t_i \notin (FV(v_1) \cup\ \cdots\ \cup FV(v_k)) \\
&FV(t_i) \cap FV(t_j) \neq \emptyset\ \Rightarrow\ i = j
\end{aligned}
$$

In essence, the use of intermediate variables decouples the order in which the simple substitutions may be done. In fact, for any $\theta, \pi \in S_k$, permutations of $\{1,\ \cdots\ , k\}$, we have that:

$$
\begin{aligned}
&E\,[v_1 := E_1,\ \cdots\ , v_k := E_k] \equiv \\
&\qquad E\,[v_{\theta(1)} := t_{\theta(1)}]\ \cdots\ [v_{\theta(k)} := t_{\theta(k)}]\,[t_{\pi(1)} := E_{\pi(1)}]\ \cdots\ [t_{\pi(k)} := E_{\pi(k)}]
\end{aligned}
$$

To see this, note that each of the simple substitutions on the right hand side are *independent* of each other and can therefore be permuted arbitrarily.

6.2 Mapping based description of simultaneous substitution

A more direct approach to simultaneous substitution uses mappings to represent the substitution, avoiding the use of intermediate variables and uses only a single pass over the expression graph. We shall proceed by introducing a new construct $E\,[m]$, where $m \in Lit \xrightarrow{\mathsf{map}} Expr$.

Before giving the inductive equations for $E\,[m]$, we need to (naturally) extend the literal operators from $Expr$ to $Lit \xrightarrow{\mathsf{map}} Expr$ as follows, for $LOp \in LitOp, m \in Lit \xrightarrow{\mathsf{map}} Expr$:

$$
LOp(m) \triangleq \bigcup(\{LOp(v) \mid v \in \mathsf{dom}\, m\} \cup \{LOp(e) \mid e \in \mathsf{rng}\, m\})
$$

The equations are:

$$p\,[m] \;\equiv\; A \qquad (\text{where } p \in \mathsf{dom}\, m \wedge A = m(p)) \qquad \boxed{\text{msubst lit +}}$$

$$\overline{p}\,[m] \;\equiv\; \neg A \qquad (\text{where } p \in \mathsf{dom}\, m \wedge A = m(p)) \qquad \boxed{\text{msubst lit -}}$$

$$v\,[m] \;\equiv\; v \qquad (\text{where } FV(v) \not\subseteq \mathsf{dom}\, m) \qquad \boxed{\text{msubst lit}}$$

$$F(E)\,[m] \;\equiv\; F(E\,[m]\,) \qquad \boxed{\text{msubst uop}}$$

$$(E_1\ Op\ E_2)\,[m] \;\equiv\; (E_1\,[m]\ Op\ E_2\,[m]\,) \qquad \boxed{\text{msubst bop}}$$

$$\mathbf{if}(A,B,C) \;\equiv\; \mathbf{if}(A\,[m]\,,B\,[m]\,,C\,[m]\,) \qquad \boxed{\text{msubst cond}}$$

$$(Q\,v \bullet E)\,[m] \;\equiv \qquad \boxed{\text{msubst quant 1}}$$
$$(Q\,v \bullet E\,[m]\,)$$
$$(\text{where } v \notin FV(m))$$

$$(Q\,v \bullet E)\,[m] \;\equiv \qquad \boxed{\text{msubst quant 2}}$$
$$(Q\,v' \bullet E\,[v{:=}\,v']\,[m]\,)$$
$$(\text{where } v' \notin (FV(E) \cup FV(m)))$$

$$(Q\,v \bullet E)\,[m] \;\equiv \qquad \boxed{\text{msubst quant 3}}$$
$$(Q\,v' \bullet E\,[m \dagger \{v \mapsto v'\}]\,)$$
$$(\text{where } v' \notin (FV(E) \cup FV(m)))$$

We have the following theorem:

Theorem 6.2 (multiple substitution theorem)
Let $E, E_1, \cdots, E_k \in Expr$ and $v_1, \cdots, v_k \in Lit$:

$$E\,[v_1{:=}\,E_1, \cdots, v_k{:=}\,E_k] = E\,[\{v_1 \mapsto E_1, \cdots, v_k \mapsto E_k\}\,]$$

6.3 Existential and Universal composition forms

It will turn out that the notion of *composition* of predicate forms is of great use in applications. There are two kinds of operators, dual to each other, and we give them a general presentation. Let $X \subseteq Lit$ and $A, B \in Expr$. We define the *Existential composition form* as follows:

$$(A\ \Delta_X B) \quad \underline{\triangle} \quad \exists\, X \bullet (A \wedge B)$$

and define the *Universal composition form* as:

$$(A\ \nabla_X B) \quad \underline{\triangle} \quad \forall\, X \bullet (A \vee B)$$

Note that we are defining composition *forms* here as opposed to composition *operators* — the distinction is that the choice of bound variables is *exposed* for these forms and so requires special rules for substitution and calculation of free literals and variables. A consequence of this difference is that the relational composition *operator* of two binary relations is non-commutative, whereas the corresponding existential composition form is *commutative*.

To formulate general rules concerning the composition forms, we let $\mathsf{cmp} \in \{\Delta, \nabla\}$ and use the meta operator compop_ defined by:

$$\begin{aligned}
\mathsf{compop}_\Delta &\equiv \wedge \\
\mathsf{compop}_\nabla &\equiv \vee
\end{aligned}$$

The following give a number of general rules:

$$FV((A\ \mathsf{cmp}_X B)) \equiv (FV(A) \cup FV(B)) \setminus X \qquad \boxed{\text{FV comp}}$$

$$(B\ \mathsf{cmp}_X A) \equiv (A\ \mathsf{cmp}_X B) \qquad \boxed{\text{comp commute}}$$

$$(A\ \mathsf{cmp}_X B) \equiv \qquad \boxed{\text{comp reduce}}$$
$$(A\ \mathsf{cmp}_{X'} B) \qquad (\text{where } X' = X \cap (FV(A) \cup FV(B)))$$

$$(A \mathsf{compop}_{\mathsf{cmp}} B) \equiv (A\ \mathsf{cmp}_\emptyset B) \qquad \boxed{\text{comp null}}$$

$$(A\ \Delta_X A) \equiv \exists\, X \bullet A \qquad \boxed{\text{Exi-comp absorp}}$$
$$(A\ \nabla_X A) \equiv \forall\, X \bullet A \qquad \boxed{\text{Uni-comp absorp}}$$

From the absorption rules above, we can see that these composition operators generalise the quantifiers.

7 Acknowledgement

Much of the above material has clearly drawn upon the work of Martin Peim and the many conversations that we have had over the last year concerning conceptual and implementation issues. In addition, several conversations with Howard Barringer, Graham Gough, Thomas Longshaw and Alan Williams have been helpful and encouraging.

A Partial orderings

To standardise notation, we briefly review various notions concerning partial orderings. Let $\sqsubseteq$ be any partial ordering on set A (i.e. $\sqsubseteq$ is a binary relation on A that is reflexive, anti-symmetric and transitive). We may say that $\sqsubseteq$ is a *total* ordering of A when:

$$\forall\, x, y{:}A \bullet (x \sqsubseteq y) \vee (y \sqsubseteq x)$$

A.1 Restriction

Let $B \subseteq A$ and we denote the *restriction* of $\sqsubseteq$ to subset B by $\sqsubseteq\lfloor_B$, defined by the equation:

$$\sqsubseteq\lfloor_B \quad \underline{\triangle} \quad (\sqsubseteq) \cap (B \times B)$$

A.2 Compatibility

We can define the notion of *compatibility* between two partial orderings defined upon subsets of A. Let $B_1, B_2 \subseteq A$ and let $\sqsubseteq_1, \sqsubseteq_2$ partially order the subsets B_1, B_2 respectively. We say that $\sqsubseteq_1$ *is compatible with* $\sqsubseteq_2$ whenever:

$$\begin{aligned} \exists\ \sqsubseteq_{12} \in\ & \mathbb{P}(B_1 \cup B_2)^2 \bullet \\ & (\sqsubseteq_{12}) \text{ is a partial ordering of } (B_1 \cup B_2) \\ \wedge\ & \sqsubseteq_1 = (\sqsubseteq_{12})\lfloor_{B_1} \\ \wedge\ & \sqsubseteq_2 = (\sqsubseteq_{12})\lfloor_{B_2} \end{aligned}$$

A.3 Above and below

The concepts of *above* and *below* arise frequently when discussing partial orders. The set of elements *above* $a \in A$ are denoted by:

$$\uparrow_A a \quad \underline{\triangle} \quad \{x \in A \mid a \sqsubseteq x\}$$

and, dually, the set of elements *below* $a \in A$ are denoted:

$$\downarrow_A a \quad \underline{\triangle} \quad \{x \in A \mid x \sqsubseteq a\}$$

When the partial ordering, A, is determined within the context we may write e.g. $\uparrow x$ instead of $\uparrow_A x$.

These operators have the following properties:

$$a \sqsubseteq b \;\Leftrightarrow\; b \in \uparrow a \qquad \boxed{\text{order above}}$$
$$a \sqsubseteq b \;\Leftrightarrow\; a \in \downarrow b \qquad \boxed{\text{order below}}$$
$$a \sqsubseteq b \;\Leftrightarrow\; \uparrow b \subseteq \uparrow a \qquad \boxed{\text{order above subset}}$$
$$a \sqsubseteq b \;\Leftrightarrow\; \downarrow a \subseteq \downarrow b \qquad \boxed{\text{order below subset}}$$

We shall, in addition, mention the following *binary* relations, for $a, b \in A$:

$$a \,\mathsf{above}\, b \;\Leftrightarrow\; a \in \uparrow b \qquad \boxed{\text{above}}$$
$$a \,\mathsf{below}\, b \;\Leftrightarrow\; a \in \downarrow b \qquad \boxed{\text{below}}$$

with obvious properties:

$$a \,\mathsf{above}\, b \;\Leftrightarrow\; b \sqsubseteq a \qquad \boxed{\text{above } \sqsubseteq}$$
$$a \,\mathsf{below}\, b \;\Leftrightarrow\; b \,\mathsf{above}\, a \qquad \boxed{\text{below above}}$$
$$a \,\mathsf{above}\, b \;\Leftrightarrow\; \uparrow b \subseteq \uparrow a \qquad \boxed{\text{above subset}}$$

These equations show that the concepts of above and below each encode the partial ordering information for $\_ \sqsubseteq \_$.

A.4 Upper and lower closed subsets

We now consider the idea of upper and lower closed subset. Let $S \subseteq A$ and we say that S is *upper closed* whenever:

$$\forall\, x, y \in A \bullet (x \in S) \wedge (x \sqsubseteq y) \;\Rightarrow\; y \in S$$

Dually, we say that S is *lower closed* whenever:

$$\forall\, x, y \in A \bullet (x \in S) \wedge (y \sqsubseteq x) \;\Rightarrow\; y \in S$$

By these definitions, we note that A is both an upper and lower closed subset of itself.

Given a subset $B \subseteq A$, we now define a couple of useful closure operators. The *upper closure* of B is denoted by $\uparrow B$ and defined to be:

$$\begin{aligned} \uparrow B \;&\triangleq\; \{x \in A \mid \exists\, a \in B \bullet a \sqsubseteq x\} \\ &\equiv\; \bigcup \{\uparrow x \mid x \in B\} \end{aligned}$$

The following properties are straightforward:

$$\uparrow a = \uparrow\{a\} \qquad \boxed{\text{singleton} \uparrow}$$
$$\uparrow(S_1 \cup S_2) = \uparrow S_1 \cup \uparrow S_2 \qquad \boxed{\text{union} \uparrow}$$
$$S_1 \subseteq S_2 \;\Rightarrow\; \uparrow S_1 \subseteq \uparrow S_2 \qquad \boxed{\text{monotone} \uparrow}$$
$$S \subseteq \uparrow S \qquad \boxed{\text{reflex} \uparrow}$$
$$\uparrow\uparrow S \subseteq \uparrow S \qquad \boxed{\text{trans} \uparrow}$$

Similarly, the *lower closure* of B is denoted by $\downarrow B$ and defined to be:

$$\begin{array}{lcl} \downarrow B & \triangleq & \{x \in A \mid \exists\, a \in B \bullet x \sqsubseteq a\} \\ & \equiv & \bigcup\{\downarrow x \mid x \in B\} \end{array}$$

The following properties are reasonably straightforward:

$$\downarrow a = \downarrow\{a\} \qquad \boxed{\text{singleton} \downarrow}$$
$$\downarrow(S_1 \cup S_2) = \downarrow S_1 \cup \downarrow S_2 \qquad \boxed{\text{union} \downarrow}$$
$$S_1 \subseteq S_2 \;\Rightarrow\; \downarrow S_1 \subseteq \downarrow S_2 \qquad \boxed{\text{monotone} \downarrow}$$
$$S \subseteq \downarrow S \qquad \boxed{\text{reflex} \downarrow}$$
$$\downarrow\downarrow S \subseteq \downarrow S \qquad \boxed{\text{trans} \downarrow}$$

We can now justify the terminological connection between these operators and the concepts of upper and lower closed subsets via the following theorems:

Theorem A.1 (Closed subsets and Closure operators)

(a) $\uparrow B$ is upper closed

(b) B is upper closed $\Leftrightarrow \uparrow B \subseteq B$
$\Leftrightarrow \uparrow B = B$
$\Leftrightarrow \exists\, S \subseteq A \bullet \uparrow S = B$

(a') $\downarrow B$ is lower closed

(b') B is lower closed $\Leftrightarrow \downarrow B \subseteq B$
$\Leftrightarrow \downarrow B = B$
$\Leftrightarrow \exists\, S \subseteq A \bullet \downarrow S = B$

Furthermore, there are optimality properties as well which define the closure operators:

Theorem A.2 (Optimality of Closure operators)

(a) $\forall\, S \subseteq A \bullet B \subseteq S \wedge S$ is upper closed $\Rightarrow \uparrow B \subseteq S$

(a') $\uparrow B \equiv \bigcap\{S \subseteq A \mid B \subseteq S \wedge S \text{ is upper closed }\}$

(b) $\forall\, S \subseteq A \bullet B \subseteq S \wedge S$ is lower closed $\Rightarrow \downarrow B \subseteq S$

(b) $\downarrow B \equiv \bigcap\{S \subseteq A \mid B \subseteq S \wedge S \text{ is lower closed }\}$

In a prototypical sense, all optimality (and so all induction) principles derive from the above.

Finally, we define the concept of *convex* subset and of *convex* closure. Let $S \subseteq A$ and we say that S is a *convex* subset whenever:

$$\forall\, x,y,z \in A \bullet \qquad (x \in S) \wedge (y \in S) \wedge (x \sqsubseteq z \sqsubseteq y) \;\Rightarrow\; z \in S$$

Clearly, A is itself a convex subset.

We define the *convex* closure operator, $\updownarrow S$, as follows:

$$\updownarrow S \triangleq \bigcap\{S \subseteq A \mid B \subseteq S \wedge S \text{ is convex }\} \qquad \boxed{\text{convex closure}}$$

This satisfies the following optimality and closure properties:

$$\begin{array}{ll}
\updownarrow S\text{is convex} & \boxed{\text{convex convex}} \\
\forall\, S \subseteq A \bullet & \boxed{\text{convex optimal}} \\
\qquad B \subseteq S \wedge S\text{is convex} \;\;\Rightarrow\;\; \updownarrow B \subseteq S & \\
S_1 \subseteq S_2 \;\;\Rightarrow\;\; \updownarrow S_1 \subseteq \updownarrow S_2 & \boxed{\text{monotone } \updownarrow} \\
S \subseteq \updownarrow S & \boxed{\text{reflex } \updownarrow} \\
\updownarrow\updownarrow S \subseteq \updownarrow S & \boxed{\text{trans } \updownarrow}
\end{array}$$

Finally, we can characterise convex closure in terms of upper and lower closure by the following observation:

$$\updownarrow S = \uparrow S \cap \downarrow S \qquad \boxed{\updownarrow\_ = \uparrow \wedge \downarrow}$$

A.5 Upper and lower bounds of sets

We now define the notion of *upper* and *lower* bounds for a subset $S \subseteq A$ of partial ordering A. We denote the set of all *upper bounds* of S by S^u, which is defined by:

$$\begin{array}{lll}
S^u & \triangleq & \{x \in A \mid \forall\, s \in S \bullet s \sqsubseteq x\} \\
 & \equiv & \bigcap\{\uparrow s \mid s \in S\}
\end{array}$$

Dually, we have the set of all *lower bounds* of S denoted by S^l and defined by:

$$\begin{array}{lll}
S^l & \triangleq & \{x \in A \mid \forall\, s \in S \bullet x \sqsubseteq s\} \\
 & \equiv & \bigcap\{\downarrow s \mid s \in S\}
\end{array}$$

We note the following properties of these operations:

S^u is upper closed $\boxed{\text{up closed}}$

$S^u \subseteq \uparrow S$ $\boxed{\text{up subset }\uparrow}$

$(S_1 \cap S_2)^u = S_1^u \cap S_2^u$ $\boxed{\text{up intersect}}$

S^l is lower closed $\boxed{\text{down closed}}$

$S^l \subseteq \downarrow S$ $\boxed{\text{down subset }\uparrow}$

$(S_1 \cap S_2)^l = S_1^l \cap S_2^l$ $\boxed{\text{down intersect}}$

Next, we introduce the concepts of *maximum* and *maximal* (resp. *minimum* and *minimal*). Let $S \subseteq A$:

x is maximal in $S \;\Leftrightarrow\; (x \in S) \wedge \forall\, b \in S \bullet x \sqsubseteq b \;\Rightarrow\; x = b$ $\boxed{\text{maximal}}$

x is minimal in $S \;\Leftrightarrow\; (x \in S) \wedge \forall\, b \in S \bullet b \sqsubseteq x \;\Rightarrow\; b = x$ $\boxed{\text{minimal}}$

Note that we have the following properties related to the empty subset, for any $a \in A$:

$\neg\,(a$ is maximal in $\emptyset)$ $\boxed{\text{empty maximal}}$

$\neg\,(a$ is minimal in $\emptyset)$ $\boxed{\text{empty minimal}}$

We then define the *suprema* and *infima* of S by:

$\sup S \;\underline{\triangle}\; \{x \in A \mid x \text{ is maximal in } S\}$ $\boxed{\text{sup defn}}$

$\inf S \;\underline{\triangle}\; \{x \in A \mid x \text{ is minimal in } S\}$ $\boxed{\text{inf defn}}$

Note that $\sup S \subseteq S$ and $\inf S \subseteq S$, for arbitrary $S \subseteq A$.

If $\mid \sup S \mid = 1$, we say that S has a *maximum* element, which we denote as $\max S$. Dually, if $\mid \inf S \mid = 1$, we say that S has a *minimum* element, which we denote as $\min S$.

Using the above, we can define *least upper bound* and *greatest lower bound* for a set S. When S^l has a maximum element, this value is called the *greatest lower bound* (abbrev. *glb*) and is denoted by $\sqcap S$. Similarly, when S^u has a minimum element, this value is called the *least upper bound* (abbrev. *lub*) and is denoted by $\sqcup S$. When these quantities exist, the following equations hold:

$\sqcup S = \min S^u$ $\boxed{\text{lub defn}}$

$\sqcap S = \max S^l$ $\boxed{\text{glb defn}}$

A.6 Lattices and Complete Lattices

References

[BCM$^+$90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS Proceedings*, 1990.

[BRB90] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[Bry91] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[MB88] J-C. Madre and J-P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 205–210, 1988.

[Sha38] C.E. Shannon. A symbolic analysis of relay and switching circuits. *Trans AIEE*, 57:713–723, 1938.

[Sha49] C.E. Shannon. The synthesis of two-terminal switching circuits. *Bell Syst. Tech. J.*, 28:59–98, 1949.