VESPER

J. Alan Robinson & Jonas Barklund
Syracuse University and Uppsala University*
Computing Science Department
Box 311, S-751 05 Uppsala, Sweden
Phone: +48-18-182500
Fax: +46-18-511925

Abstract

VESPER is a denotational formalism, that is, a collection of expressions each of which denotes some entity. Its logic, when viewed at the topmost level, is given by a single simplification function. The simplification is vertically extended in the sense that in a single act of simplification an expression and (simultaneously) all or most of its subexpressions are recast into a simpler form. All VESPER computations are reductions to normal form, in which the simplification function is iterated on a given expression until it has no further effect. Expressions are by definition in normal form if they are fixed points of the simplification function.

*This work was partly supported by Göran Gustafssons Stiftelse and the Swedish Research Council for Engineering Sciences (TFR) under grants 93:38 and 94-939, respectively.
1 Introduction

VESPER\(^1\) is a denotational formalism that is a collection of expressions each of which denotes some entity. Its logic when viewed at the topmost level is given by a single simplification function. The simplification is vertically extended in the sense that in a single act of simplification an expression and (simultaneously) all or most of its subexpressions are recast into a simpler form. All VESPER computations are reductions to normal form in which the simplification function is iterated on a given expression until it has no further effect. Expressions are by definition in normal form if they are fixed points of the simplification function.

VESPER was designed as a tool for exploring two ideas:

1. the idea that functional computing (based on the lambda-calculus) and relational computing (based on Horn clause resolution) are essentially the same thing namely logical simplification;

2. the idea that such logical simplifications can admit extensive parallelism.

There seems to be no absolute notion of parallel computation at least in the case of simplification. The simultaneous application of several different simplification functions to (different subexpressions of) an expression can always (trivially) be viewed instead (as we do here) as the application of a single more complex function to the entire expression. This vertically extended simplification will (if the expression is not in normal form) produce an expression which differs from it in several places but that fact alone does not disqualify these differences from being regarded collectively as those arising from a single distributed step.\(^2\)

Intuitively the simplification function decomposes an expression into its immediate constituents simplifies these and reassembles the resulting expressions into a perhaps different syntactic pattern.

When the expression to be simplified is an atomic symbol for which a user-declared definition is in force the symbol is replaced by the definitions of the definition. However such defined symbols are rewritten only at the

\(^1\)VESPER stands for Vertically Extended Simplification Parallel Expression Reduction, the computational logic described in this paper.

\(^2\)The familiar instantiation operation of simultaneously substituting \(n\) terms \(t_1, \ldots, t_n\) respectively for \(n\) variables \(x_1, \ldots, x_n\) throughout an expression \(E\) is an obvious example of this. We can regard it either as one single ‘large’ global operation performed on \(E\) by a ‘many-armed’ agent, or as \(n\) distinct ‘small’ local operations performed on \(E\) by \(n\) ‘one-armed’ agents working concurrently. Even substituting one term for one variable throughout \(E\) already poses the issue if there are many occurrences of the variable in \(E\). Substituting the term for each occurrence of the variable throughout \(E\) could itself be considered as the simultaneous (parallel) replacement of each occurrence of the variable \(x\) in \(E\) by an occurrence of the term \(t\).
top level of an expression and never when they are subexpressions of some other expression which is being simplified. The reason for this exception is that without it, the iteration of the simplification function would be free to ‘explode’ in a runaway unfolding of recursive definitions thus consuming all available resources.

2 Underlying model of a VESPER machine as an army of cooperating identical automata

Our purpose in this account is to state the vertically extended simplification logic of VESPER in an implementation-neutral way. However, we do have in mind a parallel implementation in which the expressions are represented as directed graphs. Each node in the graph is represented by an active processor of identical design. Each processor has its own unique address and represents an expression by virtue of the information currently in its various registers. This information records whether the expression is a conjunction, disjunction, set abstraction, etc., and gives the addresses of the other processors which represent its immediate subexpressions. It is possible for such a nodal processor both to read from and to write into the registers of other processors whose addresses it has in its own memory. Hard-wired inside each processor is a copy of the algorithm for applying VESPER’s simplification function and the processor repeatedly applies this function to the expression that it represents by changing the information in its registers so as to represent the new expression resulting from the simplification. Since all processors are active in this way all the time, the desired effect of simultaneous application of all local transformations is obtained. In general, the action of a nodal processor may call for new expressions to be formed in which case the necessary new nodes are represented by processors freshly allocated from a heap. The processor heap is replenished continuously by returning to it the processors whose expressions are no longer subexpressions of the expression being collectively represented by the [unique] root of the graph.

To realize this implementation in practice involves several difficult issues, which we will not discuss further here. The following account is strictly concerned with the calculus and its logic and does not deal with any implementational problems.

3 The language

VESPER is an applied, typeless, combined lambda-calculus and predicate-calculus. It has several familiar built-in or defined notions in addition to lambda abstraction and existential quantification: equality, numerals and numerical operators, lists, tuples, set abstractions, set itemizations, union expressions, boolean expressions, and conditional expressions.

Expressions are either atomic or composite. Atomic expressions are either
Constants, or abbreviations, or variables.

- **Constants** are either internal or external. In addition, they may also be declared to be Herbrand constants (see below).
  
  - **Internal constants** are:

    
    \[
    \text{AND OR NOT NULL SET } \cup \in \subseteq \\
    \bullet = + - \ast / < >
    \]

    and the numerals. The internal constant \( \bullet \) is an Herbrand constant.

  - **External constants** are atomic expressions which are declared to be such by the user.

All constants denote different entities; the numerals denote numbers\(^3\) and the other internal constants denote certain functions identified below.

- **Abbreviations** are atomic expressions which are distinct from the constants and which have been declared to be such. To declare an atomic expression \( \alpha \) to be an abbreviation one asserts a *definition* \( \alpha =_{\text{def}} \beta \Gamma \) where \( \beta \) is an expression different from \( \alpha \). The symbol \( \alpha \) is then regarded as simply “short for” the expression \( \beta \) (the *definiens* of the definition).

- **Variables** are atomic symbols which are neither constants nor abbreviations. Variables may be free or bound.

A *composite expression* is either a binding expression or an application.

- **Binding expressions.** These are expressions of one of the three forms

  \[
  (\lambda V B) \Gamma (\exists V B) \Gamma \{ V | B \}
  \]

  The first is a *lambda abstraction* \( \Gamma \) the second is an *existential quantification* \( \Gamma \) and the third is a *set abstraction* (it is read: the set of all \( V \) such that \( B \)). Each binding-expression has a bound-variable-list \( \Gamma \) which is a list of distinct variables \( \Gamma \) and a body \( B \Gamma \) which is an expression.

- An *application* is a nonempty list of expressions. Its *operator* is the first expression in the list \( \Gamma \) and the list of its *operands* is the rest of the list. For example, \( \Gamma \) the operator might be a defined constant \( \Gamma \) for a composite expression.

---

\(^3\) We will identify the natural numbers with certain indexing functions; cf. below.
The constants \texttt{AND}, \texttt{OR} and \texttt{NOT} denote the usual conjunction, disjunction and negation functions. The application of \texttt{AND} to no argument denotes truth, similarly for \texttt{OR} and falsity. In order to please the eye and the mind we often use the “sugared” forms \texttt{TRUE}, \texttt{FALSE} for (\texttt{AND}) and (\texttt{OR}) respectively.

Herbrand constants denote functions such that applications of two Herbrand constants denote the same entity if, and only if, the constants are the same; they both have the same number of operands and for each pair of corresponding operands the two expressions denote the same entity.

An occurrence of an expression $\alpha$ in an application $\beta$ of an Herbrand constant is an \textit{Herbrand occurrence} if $\alpha$ is an operand of $\beta$ or is a Herbrand occurrence in an operand of $\beta$.

\textit{Tuplings} are applications of the Herbrand constant $\bullet$. We can write the tupling ($\bullet \alpha_1 ... \alpha_n$) in a sugared form as $\langle \alpha_1 ... \alpha_n \rangle$. $\square$ is sugared writing for the empty tupling.

An application of a natural numeral $i$ to a tupling $\langle \alpha_1 ... \alpha_n \rangle \Gamma$ where $0 \leq i < n \Gamma$ denotes the same value as $\alpha_{i+1}$. \texttt{HEAD} and \texttt{TAIL} are sugared forms of 0 and 1 $\Gamma$ respectively.

\textit{Dotted pairs} are the binary special case of tuplings; i.e., applications of the form ($\bullet \alpha \beta$). There is a very useful convention from traditional LISP that a nest of dotted pairs

$$(\bullet \alpha_1 (\bullet \alpha_2 ... (\bullet \alpha_n \beta) ...) \Gamma)$$

where $n \geq 1 \Gamma$ can be sugared using square brackets as a “dotted listing” as follows:

$$[\alpha_1 \alpha_2 ... \alpha_n \bullet \beta]$$

with a very special case when $\beta$ is the empty listing $\square \Gamma$ namely the non-empty listing:

$$[\alpha_1 \alpha_2 ... \alpha_n \bullet \square] = [\alpha_1 \alpha_2 ... \alpha_n] .$$

In computations (such as Example 6 below) the listings are written unsugared to reveal the actual applicative structure on which the logic of the computation depends.

\textit{Set itemizations} are applications of the constant \texttt{SET}. The constant \texttt{SET} denotes a function such that (\texttt{SET} $\alpha_1 ... \alpha_k$) is the set consisting of the entities denoted by $\alpha_1 \Gamma ... \alpha_k \Gamma$. We can write a set itemization (\texttt{SET} $\alpha_1 ... \alpha_k$) in sugared form as $\{\alpha_1 \ldots \alpha_k\}$. A special case is the empty set itemization $\{\}$. \texttt{SET} is clearly not a Herbrand constant: (\texttt{SET} 1 2)$\Gamma$ (\texttt{SET} 1 1 2 1 2 2) and (\texttt{SET} 2 1) all denote the same entity.
The constant $\cup$ denotes a function such that an application $(\cup \alpha_1 \ldots \alpha_k)$ denotes the set which is the union of the sets denoted by $\alpha_1 \Gamma \ldots \alpha_k \Gamma$. The constants $\in$ and $\subseteq$ denote the usual set membership and subset relations.

- A **conditional** is an application of the constant $\text{IF}$ to three arguments. The constant $\text{IF}$ denotes a function such that $(\text{IF } \alpha \beta \gamma)$ is equal to $\beta$ if $\alpha$ is true and it is equal to $\gamma$ if $\alpha$ is false.

- A **conjunction** is an application of $\text{AND}$; a **disjunction** is an application of $\text{OR}$; a **union** is an application of $\cup$ and a **negation** is an application of $\text{NOT}$.

**Types, Equality and Inequality.** We can partition the denotable values into five **types**: numbers, truth values, sets, functions and other values. Two expressions denoting values of different types thus cannot denote the same value.

- Numerals and applications of the internal constants $+\Gamma -\Gamma \ast$ and $/\Gamma$ denote numbers.

- Existential quantifications and applications of the internal constants $\text{AND} \Gamma \text{OR} \Gamma \text{NOT}=\Gamma <\Gamma >\Gamma \in$ and $\subseteq\Gamma$ denote truth values.

- Set abstractions and applications of the internal constants $\text{SET}$ and $\cup\Gamma$ denote sets.

- Lambda abstractions and all internal constants except numerals denote functions.

- External constants and applications of Herbrand constants denote other values.

An equality $(\alpha = \beta)$ is true if $\alpha$ and $\beta$ denote the same entity and false if they do not.

**Obviousness.** The somewhat elusive notion of **obviousness** is partially formalized by the following sufficient conditions for concluding respectively that two expressions obviously do or obviously do not denote the same entity.

Two expressions $\alpha$ and $\beta$ **obviously denote the same entity** if one of the following holds:

- $\alpha$ and $\beta$ are identical atomic expressions;
• $\alpha$ and $\beta$ are set itemizations and for each operand $\gamma$ of the one there is an operand $\delta$ of the other such that $\gamma$ and $\delta$ obviously denote the same entity.

Two expressions $\alpha$ and $\beta$ obviously denote different entities if one of the following holds:

• $\alpha$ and $\beta$ are known to denote values of different types;

• $\alpha$ and $\beta$ are distinct constants;

• one of $\alpha$ and $\beta$ is a constant and the other is an application of an Herbrand constant;

• $\alpha$ and $\beta$ are applications of different Herbrand constants;

• $\alpha$ and $\beta$ are applications of the same Herbrand constants but with different number of operands;

• one of $\alpha$ and $\beta$ is (AND) and the other is (OR);

• $\alpha$ and $\beta$ are set itemizations and one of them has an operand $\gamma$ such that for all operands $\delta$ of the other $\Gamma \gamma$ and $\delta$ obviously denote different entities;

• one of $\alpha$ and $\beta$ has an Herbrand occurrence in the other.

**Computation.** There is a function $\text{simplify}$ which when applied to an expression yields an expression that is semantically equivalent to it (that is $\Gamma$ denotes the same entity). If $\text{(simplify } e\text{)}$ is exactly the same expression as $e\Gamma$ then $e$ is said to be in normal form.

A sequence of $n \geq 1$ expressions

$$e_1\Gamma \ldots \Gamma e_n\Gamma$$

which contains at most one expression in normal form and for which

$$e_{i+1} = (\text{simplify } e_i), \quad 1 \leq i < n,$$

is a VESPER computation. A computation which does not contain an expression in normal form is either finite and unfinished $\Gamma$ or infinite $\Gamma$ while a computation which contains an expression in normal form is necessarily finite $\Gamma$ and finished $\Gamma$ with that expression as its last component. Finished computations may be thought of as having ‘evaluated’ all the expressions in the sequence $\Gamma$ their common value being the entity denoted by each of them. The point of the computation is that its final expression denotes this
common value in as obvious a way as possible. The final expression in a
computation sequence is said to be the normal form of every expression in
the sequence. Expressions in infinite computation sequences have no normal
form.

Thus, the user's view of the system need only be through the function
normalform whose value for an expression \( e \) is the normal form of \( e \) if \( e \)
has one and which is undefined otherwise.

The function simplify has been designed with the intention of yielding
in this iterative manner normal forms which are indeed maximally perspic-
uous. It formalizes the notion of 'one computation step'. Its details contain
few surprises and the presentation given here is meant to appeal straightfor-
wardly to the reader's computational intuitions and to capture the processes
of traditional functional programming within applied lambda-calculi. What
may be novel about the present approach is the way in which the processes
of Horn clause logic programming are also captured within the calculus and
how easily these merge with and emerge from the lambda-calculus ideas.

**LOGLISP, SUPER and VESPER.** VESPER is the third in a series of
experiments aimed at exploring the problem of unifying the functional and
relational styles of logic programming.

The first experiment, LOGLISP [6], was a relatively crude attempt to link a
functional system with a relational system while keeping their separate iden-
tities intact. Although some useful applications were made of the resulting
hybrid, the conceptual foundation was unduly complex and unnatural.

Reflection on the shortcomings of the dualist LOGLISP approach provided
the basis for the second experiment, in which the unified SUPER system [5]
was formulated. The main idea of SUPER was to capture the logic common
to both functional and relational programming within a single set of rewrite
rules and then to make reduction to normal form the paradigm for all of
logic programming. The potential for parallel computation was noted but
not further pursued.

VESPER is a much more fully worked-out version of SUPER, with particular
emphasis placed on making completely explicit the logic of parallel reduc-
tion. Just as SUPER provided relational programming by exploiting Clark's
'completion' idea [2] so also in VESPER the user can define relations as lambda
abstractions whose bodies are disjunctions of existentially quantified conjunc-
tions. It will be recalled that Clark's idea was to take such definitions
as alternative characterizations of relations, obtaining them as translations
of corresponding sets of Horn clauses. However, there is no need to begin
with Horn clauses since one can easily and naturally express one's specifica-
tions of relations directly in this way using the lambda notation. Once
one has declared definitions of one or more relations in the VESPER style, one
may then evaluate ‘queries’ by formulating them as set-abstraction expressions (or in a different style as lambda-abstraction expressions) and then computing their normal forms.

The reader will easily see how further redex-patterns and corresponding small simplification transformations can be added to the VESPER logic by adding further cases to simplify in the obvious manner.

We were pleased to discover that essentially the same ideas have been independently developed by John Lloyd as indeed he describes in the present volume [3] and elsewhere [4]. Such a confluence of separate investigations tends to confirm one’s sense of the naturalness and inevitability of the present approach.

4 The functions simplify and subsimplify

Notation. The two functions simplify and subsimplify differ only in their behavior on abbreviations and non-redexes. The function subsimplify intuitively describes the lower-level simplifications which take place when simplification is extended vertically downwards into the substructure of an expression.

In writing the equations which define simplify and subsimplify we employ a few notational conventions.

- We use lowercase Greek letters with subscripts if needed to denote expressions.

- We use upper case Greek letters with subscripts if needed to denote (possibly empty) sequences of expressions. Concatenation of such sequences is denoted by juxtaposition. A sequence consisting of a single expression is considered to be the same as that expression.

- We write $S\alpha$ or $SS\alpha$ to denote the expression obtained by applying simplify or subsimplify respectively to the expression $\alpha$. To save writing whenever both of two equations of the form $S\alpha = \beta$ and $SS\alpha = \beta$ hold we write just the single equation: $B\alpha = \beta$.

- If $\Gamma$ is a (possibly empty) sequence of expressions then we write $S\Gamma$ or $SS\Gamma$ to denote the sequence obtained by applying simplify or subsimplify to each expression in $\Gamma$.

- By $(\equiv \alpha \beta)$ we denote either of the equations $(= \alpha \beta)\Gamma(= \beta \alpha)$.

The function simplify is then given by the following equations. Each equation is given a mnemonic label to facilitate later discussion.

**variable-or-constant:**

\[
B\alpha = \alpha
\]

if $\alpha$ is an atomic expression which is not an abbreviation.
abbreviation:
\[ S \alpha = \beta \Gamma \]
\[ SS \alpha = \alpha \Gamma \]
if \( \alpha \) is an occurrence of an abbreviation for which the definition \( \alpha =_{\text{def}} \beta \) has been declared.

member-true:
\[ B (\in \alpha \{\Gamma\}) = (\text{AND}) \Gamma \]
where \( \alpha \) obviously denotes the same entity as some expression in \( \Gamma \).

member-false:
\[ B (\in \alpha \{\Gamma\}) = (\text{OR}) \Gamma \]
where \( \alpha \) obviously does not denote the same entity as any expression in \( \Gamma \).

subset-true:
\[ B (\subseteq \{\Gamma\} \{\Delta\}) = (\text{AND}) \Gamma \]
where each expression of \( \Gamma \) obviously denotes the same entity as some expression in \( \Delta \).

subset-false:
\[ B (\subseteq \{\Gamma\} \{\Delta\}) = (\text{OR}) \Gamma \]
where some expression in \( \Gamma \) obviously does not denote the same entity as any expression in \( \Delta \).

equate-obviously-same:
\[ B (= \alpha \beta) = (\text{AND}) \Gamma \]
where \( \alpha \) and \( \beta \) obviously denote the same entity.

equate-obviously-different:
\[ B (= \alpha \beta) = (\text{OR}) \Gamma \]
where \( \alpha \) and \( \beta \) obviously denote different entities.

or-true:
\[ B (\text{OR} \Gamma (\text{AND}) \Delta) = (\text{AND}) \]

not-not:
\[ B (\text{NOT} (\text{NOT} \alpha)) = SS \alpha. \]

unit-and:
\[ B (\text{AND} \alpha) = SS \alpha. \]

unit-or:
\[ B (\text{OR} \alpha) = SS \alpha. \]

unit-union:
\[ B (\cup \alpha) = SS \alpha. \]
if-true:
\[ B (\text{IF} \ (\text{AND}) \ \beta \ \gamma) = SS \beta. \]

if-false:
\[ B (\text{IF} \ (\text{OR}) \ \beta \ \gamma) = SS \gamma. \]

if-neither:
\[ B (\text{IF} \ \alpha \ \beta \ \gamma) = (\text{IF} \ SS \ \alpha \ \beta \ \gamma) \Gamma \]
where \( \alpha \) is neither (AND) nor (OR). Note that the tactic here is ‘lazy’ that is, simplification of neither the true arm nor the false arm of a conditional expression is begun even if the expression is on the surface until its boolean part has been normalized to either (AND) or (OR) and then only the relevant arm is selected. It would certainly be logically acceptable to allow ‘eager’ or ‘speculative’ simplification of both true- and false-arms to take place in parallel with that of the boolean part—a tactic which however may result in excessive consumption of resources.

numeric-redex:
\[ B (\delta \ \mu_1 \ \ldots \ \mu_k) = \mu \Gamma \]
where
- \( \delta \) is either \( +\Gamma \cdot \Gamma \cdot \Gamma \cdot \Gamma < \) or \( > \);
- \( \mu_1 \ldots \mu_k \) are numerals;
- and \( \mu \) is the numeral or truth value which denotes the result of applying the corresponding operation to the corresponding numbers.

and-and:
\[ B (\text{AND} \ \Gamma_1 \ (\text{AND} \ \Delta_1) \ \ldots \ \Gamma_n \ (\text{AND} \ \Delta_n) \ \Gamma_{n+1}) = \]
\[ (\text{AND} \ SS \Gamma_1 \ SS \Delta_1 \ \ldots \ SS \Gamma_n \ SS \Delta_n \ SS \Gamma_{n+1}) \Gamma \]
where the sequence \( \Gamma_1 \ldots \Gamma_n \ \Gamma_{n+1} \) does not contain the expression (OR) \( \Gamma \) or any conjunctions.

or-or:
\[ B (\text{OR} \ \Gamma_1 \ (\text{OR} \ \Delta_1) \ \ldots \ \Gamma_n \ (\text{OR} \ \Delta_n) \ \Gamma_{n+1}) = \]
\[ (\text{OR} \ SS \Gamma_1 \ SS \Delta_1 \ \ldots \ SS \Gamma_n \ SS \Delta_n \ SS \Gamma_{n+1}) \Gamma \]
where the sequence \( \Gamma_1 \ldots \Gamma_n \ \Gamma_{n+1} \) does not contain the expression (AND) \( \Gamma \) or any disjunctions.

union-union:
\[ S (\cup \ \Gamma_1 \ (\cup \ \Delta_1) \ \ldots \ \Gamma_n \ (\cup \ \Delta_n) \ \Gamma_{n+1}) = \]
\[ (\cup \ SS \Gamma_1 \ SS \Delta_1 \ \ldots \ SS \Gamma_n \ SS \Delta_n \ SS \Gamma_{n+1}) \Gamma \]
where the sequence \( \Gamma_1 \ldots \Gamma_n \ \Gamma_{n+1} \) contains no set itemizations and no unions.
exists-exists:
\[ \mathcal{B}(\exists (\Phi) (\exists (\Psi) \beta)) = (\exists (\Phi \Psi) SS \beta). \]

exists-vacuous:
\[ \mathcal{B}(\exists (\Phi) \beta) = SS \beta \Gamma \]
where \( \Phi \) is empty or none of the variables in \( \Phi \) are free in \( \beta \).

and-exists:
\[ \mathcal{B}(\text{AND } \Gamma_1 (\exists (\Phi_1) \beta_1) ... \Gamma_n (\exists (\Phi_n) \beta_n) \Gamma_{n+1}) = \\
(\exists (\Psi_1 ... \Psi_n) (\text{AND } SS \Gamma_1 SS \gamma_1 ... SS \Gamma_n SS \gamma_n SS \Gamma_{n+1})) \Gamma \]
where \( \exists (\Psi_i) \gamma_i \) is a variant of \( (\exists (\Phi_i) \beta_i) \Gamma \leq i \leq n \Gamma \) and the sequence \( \Gamma_1 ... \Gamma_n \Gamma_{n+1} \) contains no conjunctions or disjunctions.

and-or:
\[ \mathcal{B}(\text{AND } \Gamma_1 (\text{OR } \Delta_1) ... \Gamma_n (\text{OR } \Delta_n) \Gamma_{n+1}) = \\
(\text{OR } (\text{AND } SS \Pi_1) ... (\text{AND } SS \Pi_r) \Gamma) \]
where \( \Pi_1 \Gamma ... \Pi_r \) are all of the different sequences of the form \( \Gamma \delta_1 ... \Gamma \delta_n \Gamma \in \Gamma \) in which \( \delta_i \) is an expression in the sequence \( \Delta_i \Gamma \)
\( \leq i \leq n \Gamma \) and the sequence \( \Gamma_1 ... \Gamma_n \Gamma_{n+1} \) contains no conjunctions or disjunctions.

union-sets:
\[ \mathcal{B}(\cup \Gamma_1 \{\Delta_1\} ... \Gamma_n \{\Delta_n\} \Gamma_{n+1}) = \\
(\cup \{SS \Delta_1 ... SS \Delta_n\} SS \Gamma_1 ... SS \Gamma_n SS \Gamma_{n+1}) \Gamma \]
where the sequence \( \Gamma_1 ... \Gamma_n \Gamma_{n+1} \) contains no set itemizations.

solved-setof:
\[ \mathcal{B}\{<x_1 ... x_n> | (\text{AND } (\equiv x_1 \alpha_1) ... (\equiv x_n \alpha_n))\} = \\
\{<SS \alpha_1 ... SS \alpha_n>\}. \]

setof-or:
\[ \mathcal{B}\{V | (\text{OR } \alpha_1 ... \alpha_n)\} = (\cup \{V | SS \alpha_1\} ... \{V | SS \alpha_n\}). \]

thin-set-itemization:
\[ \mathcal{B}\{\Gamma\} = SS \{\Gamma'\} \Gamma \]
where \( \Gamma \) contains two expressions \( \alpha \Gamma \beta \) that obviously denote the same entity and \( \Gamma' \) is \( \Gamma \) with one of \( \alpha \Gamma \beta \) removed.

selection:
\[ \mathcal{B}\{i | \alpha_1 ... \alpha_n\} = SS \alpha_{i+1} \Gamma \]
where \( i \) is a natural numeral \( 0 \leq i < n \).

equate-Herbrand-applications:
\[ \mathcal{B}(= (\alpha_0 \alpha_1 ... \alpha_n) (\beta_0 \beta_1 ... \beta_n)) = \\
(\text{AND } (= SS \alpha_1 SS \beta_1) ... (= SS \alpha_n SS \beta_n)) \Gamma \]
where \( \alpha_0 \) and \( \beta_0 \) are the same Herbrand constant.
exists-or:
\[ B (\exists (\Phi) (\text{OR } \alpha_1 \ldots \alpha_n)) = \]
\[ (\text{OR } (\exists (\Phi) SS \alpha_1) \ldots (\exists (\Phi) SS \alpha_n)). \]

exists-and-equation:
\[ B (\exists (\Phi) (\text{AND } \Gamma (\equiv x \alpha) \Delta)) = (\exists (\Psi) (\text{AND } SS \Gamma SS \Delta \theta) \Gamma) \]
where
- \( x \) is a variable in the sequence \( \Phi \);
- \( x \) does not occur in \( \alpha \);
- \( \Gamma \) does not contain an equation \( (\equiv y \beta) \) where \( y \) is a variable in the sequence \( \Phi \);
- \( \theta = \{\alpha/x\} \);
- and \( \Psi \) is \( \Phi \) with \( x \) omitted.

beta-redex:
\[ B ((\lambda (x_1 \ldots x_n) \beta) \alpha_1 \ldots \alpha_n) = \beta'\{SS \alpha_1/x_1, \ldots, SS \alpha_n/x_n\} \Gamma \]
where \( \beta' = SS \beta \).

An expression to which none of the preceding equations applies may nevertheless be simplifiable but only at lower [syntactic] levels (‘interior’) and not at the top [syntactic] level (‘surface’). The following equations cover the possible cases that arise. The principle followed is the same in all cases: namely the simplification or subsimplification is merely applied to the immediate subexpressions:

simplify-interior:
\[ S (\Gamma) = (SG) \text{ if } (\Gamma) \text{ is an application.} \]
\[ SS (\Gamma) = (SSG) \text{ if } (\Gamma) \text{ is an application.} \]
\[ S (\exists V \beta) = (\exists V S \beta). \]
\[ SS (\exists V \beta) = (\exists V SS \beta). \]
\[ S (\lambda V \beta) = (\lambda V S \beta). \]
\[ SS (\lambda V \beta) = (\lambda V SS \beta). \]
\[ S \{V \mid \beta\} = \{V \mid S \beta\}. \]
\[ SS \{V \mid \beta\} = \{V \mid SS \beta\}. \]

For example, the application
\[ (+ (* 3 4) (* 4 5)) \]
is not simplifiable at the top level and so we have:
\[ S (+ (* 3 4) (* 4 5)) = (S + S (* 3 4) S (* 4 5)) = (+ 12 20). \]
5 Parallel expression rewriting

It is helpful as a general (but not strictly truthful) description of VESPER’s simplification function to say that it gives the result of the intuitive process of simultaneously rewriting all subexpressions of an expression. In fact, it gives the result of simultaneously rewriting almost all of the subexpressions. The ones which are not rewritten fall into two categories.

The most obvious departure from complete simultaneous rewriting concerns the replacement of an abbreviation by its definitions. In VESPER the replacement occurs only if the abbreviation is not a proper subexpression of an expression which is simplifiable at the top level.

The reason for this is to avoid the otherwise computationally intolerable situation in which a recursive definition (or a set of two or more mutually recursive definitions) would be "unfolded" at each simplification step. This would result in a rapid rate of growth in the size of successive expressions in the computation.

Another departure from complete rewriting (already remarked on) involves the true and false alternatives of a conditional. Conditionals are simplified 'lazily'. Only when their boolean guards have been reduced to (AND) or (OR) is one (and then only one) of their alternatives exposed to the simplification process. Both alternatives of a conditional and all of their subexpressions are thus left untouched until the conditional is decided.

The reason for this is to avoid committing one’s finite resources "eagerly" to the simplification of alternatives which are destined not to be selected. The price paid for this cautious strategy is of course that the alternatives which will be selected are made to wait for their simplification until their conditionals are decided. The more that time is of the essence the more worth while it might be to risk running out of resources by getting a running start on simplifying the alternatives concurrently with the process of reducing the guard to normal form. In some fortunate cases indeed the selected alternative will already be reduced to normal form by the time the conditional is decided. The "lazy" strategy adopted in VESPER guarantees that this will never happen except in cases where the selected alternative is in normal form to begin with.

6 Further examples

Example 1. If \( p \) is a variable the simplification of the expression

\[
(\text{AND} \ (\text{OR} \ \text{FALSE} \ p) \ (> \ 5 \ 7) \ \text{TRUE} \ (= \ 4 \ 6) \ \text{TRUE})
\]
is found by applying these equations recursively. In order to make the details clearer we desugar `TRUE` and `FALSE` to `(AND)` and `(OR)` respectively:

\[
S (\text{AND} (\text{OR} (\text{OR}) \text{p}) (> 5 7) (\text{AND}) (= 4 6) (\text{AND})) \\
= (\text{AND} S (\text{OR} (\text{OR}) \text{p}) S (> 5 7) S (= 4 6)) \text{ and-and} \\
= (\text{AND} (\text{OR} S \text{p}) (\text{OR}) (\text{OR})) \text{ or-or,} \\
= (\text{AND} (\text{OR} \text{p}) (\text{OR}) (\text{OR})) \text{ atom.}
\]

**Example 2.** If `a b c d e` and `f` are variables then

\[
S (\text{AND} \text{a} (\text{OR} \text{b} \text{c} \text{d} \text{e} (\text{OR} (> 5 7) (< 3 4)) \text{f}) \\
= (\text{OR} (\text{AND} \text{a} \text{b} \text{d} \text{e} S (> 5 7) \text{f}) \text{ and-or} \\
(\text{AND} \text{a} \text{b} \text{d} \text{e} S (< 3 4) \text{f}) \\
(\text{AND} \text{a} \text{c} \text{d} \text{e} S (> 5 7) \text{f}) \\
(\text{AND} \text{a} \text{c} \text{d} \text{e} S (< 3 4) \text{f})) \\
= (\text{OR} (\text{AND} \text{a} \text{b} \text{d} \text{e} (\text{OR}) \text{f}) \text{ numeric-redex} \times 4 \\
(\text{AND} \text{a} \text{b} \text{d} \text{e} (\text{AND}) \text{f}) \\
(\text{AND} \text{a} \text{c} \text{d} \text{e} (\text{OR}) \text{f}) \\
(\text{AND} \text{a} \text{c} \text{d} \text{e} (\text{AND}) \text{f})).
\]

**Example 3.** The following expression fits the `exists-and-equation` case. So we have:

\[
S (\exists (x y z) (\text{AND} (A x y z) (= x 4) (B x y z) (= y (C x)))) \\
= (\exists (y z) (\text{AND} A (4 y z) S (B 4 y z) S (= y (C 4)))) \\
= (\exists (y z) (\text{AND} (SA S 4 S y S z) \\
(SB S 4 S y S z) \\
(S = S y S (C 4)))) \\
= (\exists (y z) (\text{AND} A (4 y z) (B 4 y z) (= y (C 4)))) \\
= (\exists (y z) (\text{AND} A (4 y z) (B 4 y z) (= y (C 4))))
\]

This expression again fits the `exists-and-equation` case. So another round of simplification can begin:

\[
S (\exists (y z) (\text{AND} (A 4 y z) (B 4 y z) (= y (C 4)))) \\
= (\exists (z) (\text{AND} (A (4 (C 4) z) S (B 4 (C 4) z))) \\
= (\exists (z) (\text{AND} (SA S 4 S (C 4) S z) (SB S 4 S (C 4) S z))) \\
= (\exists (z) (\text{AND} (A (4 (S C S 4) z) (B 4 (S C S 4) z))) \\
= (\exists (z) (\text{AND} (A (4 (C 4) z) (B 4 (C 4) z))))
\]

yielding an expression in normal form.

The `exists-and-equations` case is one of three simplification cases which cooperate with each other so to speak to perform unification computations. The other two are `equate-Herbrand-applications` and `equate-obviously-different`.
Example 4. Unification.

If \( P, G, H \) and \( K \) have been declared by the user to be external constants and at least \( P \) has been declared to be a Herbrand constant, then the expression

\[
(\exists (x \ y \ z \ u \ v \ w \ r \ s \ a \ b \ t)
\AND (\begin{align*}
&= (P \ x \ y \ u) \ (P \ (G \ r \ s) \ r \ s) \\
&= (P \ y \ z \ v) \ (P \ a \ (H \ a \ b) \ b) \\
&= (P \ x \ v \ w) \ (P \ (G \ r \ s) \ r \ s) \\
&= (P \ u \ z \ w) \ (P \ (K \ t) \ t \ (K \ t))
\end{align*})
\]

\(x\)-becomes \(x\)
\(y\)-becomes \(y\)
\(z\)-becomes \(z\)
\(u\)-becomes \(u\)
\(v\)-becomes \(v\)
\(w\)-becomes \(w\)
\(r\)-becomes \(r\)
\(s\)-becomes \(s\)
\(a\)-becomes \(a\)
\(b\)-becomes \(b\)
\(t\)-becomes \(t\))

asserts in effect that there is a substitution for the bound variables which will simultaneously unify the left- and right-sides of each of the four equations. The normal form of the sentence is obtained by iteration of the \textbf{exists-and-equation} case:

\[
(\exists (v) \ (\AND (x\text{-becomes} \ (G \ v \ (K \ (H \ v \ v))))
\AND (y\text{-becomes} v)
\AND (z\text{-becomes} \ (H \ v \ v))
\AND (u\text{-becomes} \ (K \ (H \ v \ v)))
\AND (v\text{-becomes} v)
\AND (w\text{-becomes} \ (K \ (H \ v \ v)))
\AND (r\text{-becomes} v)
\AND (s\text{-becomes} \ (K \ (H \ v \ v)))
\AND (b\text{-becomes} v)
\AND (t\text{-becomes} \ (H \ v \ v))))
\]

When rewritten in parallel, this computation of unification is similar to a parallel unification algorithm based on rewriting of equalities \cite{1} and it might be possible to incorporate other parts of that algorithm as well.

Example 5. The Fibonacci function in a functional style

The classic Fibonacci program serves to illustrate programming in the functional style using a conditional expression:
\[ FIB =_{def} (\lambda (n) (FIB-ITER 1 1 n)) \]

\[ FIB-ITER =_{def} (\lambda (a b n) (IF (= n 1) a (FIB-ITER b (+ a b) (- n 1)))) \]

The rewriting of an application \( (FIB \ n) \) proceeds just as in an ordinary functional programming language.

**Example 6. A logic programming query.**

We now use the definition of \( APPEND \) given earlier to work through an illustration of how a simple logic programming query is handled in this logic. We wish to compute the set of all pairs \( <x \ y> \) such that \( (APPEND x \ y \ (\bullet \ 1 \ (\bullet \ 2 \ []))) \) is true. This set is denoted by the set-abstraction expression:

\[ \{<x \ y> \mid (APPEND x \ y \ (\bullet \ 1 \ (\bullet \ 2 \ [])))\} \]

and therefore we normalize this expression. Its normal form is:

\[ \{<\square \ (\bullet \ 1 \ (\bullet \ 2 \ []))> <(\bullet \ 1 \ [] \ (\bullet \ 2 \ []))> <(\bullet \ 1 \ (\bullet \ 2 \ [])) \square>\} \]

a set itemization expression giving all such pairs explicitly. The first few expressions in the computation are:

1. The original expression:

\[ \{<x \ y> \mid (APPEND x \ y \ (\bullet \ 1 \ (\bullet \ 2 \ [])))\} \]

2. The first simplification is to replace the abbreviation \( APPEND \) by its definiens yielding:

\[ \{<x \ y> \mid ((\lambda (a \ b \ c)
(\text{OR} \ (\exists (x) \ (\text{AND} (= a \ [])
\quad (= b \ x)
\quad (= c \ x)))

\quad (\exists (x \ v \ w) \ (\text{AND} (= a \ (\bullet \ x \ y))
\quad (= b \ v)
\quad (= c \ (\bullet \ x \ w)))

\quad (APPEND y \ v \ w))))

\quad x \ y \ (\bullet \ 1 \ (\bullet \ 2 \ []))))\} \]

3. Next the beta-redex is replaced and further within it there are two cases of exist-and-equation resulting in quantifier eliminations.
\{x\ y\ | \ (\text{OR} \ (\text{AND} \ (= \ x \ [\ ])) \ (= \ y \ (\bullet \ 1 \ (\bullet \ 2 \ [])))
\ (\exists \ (x0 \ x1 \ x3) \ (\text{AND} \ (= \ x \ (\bullet \ x0 \ x1))
\ (= \ (\bullet \ 1 \ (\bullet \ 2 \ [])) \ x3)
\ (\bullet \ x0 \ x3))
\ (\text{APPEND} \ x1 \ y \ x3))))\} \\

4. This expression is a case of setof-or with an internal equate-Herbrand-applicationsΓ \\
\{x\ y\ | \ (\text{AND} \ (= \ x \ [\ ])) \ (= \ y \ (\bullet \ 1 \ (\bullet \ 2 \ [])))\}
\{x\ y\ | \ (\exists \ (x0 \ x1 \ x3)
\ (\text{AND} \ (= \ x \ (\bullet \ x0 \ x1))
\ (= \ 1 \ x0)
\ (= \ (\bullet \ 2 \ [])) \ x3)
\ (\text{APPEND} \ x1 \ y \ x3))))\} \\

5. The first component of the union is a solved-setoff and within the second there is an and-and. \\
\{x\ y\ | \ (\exists \ (x0 \ x1 \ x3)
\ (\text{AND} \ (= \ x \ (\bullet \ x0 \ x1))
\ (= \ 1 \ x0)
\ (= \ (\bullet \ 2 \ [])) \ x3)
\ (\text{APPEND} \ x1 \ y \ x3))))\} \\

6. An exists-and-equation within the second component of the union. \\
\{x\ y\ | \ (\exists \ (x0 \ x1)
\ (\text{AND} \ (= \ x \ (\bullet \ x0 \ x1))
\ (= \ 1 \ x0)
\ (\text{APPEND} \ x1 \ y \ (\bullet \ 2 \ []))))\} \\

7. And another. \\
\{x\ y\ | \ (\exists \ (x1) \ (\text{AND} \ (= \ x \ (\bullet \ 1 \ x1))
\ (\text{APPEND} \ x1 \ y \ (\bullet \ 2 \ []))))\} \\

8. Now the abbreviation APPEND is an outermost redexΓ so it is eligible for unfolding.
By now the first solution has appeared and the development continues in a similar manner for a total of 23 simplification steps.

7 Conclusions and future work

The vertically extended simplification approach described here provides a simple, sound and intuitive basis for a massively parallel computational logic which formalizes the merging of the functional and relational programming styles.

Except for (1) the ‘lazy’ treatment of conditionals and (2) the ‘surface only’ treatment of abbreviations, VESPER exploits every logically permissible opportunity for parallel computation. In this sense it comes close to the maximum possible degree of parallelism. Neither (1) nor (2) is logically necessary but only pragmatically so and there is plenty of scope for further investigation as to how much ‘speculative’ eagerness can be tolerated in relaxing (1) and how much recursive unfolding of definitions might be allowed in a more permissive version of (2).

While this discussion has been largely confined to logical and linguistic issues, the implementation issues are extremely interesting and deserve a separate investigation. As was mentioned briefly, our present view is that the VESPER logic would most naturally be embodied in the collective behaviour of a large body of small identical processors each playing the role of a node in a graphical representation of expressions and each capable of recognizing

\[
(\cup \{ \{1 (\bullet 2)\}\})
\]

\[
\left\{ x, y \mid (\exists (x1)
\quad (\text{AND} \, (= x (\bullet 1 x1))
\quad ((\lambda (a \, b \, c)
\quad (\text{OR} \, (\exists (x) (\text{AND} \, (= a []))
\quad (= b \, x))
\quad (= c \, x))))
\quad (\exists (x \, y \, v \, w)
\quad (\text{AND} \, (= a (\bullet x y))
\quad (= b \, v))
\quad (= c (\bullet x w))
\quad (\text{APPEND} \, y \, v \, w))))
\quad x1 \, y \, (\bullet 2 [])))\}
\]

The number of processors required to represent an expression is approximately the same as the number of nodes in its parse tree. If suitable advantage is taken of ‘sharing’ (for example, the several occurrences of a bound variable can be pointers to a single processor representing that variable) the total number of processors need not be as large as this. A VESPER machine with, say, a million processors would be quite useful.
and acting on the appropriate case of simplification corresponding to its current state.

A preliminary look at some of the problems of designing such an implementation has persuaded us that the next step in the VESPER project should be to study this cellular automaton model in detail.

Meanwhile, to run the examples given in this paper and for general exploration purposes, we have written (in Common Lisp) a usefully fast implementation of the functions simplify, subsimplify, and normalform.

References


