Sal/Svm: An Assembly Language and Virtual Machine for Computing with Non-Enumerated Sets

Phillip Stanley-Marbell
IBM Research—Zürich
Säumerstrasse 4, 8803 Rüschlikon, Switzerland

Abstract

Presented is the design, implementation and evaluation of a system for computing with *non-enumerative set representations*. The implementation is in the form of a *set assembly language (Sal)* whose operations correspond to an implementation of the algebra of sets, with minimal added syntactic sugar; a compiler *(Salc)* for validation and static optimization of Sal definitions; and a *virtual machine architecture (Svm)* for executing Sal definitions.

Sal/Svm has turned out to be a surprisingly versatile framework for a growing number of problems. One such application, as a framework for declaratively specifying *computational problems* with the same level of precision that traditional machine languages enable the specification of *computational algorithms*, is presented.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Macro and assembly languages, Nonprocedural languages, Specialized application languages*; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

General Terms

Algorithms, Languages

Keywords

Virtual Machines, Set Theory, Declarative Problem Specification, Languages for Future Device Technologies

1 Introduction

Sets play an important role in many areas of computing systems. Their uses range from application in symbol tables for compilers, to representing preferred system set-points in runtime systems; from representing system configurations in analog and digital design automation, to the representation and solution of multi-objective optimization problems [18]. Across these varied application domains, set representations may be either *enumerative*—with all members encoded in the set's representation—or *non-enumerative*. Non-enumerative sets represent their elements with a *method* for identifying the set's members, rather than with a *representation* (e.g., a list, tree, or encoded representation thereof), of the collection of set members.

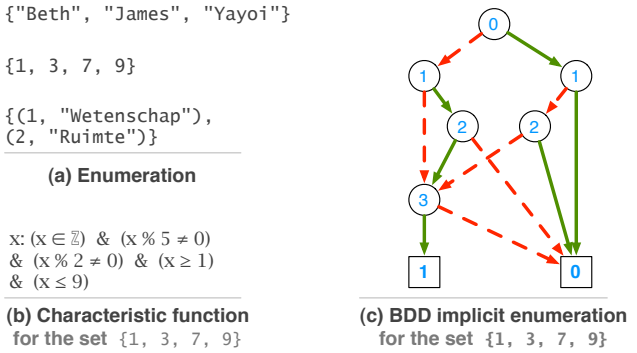


Figure 1. (a) explicit enumerative, (b) non-enumerative and (c) implicit enumerative representations of sets.

Most examples of sets in the computing systems milieu are enumerative set representations—they list members of a set, as illustrated in Figure 1(a). Non-enumerative set representations on the other hand have fewer realizations in the literature and in practice. This is partly due to the fact that computation has traditionally been a more expensive resource than storage; it has therefore typically been a better tradeoff to *store* the elements of a list, rather than *compute* them when needed. Recent technology trends are however leading to a shift in performance bottlenecks from computation to memory accesses. As accessing a single word in main memory is now easily several orders of magnitude more costly than executing most arithmetic operations, it may be more efficient, in some instances, to *compute* the elements of a set, than to retrieve those same elements from memory.

Enumerative representations of sets may be further classified into *implicit enumerations* versus *explicit enumerations*. An example of an explicit enumeration of a set is a list of elements, while an implicitly enumerated set would be a representation of the same list of elements, using, say, a compact representation such as a *binary decision diagram (BDD)*. The qualifier "implicit" captures the fact that the elements of the set cannot be seen directly by inspection of the representation, even though the representation "stores" the elements, and thus grows with increasing set size. Figure 1(c) shows the BDD representation of the set of integers {1, 3, 7, 9}, the individual members of which can be represented with 4-bit integers. In the BDD of Figure 1(c), the circled nodes labeled n are associated with properties of the n'th least-significant bit of the set member.

VMIL'10, 17-OCT-2010, Reno, USA

A dashed edge along a path denotes a 0 decision, and a solid edge a 1. The sequence of decisions along a path from the root, ⓪, to the terminal [1], indicate a bit vector that belongs to the set represented by the BDD. Edges that skip one or more levels in the BDD indicate the bit positions in question can take on any value. Thus, for example, the leftmost path in the figure corresponds to the bit pattern 00X1, where X here denotes a logical "don't care". The 4-bit vectors along the paths from the root to the terminal [1], i.e., $00X1_2 = \{1_{10}, 3_{10}\}$, $0111_2 = 7_{10}$, and $1001_2 = 9_{10}$, are an *implicit enumeration* of all members of the set. As more items need to be added to the set, the representation grows in size (even if only at 2^n boundaries for a set that can be mapped into n bits).

A *non-enumerative* representation of a set, on the other hand, does not *represent* the members of the set, and will have a size independent of the number of elements in the set. It instead embodies some property that holds only for members of the set (Figure 1(b)). One example of the representation of such properties is as *characteristic functions* (Boolean predicates or *set comprehensions*). Non-enumerative set representations are of particular interest when set membership is defined by a property of individual members, or relations between tuples of members. Such representations of sets may either occur due to constructive properties of a set to be represented (e.g., "the set of odd positive integers less than 100"), or may result from derivative properties deduced via, e.g., regression analysis. An example of the latter is a representation of a set of measurements of a system which has been curve fit to an analytic function of the system's parameters.

Both enumerative and non-enumerative set representations may be handled *symbolically*, e.g., by Boolean algebra operations in the case of BDD representations, and by symbolic algebraic manipulation operations in the case of characteristic functions.

1.1 Contributions and outline

This paper presents the design, implementation and preliminary evaluation of a system for constructing and manipulating *non-enumerative set representations*. The implementation is in the form of a *set-algebra representation assembly language (Sal)*, a *virtual machine (Svm)* for executing Sal definitions, and a *compiler for Sal (Salc)*, which validates, optimizes and generates an intermediate executable form of Sal definitions, for execution on Svm.

Section 1.2 outlines the terminology used in the remainder of the article. Section 2 presents a description of the Sal language and the virtual machine, Svm, for its execution. Section 3 details the Svm internal architecture, by providing an overview of the internal representations used to facilitate Svm's implementation of the Sal semantics. Section 4 briefly outlines the properties of the current implementation of the compiler and runtime system, and presents a preliminary performance evaluation.

One of the applications for which Sal is currently being used, is in exploring execution platforms for future device technologies (e.g., technologies that will eventually replace CMOS). Section 5 presents an application in this context, using Sal as a *problem specification language* to facilitate energy-efficient computation by taking advantage of particular device characteristics. Related research is presented in Section 6, and the article is concluded in Section 7 with a summary and a discussion of ongoing and planned research directions.

1.2 Terminology and definitions

The following terminology is used in the remainder of this article:

- **Universe**, U_x: a collection of possible basis values that may be taken on by the elements of a set. Universes may be *ordered* or *unordered*; ordered universes will be denoted with $\langle \ldots \rangle$, and unordered universes with $\{\ldots\}$.
- **Dimension**, $U_x[i]$: When a product universe of multiple one-dimensional universes is formed, each constituent sub-universe in the multi-dimensional product space U_x is referred to as a *dimension*. Each k-th sub-dimension of an n-dimensional universe is notated as $U_x[k]$, $1 \leq k \leq n$. Thus, in the following, the term *dimension* will sometimes be used interchangeably with the term *universe*.
- **Predicate**, P_x: a Boolean predicate on one dimension.
- **Predicate Tree**, T_n: a tree of predicates, each of which may act on a separate dimension.
- **Set**, S_n^x: A *set* is a particular collection of instance elements drawn from a single or multi-dimensional universe. It is represented by a pairing of a predicate tree T_n with a universe U_x; the superscript will be omitted when the set's universe is either obvious or irrelevant.
- **Boolean values** will be represented with words in small capitals—TRUE and FALSE—and **operators (Boolean and arithmetic)** in small capital boldface, e.g., **AND** (logical AND), **POW** (exponentiation).
- **Equality** in expressions will be denoted with ==, and assignment with =.

Predicate trees may contain *parameters*, which may be either *bound* or *free*, defined as follows:

DEFINITION 1 (BOUND PARAMETERS IN A PREDICATE TREE). *A parameter p is said to be bound in a predicate tree T_n, if the meaning of T_n is unchanged by the uniform replacement of p by another variable q, not occurring in T_n. Each bound variable is bound by a quantifier in the closest enclosing scope. Bound variables have a type, embodied in one of the sub-dimensions of the predicate tree in which they reside.*

DEFINITION 2 (FREE PARAMETERS IN A PREDICATE TREE). *A parameter p is said to be free in a predicate tree T_n, if the meaning of T_n is unchanged by (possibly non-uniform) replacement of any occurrence of p by another variable q. Free variables have a type, embodied in one of the sub-dimensions of the predicate tree in which they reside*[1].

2 Sal language and Svm virtual machine

Sal is an assembly language for describing computations on non-enumerative sets (as defined in the preceding section). It is termed a *set assembly language* since it provides only the basic low-level primitives essential to define computations of interest, and does not provide the variety of constructs typically found in a high-level programming language; the complete Sal grammar in EBNF form is listed in Appendix A. The execution model for Sal is the *Set virtual machine (Svm)*, described in more detail in Section 3.

Sal is not used to represent programs in the traditional sense—i.e., it does not describe a series of computations. Instead, Sal is best thought of as a language for representing *computational problems* using set-theoretic constructs. These problem representations are formulated in terms of operations on the state represented in the Svm virtual machine. Sal is thus in principle a form of declarative assembly language. Unlike traditional declarative higher-level languages, which can be used to express algorithms, and, e.g., achieve computation as a side effect of goal-directed evaluation, Sal is purposefully restricted to problem definitions. This separation of problem definition from particular solution of problems via algorithms is key to enabling the applications described in Section 5.

[1] Note the difference of this definition from definition of *free*

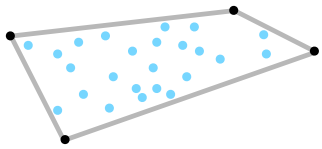


Figure 2. A set of 28 points in the plane; the four dark points define the convex hull of the set.

```
1   --
2   --   The sign of the determinant
3   --
4   --          | 1 px py |
5   --    D  =  | 1 qx qy |  = (qx*ry - qy*rx)
6   --          | 1 rx ry |    - px(ry - qy) + py(rx-qx),
7   --
8   --   denotes whether r is on left or right of line pq.
9   --
10  U0       :  Integers = <1 ... 10 delta 2*iota>
11  U1       :  Integers = <1 ... 10 delta (2*iota)+1>
12  U2       =  U0 >< U1
13
14  P10      = !((qy == py:U2[2]) & (qx == px:U2[1]))
15  P11      = ((qx*ry - qy*rx) - px:U2[1]*(ry - qy) +
16               py:U2[2]*(rx - qx)) >= 0
17  P1       = exists qx:U2[1] exists qy:U2[2]
18             forall rx:U2[1] forall ry:U2[2] (P10 & P11)
19
20  --      S1 is the input set
21  S1       = (true : U2)
22
23  --      S2 is the convex hull of the input set
24  S2       = (P1   : U2)
25
26  echo    "S2 = " print enum S2
```

Figure 3. Sal assembler representing the subset of a set of points that denotes the latter's convex hull.

2.1 Sal by example: Convex Hull

To illustrate the nature and notation of Sal problem definitions, the computational problem of the *convex hull* will be used here and in the remainder of the paper. Computing the convex hull of a set of points in the plane is an important problem with applications in a variety of domains, ranging from image processing to VLSI design.

Informally, if the points in a plane are represented by pins stuck in, say, a board (Figure 2), the convex hull of the set is the subset of pins which, if one were to place an elastic band around the grouping of pins, would be touching the elastic band. Formally, the convex hull can be defined as follows:

DEFINITION 3 (CONVEX HULL). *The convex hull,* **CH**(**S**)*, of a set* **S** *of points in the plane, is the smallest convex polygon for which each point in* **S** *is either on the boundary thereof, or in its interior* [7].

The Sal assembler specification of the convex hull problem is presented in Figure 3. Comments are introduced by `--`, and extend to the end of a line. Statements in Sal primarily involve operations on, and between, registers for holding universe, predicate, and set expressions. Universe registers, predicate registers and set registers, are denoted by the letters `U`, `P` and `S` (respectively), followed by a positive integer. In what follows, the contents of a particular register, e.g., `P5`, will be interchangeably referred to as "predicate `P5`", rather than the more accurate but cumbersome "the predicate held in register `P5`", for simplicity of exposition.

Lines 10 and 11 in Figure 3 define two basis universes of scalar integer type in the universe registers `U0` and `U1`, assigning to them a range of values. Line 12 defines a new universe, whose implied type

variables in other contexts [1].

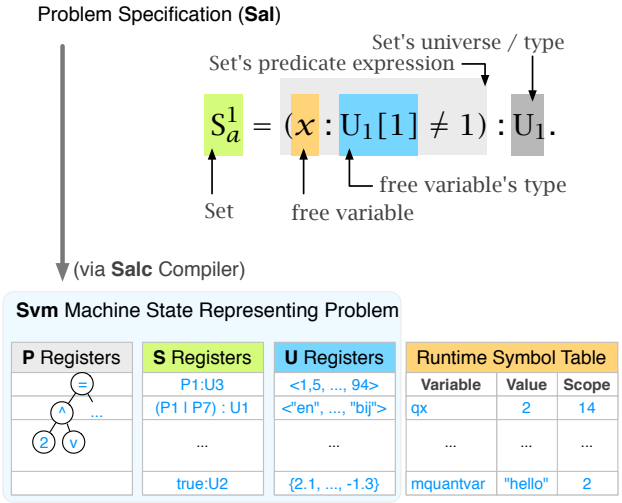


Figure 4. Overview of the Svm architecture.

is the two-dimensional plane of integers, and assigns to it the cross product of `U0` and `U1`. The next three statements define three predicates, `P10`, `P11`, and `P1`. As described in Section 1.2, a predicate is a set-theoretic expression that evaluates to a Boolean value. It may contain constants, bound, and free variables, held together by arithmetic and Boolean connectives. For example, in the predicate `P10`, `px` and `py` are free variables having types `U2[1]` and `U2[2]` respectively, corresponding to the first and second sub-dimensions of the product universe `U2`. Lines 21 and 24 define sets `S1` and `S2`, which are pairings of predicates to universes. In this case, `S1` represents all elements in the universe held in register `U2`, while `S2` represents the restricted subset of the universe in register `U2`, which satisfies the predicate held in register `P1`. The set of points on the convex hull of `U2` can be enumerated using the last statement in the example.

The Sal problem specification is only 26 lines, 15 of which are comments and spacer lines. This succinct Sal *problem definition* can be applied to any set of points in the plane, to obtain the subset thereof corresponding to the convex hull. The definition is *declarative* as opposed to being *imperative*. More importantly, it is a *declarative problem specification*, as opposed to a *declarative algorithm implementation*. As an example of the latter, Franklin et al. [9] describe a Prolog (declarative) implementation of a convex hull solution that required about 200 lines of Prolog, implementing a specific convex hull algorithm.

Such a Sal specification purposefully leaves the algorithm by which it is solved open. Section 5 gives one example of a counterintuitive solution method for this problem specification, which, in the particular scenario considered, yields a more energy-efficient solution than optimal algorithms such as Graham's scan or the Jarvis march, for computing the convex hull.

2.2 The Svm virtual machine architecture

Problem specifications written in Sal are transformed via an assembler/compiler, Salc, into machine state for the Svm virtual machine, whose instruction set is the Sal assembly language. Svm's state is organized into sets of machine registers for holding predicate expressions (*P registers*), set expressions (*S registers*) and universe values (*U registers*), as illustrated in Figure 4. Conceptually, there are an unlimited number of registers of each kind, thus Svm can be thought of as a *memory-to-memory* or *infinite register architecture*, in the vein of other virtual machines such as the Dis virtual machine [26] of the Bell-labs Inferno operating system. Unlike reg-

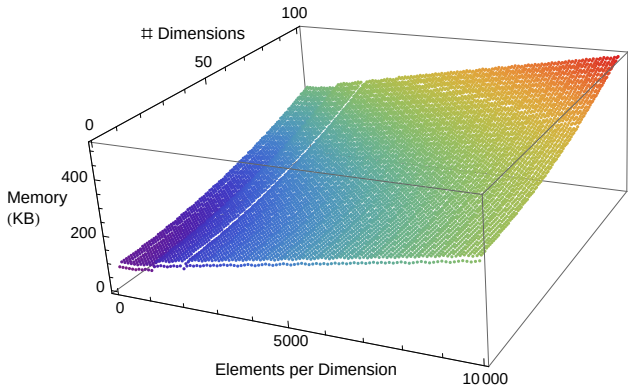


Figure 5. Measured memory usage for creating large multi-dimensional universes in the current Svm implementation.

isters in a real machine, all of these registers hold sophisticated data structures as opposed to simple scalar values. The internal representation of registers is detailed in Section 3, which follows.

3 Svm internal representations

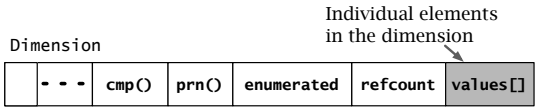
Central to the implementation of Svm are the structures for representing universes, predicates and sets. These structures were designed to facilitate the implementation of operations with minimal overhead, and to facilitate operations such as garbage collection of unreachable items (detailed in Section 3.3.1), and type inference and type checking of expressions to be assigned to registers (described further in Section 3.3.2).

3.1 Representation of universes

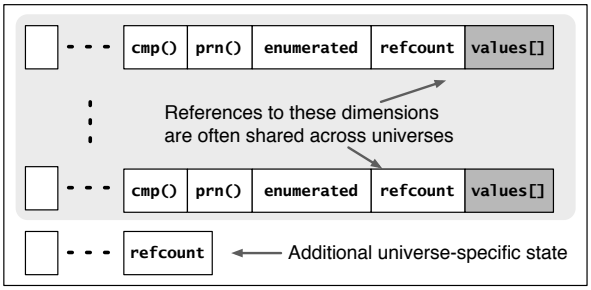
Universes form the basis of Svm's operation. They are single or multi-dimensional collections of integer, real or string values from which sets are defined. In normal usage, there will be multiple universes, corresponding to multiple fundamental quantities from which sets are drawn. In what follows, universes are described in terms of the data structures for their representation.

There are three built-in universes which are always defined in the Svm runtime, and represent the basic non-enumerated single dimensions: integers ($U_{Integers}$), strings ($U_{Strings}$), and reals (U_{Reals}). New universes may be created by taking the cross product of these basic 1-D dimensions. For example, a universe U_a may be defined as $U_a = U_{Integers} \times U_{Integers} \times U_{Integers}$, to represent all possible 3-tuples of integers. Similarly, $U_b = U_{Integers} \times U_{Strings}$ is the universe of 2-tuples of integer-string pairs. Universes may also be defined by lists of 1-D elements, e.g., $U_c = \{1.0, 2.0, 33.5, 29.7\}$ (unordered) or $U_d = \langle 1, 4, 9, 10, 442 \rangle$ (ordered), or ranges, e.g., $U_e = \langle 1 \ldots 1000 \quad \Delta 3 * \imath + 1 \rangle$.

The manner in which universes and dimensions are handled facilitates compact representation. First, dimensions are maintained copy-on-write, with, e.g., a dimension used in two different product universes maintained as references to a single copy. Reference counts are maintained to determine when such copies are really necessary, and enable a dimension to be garbage-collected when all the universes referring to it have been deleted. Second, in a universe cross product, the resulting universes grow linearly, as *the product space is never enumerated*; instead, a "latent" or "lazy" symbolic representation of the product space is maintained. Because a single dimension data structure is used to represent each dimension (with the potential for multiple universes referencing the same dimension), an n-dimensional product universe of dimensions having size $|U_x[i]|$, grows as $k \cdot n$, rather than as $|U_x[i]|^n$, where k is a small constant on a given host platform, a function of the size of a pointer



(a) The `Dimension` data structure is used to represent a one-dimensional universe, holding its elements when finite.



(b) The `Universe` data structure is used to represent a (possibly) multi-dimensional universe.

Figure 6. Logical structure of universes, and the structure of their sub-dimensions.

on the host architecture. To illustrate, Figure 5 plots the memory usage in the current implementation of Svm, as a function of number of elements per dimension (from 1 to 10,000 integer elements), and number of dimensions (from 1 to 100 dimensions). The space of elements represented is from 1 to 10^{400}. Although the space of elements represented is enormous, because of the efficient product space representation, it requires less than 500 KB of memory.

Figure 6(a) illustrates the structure of the individual dimensions in a universe, and Figure 6(b) illustrates the logical structure of universes, which may have one or more sub-dimensions. The actual realization of these structures contain a few additional implementation-specific fields, which are not discussed further here. The `values[]` arrays in a dimension, if the dimension is not defined to be continuous (i.e., the `enumerated` flag is set), contains the primitive elements of the dimension. The `cmp()` function of a dimension returns a number greater than, less than, or equal to zero, if the difference of the values of two arguments is the self-same, and is used in operations such as checking set dominance; if not defined, the dimension is considered to be *unordered*. The `prn()` function is used to print individual dimension elements into a string buffer.

3.2 Specification of Boolean predicates

Figure 8 illustrates the structure of the elements which make up the Boolean predicate tree of a set. There are eight types of Boolean predicates in Svm:

- **Boolean constants** ($bool$: TRUE, FALSE).
- **Quantifiers** ($(var, bool) \mapsto bool$: **EXISTS**, **FORALL**) have a bound variable and Boolean predicate as their children, and evaluate to a Boolean value.
- **Boolean connective predicates** ($bool \mapsto bool$: **AND**, **OR**, **XOR**, **NOT**) are binary and unary Boolean functions of the truth (Boolean) value of their child predicates. The connective predicate nodes can only be parents of Boolean-valued nodes (comparator predicates or other connective predicates).
- **Comparator predicates** ($arith \mapsto bool$: **EQ**, **NE**, **GT**, **GE**, **LT**, **LE**) are functions of arithmetic values from the dimensions of the given set, e.g., a function

```
DIDXS2ELEM(U_a, didxs)
 1  ▷ U_a is a Universe; didxs is a dimension index array.
 2  n ← 0
 3  for i ← 0 to |U_a| − 1
 4      do
 5         if i == 0
 6           then n ← didxs[0] + n
 7                continue
 8         tmp ← |U_a[0]|
 9         for j ← 1 to i − 1
10             do tmp ← |U_a[j]| · tmp
11         n ← tmp · didxs[i] + n
12  return n
```

(a) Converting n-dimensional *coordinate value* to an *element index*.

```
ELEM2DIDXS(U_a, elem)
 1  ▷ U_a is a Universe, and elem an element index.
 2  for i ← |U_a| − 1 downto 0
 3      do
 4         if i == 0
 5           then didxs[i] ← elem
 6                break
 7           else nlower ← |U_a[0]|
 8                for j ← 1 to i
 9                    do nlower ← |U_a[j]| · nlower
10         if nlower > elem
11           then didxs[i] ← 0
12           else didxs[i] ← elem/nlower
13                elem ← elem − nlower · didxs[i]
14  return didxs
```

(b) Converting *element index* into an n-dimensional *coordinate value*.

Figure 7. Algorithms for converting between unique element indices and coordinates in a multi-dimensional space.

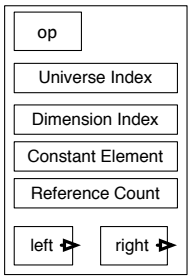


Figure 8. Predicate elements making up the Boolean predicate tree of a set defined on a universe. The dimension index is used to enable Boolean predicates from different universes which are used to form a product universe to still make sense of the product space.

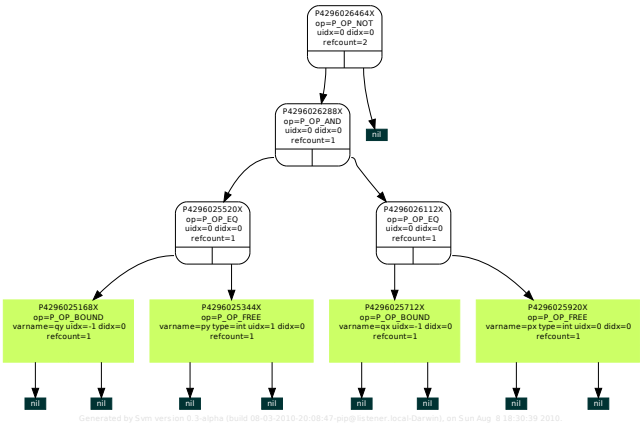


Figure 9. Rendered predicate tree for predicate P10 of Figure 3, auto-generated by Svm runtime.

representing the expression $\{U_1[1] \neq 1\}$; they take two universe values or variables, and yield a Boolean value. Comparator predicates can only be parents of arithmetic and constant or variable predicate nodes.

- **Arithmetic operators** ($arith \mapsto arith$: **ADD**, **SUB**, **MUL**, **DIV**, **MOD**, **POW**, **NRT**, **LOG**) take two arithmetic universe values and yield a new arithmetic value. Arithmetic predicate nodes can only be parents of values and variables (arithmetic predicates and variables/constants).
- **Literals, bindable and free variables** represent specific (enumerated) elements in the universe, free variables (which take on the per-dimension value of the set element to which they are applied) and bound variables (which may take on values determined by their binding quantifiers). These may only appear at the leaves of a Boolean predicate tree.

Each predicate in a predicate tree acts on a single sub-dimension of the universe with which the predicate tree is associated, maintaining the information on its target dimension in the *dimension index* field of its data structure (Figure 8). All dimension index fields in a given predicate tree must thus be updated when dimensions get concatenated in a set cross product, to reference the ordinal position of the dimensions in the new higher-dimensional universe. Each element in a multi-dimensional universe can be referred to by a unique *element index*. The element index is derived from the dimension indices for each sub-dimension, via the algorithm listed in Figure 7(a). Conversely, an element index can be converted into an array of dimension indices via the algorithm in Figure 7(b).

To illustrate these concepts in the context of the convex hull example, Figure 9 shows the predicate tree for predicate P10 of the example in Figure 3, as rendered by the Svm runtime system's predicate tree rendering facility. The root of the tree is a **NOT** predicate, whose only (left) child is a conjunction of two **EQ** predicate subtrees, each having one free variable and one bound variable leaf.

The bound variables in a quantified predicate expression represent formal parameters of the predicate, to be substituted with the actual element to which the predicate is being applied during quantification. A symbol table is used to maintain the correspondence between variables and their associated binding dimensions, placing entries into the symbol table at the point of evaluating a quantifier predicate tree node, and removing it after processing its children.

3.3 Representation of sets

Sets are represented internally as pairings of a predicate tree to a universe. The association of sets with their universes is necessary for the implementation of operations such as set complement. This association also implicitly serves as a *type system* for sets, enabling safety checks such as when two sets occur in a binary set operation. Figure 10 depicts the structuring of sets, illustrating their relation to universes and the dimensions therein.

3.3.1 Reference counting in universes and sets

In the manipulation of sets and universes, many intermediate elements may be created. Some of these elements, such as the dimensions of a universe, are shared across new universes created

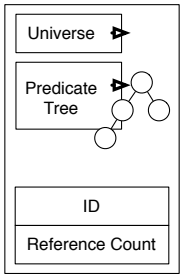


Figure 10. Sets are represented internally as pairings of a predicate tree to a universe.

therefrom (e.g., in a product operation). As sets are created and destroyed, they need to be appropriately allocated and deallocated, the latter being more involved.

A natural method for managing the deallocation of such structures is using reference counting. For universes, the reference count is initially zero, and is increased each time a set is defined on the universe, or when the universe is assigned to a register. The reference count of dimensions is initially one, as they are always associated, at creation time, with a universe. Dimension reference counts are decremented whenever an associated universe is deleted.

Sets and universes can only be deleted if their reference count is zero. A universe with a reference count of zero will have one or more sub-dimensions all with reference count greater or equal to one. Deleting such a universe decrements the reference counts of these sub-dimensions, and if their reference count reaches zero, they are also deleted. Similarly, deleting a set (if its reference count is zero) decrements the reference count of the universe on which the set is defined, and if the universe no longer has any referrers, the aforementioned process of deleting a universe occurs.

3.3.2 Types, type inference, universes, and sets

The separation of the representation of sets into predicates paired with universes enables an elegant solution to the implementation of a type system—the universe of a set is by definition its type. Operations are permitted between two sets (e.g., union, intersection, dominance checking), if they have the same type, and Svm (or the Salc compiler) issues a type error for operations between sets having different types. The only operation permitted between sets of different types is the cross product, which yields a set with a new type.

As a result of taking the cross product of sets, there will often be several new types created over the scope of a Sal problem definition. These new types, which correspond to implicitly created new universes, can be used in the definition of further new sets, in the same way explicitly declared universes can, via the `set2type` operator. An expression consisting of `set2type` with a set as its operand can be used at any point where a universe register is valid in an expression.

4 Implementation

The current implementation of Sal/Svm comprises four parts: a compiler, Salc, the core Svm implementation as a library, an interactive console interface, and a web interface to enable execution of Svm via a web server.

4.1 The Svm implementation

The runtime system of Svm is implemented in a library whose application programming interface (API) implements valid operations in the Svm semantics. This runtime system library is driven either by: an interactive command line interface; an execution engine that accepts pre-parsed and validated input in binary form, generated by the Salc compiler; or by a Web common gateway interface (CGI) front end.

The interactive command line and Web interfaces parse and execute Sal statements directly, providing interactive feedback and meaningful error messages to assist prototyping and debugging of Sal programs. For local beta users of the system at our institution, we have encapsulated this web interface in a content management system that enables users to post new example programs, comment on existing programs, search through a growing collection of examples, and so on.

4.2 The Salc compiler: Sal to Svm compilation and optimization

Salc is implemented in ANSI C. It consists of a YACC-driven parser front end which performs syntax and semantic checking, an optimization middle end, and a code generator that emits a linearized form of the intermediate abstract syntax tree (AST).

A small number of static optimizations on the in-memory representation are currently implemented, prior to emission of the binary. These optimizations include rearrangement of nodes in the AST to enable more efficient short-circuiting of evaluations. Several optimizations that take advantage of the execution semantics of Svm are currently being investigated; these are not detailed here due to space limitations.

Even though the binaries generated by Salc must be executed over Svm, and are not directly executed by the host system, Salc uses the ELF object file format for storing the intermediate representation. The intention of this design choice is to facilitate the use of a wide variety of existing tools for obtaining basic information of compiled Sal programs, such as the *objdump*, *strings(1)*, *size(1)* or the *nm(1)* utilities, across a variety of host platforms.

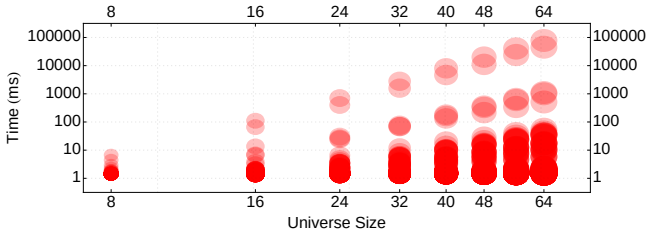
4.3 Bootstrapping and validation

A challenge faced by designers of new programming platforms, whether high-level languages or low-level virtual machines such as Svm, is the need for a collection of input programs for benchmarking and regression testing. While new applications may be developed by a system's early users, which might serve as initial benchmarks and validation tests, the number of such programs is often small. This challenge is even more acute when the language being implemented is a low-level language intended to be the target of a higher-level language compiler, and not always the direct target of human programmers.

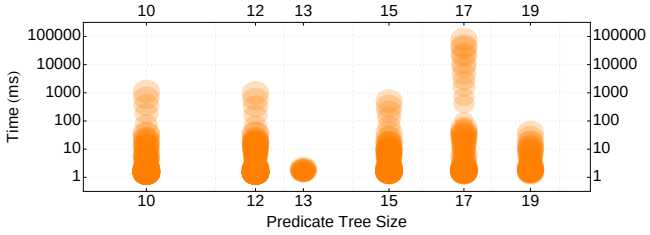
To address this *bootstrapping* challenge, a facility for automatically synthesizing valid Sal programs was implemented. Given a universe definition, the Sal program synthesizer builds a predicate tree of a user-specified size that can be paired with the universe to form a valid set definition. The synthesizer achieves this by first building a random binary tree, and then progressively converting nodes in the tree into valid predicate tree node types, based on the Sal grammar, using the same code the Salc compiler uses for checking validity.

Achieving fast synthesis rates for large programs is challenging. Martin and Orr [13] present an algorithm for generating random binary trees, and Siltaneva and Mäkinen [19] provide a comparison of several such algorithms. In our case however, the challenges lie more with the stages subsequent to building the initial binary tree, where the randomly generated tree's nodes must be assigned roles that correspond to Sal grammar elements.

Over the course of several months, the synthesizer has generated 572 valid programs that we currently use for regression testing and benchmarking. These programs, which currently range from 10- to 19-element predicate trees, can be composed to form more complex valid predicates, and hence, longer programs. Figure 11 presents an outline of the properties of these synthesized programs, showing the



(a) Correlation between universe size, runtime, and set size.



(b) Correlation between predicate tree size, runtime, and set size.

Figure 11. Characteristics of the suite of 572 synthesized Sal "microbenchmarks" used for regression testing. Each oval in (b) corresponds to a unique benchmark; oval sizes in both (a) and (b) are proportional to the set size.

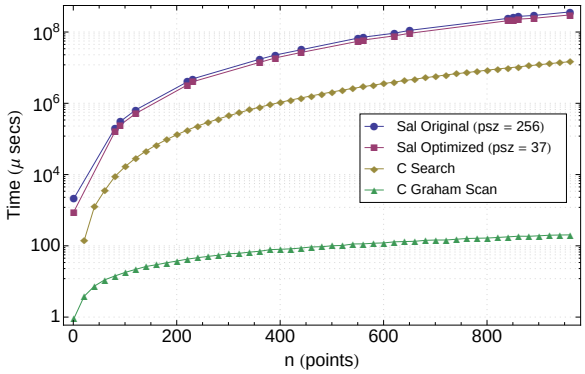


Figure 12. Performance of Sal implementations of convex hull problem solution, with and without optimization, versus C language search-based convex hull solution, and C language implementation of step-optimal Graham's scan algorithm.

range of set sizes they represent and the execution duration for their evaluation, as a function of their predicate tree size, and the size of the universe they are paired with.

Using these and our other user-provided inputs, in combination with tools such as Valgrind [15] permitted many functional and implementation bugs to be caught that were not identified during testing with only human-authored inputs.

4.4 Preliminary performance evaluation

The objective of specifying computation in Sal, rather than using an algorithm to explicitly describe the computation's behavior, is to enable the runtime system to identify problem definitions for which there is an efficient algorithm for their solution on a given platform, and to use that algorithm instead. In the absence of such a match however, as will be demonstrated in Section 5, even the seemingly poor evaluation strategy of using an enumerative search turns out to be beneficial for some classes of execution platforms.

The extension of the Svm runtime to perform such *problem isomorphism* identification is the subject of ongoing active development. Figure 12 however presents an initial evaluation of a single such problem definition—the *convex hull* of points in the plane—implemented in Sal and evaluated on Svm, using Svm's enumerative solution finder. This implementation is compared against a hard-coded C-language implementation of the same enumerative solution finding, as well as a C-language implementation of a known-optimal algorithm (the Graham's scan [8]). The measurements were performed on a 2.8 GHz Intel® Core™ i7, running MacOS® 10.6.2, with gcc 4.2.1 as the compiler, and optimization flags `-O3` for the C-language convex hull implementations; the Svm implementation was compiled with the same compiler and optimization flags. In the measurements, when the times involved were small and thus more affected by measurement noise (due to the preemptive scheduling quanta of the operating system), each plotted point is the average over 50 runs at the given input problem size.

Starting from a user-supplied Sal implementation of the convex hull *problem definition* (topmost curve in the figure), the optimized Sal problem definition removes redundant nodes from the predicate trees in the problem definition. This reduction in the predicate tree size from 256 nodes to 37 nodes, combined with transformations to optimize for the manner in which Svm evaluates predicate trees, leads to a performance gain, for the largest problem size, of 17%. However, in comparison to the C-language implementation of the enumerative search, this optimized Sal execution is still a factor of 20.5× slower. The C-language implementation of the known-optimal algorithm is even faster still—thus there is significant potential for identifying such problem definitions in Sal, and evaluating them using known-optimal algorithms for the given execution model or platform. However, as demonstrated in the following section, the optimum execution method with respect to metrics such as energy usage, may still be the enumerative approach. Even though it performs more computational steps, the overall energy usage can be smaller in certain device technologies, due to the possibility for parallel evaluation that enumeration permits, employing slower per-processor execution, at lower operating voltages, for a reduction in total energy.

5 Sal as a Computation Platform for Future Device Technologies

The most computationally-efficient algorithms for computing the convex hull of a set of n points in a 2D plane have step complexity $\Theta(n \log n)$—their *worst-case* complexity is $O(n \log n)$, and they are known to be asymptotically optimal [8]. The associated energy usage for this computation will be denoted E_{alg}, when executing on a processor operating at voltage V_{alg}. While a given Sal problem definition can be evaluated using a variety of different strategies, it can also *always* be evaluated by enumeration.

An enumeration on the Sal problem definition presented in Figure 3, computes the convex hull in deterministic $c_1 \cdot n^3 + d_1$ steps, with the search centered on each of the points in question being independent. The search is thus embarrassingly parallel, with degree of parallelism equal to the size of the set's universe, and with no need for communication between parallel partitions. If all primitive steps consume the same amount of energy, then the energy consumed in the search will be

$$\left(c_1 \cdot n^3 + d_1\right) \cdot \frac{E_{alg}}{c_2 \cdot n \log n + d_2} \quad \text{Joules.} \tag{1}$$

If n processors are employed in the search, each completes in $\Theta(n^2)$ steps, and all processors may proceed in parallel. If the processors may operate at a minimum voltage of V_{alg}/k, and if energy

varies as V^{β}, the n processors will consume a total of

$$n \cdot \frac{\left(c_1 \cdot n^2 + \frac{d_1}{n}\right) \cdot E_{alg}}{k^{\beta} \cdot (c_2 \cdot n \log n + d_2)} \quad \text{Joules.} \tag{2}$$

For search to be more energy-efficient than the algorithmic computation therefore, requires

$$k^{\beta} > \frac{c_1 \cdot n^3 + d_1}{c_2 \cdot n \cdot \log n + d_2}. \tag{3}$$

For CMOS, the theoretical maximum value of k is in the range of 10–18 for current process technologies, while β is typically 2 for any logic technology in which logic stages pass information by capacitive charge transfer.

The constants c_1, d_1, c_2 and d_2 were determined by fitting the measurements presented previously in Figure 12 to the respective quadratic and logarithmic complexity models. The range of problem sizes for which the enumerative search would yield better energy efficiency, for these values of the constants c_1, d_1, c_2 and d_2, and with a maximum value of k^{β} fixed at 200, is upper-bounded by 70 for the C-language enumerative search (Figure 13), and by 153 for the Svm enumeration (Figure 14). The latter is larger, since it would require *more* hardware parallelism in order to achieve better energy efficiency than the step-optimal Graham scan.

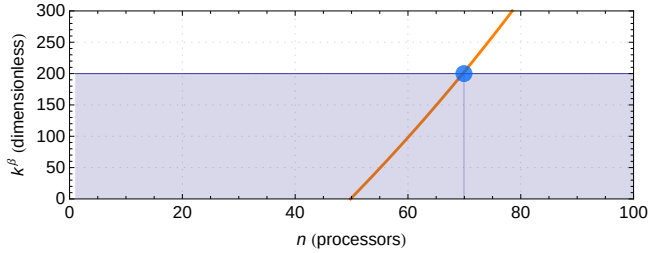


Figure 13. Plot of right hand side of Equation 3 versus number of processors, n, for the values of c_1, d_1, c_2 and d_2 from the experimental characterization of the step-optimal convex hull algorithm, versus the C-language enumerative search.

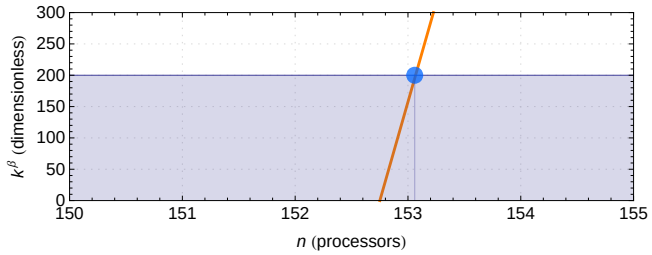


Figure 14. Plot of right hand side of Equation 3 versus number of processors, n, for the values of c_1, d_1, c_2 and d_2 from the experimental characterization of the step-optimal convex hull algorithm, versus the Svm enumerative search.

6 Related Research

Examples of the implementations of enumerative set facilities include the language-level set representation facilities postulated by Hoare and Wirth [28], set manipulation facilities in symbolic computing packages such as Mathematica® [29], the set handling facilities provided by the C++ standard template library (STL) [16], the Java® utility libraries [22], or the Mac OS® "Cocoa" framework [2]. Other relevant related research includes set manipulation languages and notation, declarative programming languages, database query languages, miscellaneous set representation techniques such as binary decision diagrams and their variants, and computer algebra systems.

6.1 Sets in programming languages

SETL [17] is a general purpose programming language, with constructs for procedures, iteration, variables, and the like. SETL is built on the representation of *finite sets* (enumerated sets), and makes no clear distinction between sets and the universes they are defined on. Sal, unlike SETL, is not intended to be a general-purpose programming language. It is purposefully a *machine language for set expressions*, to which (portions of) languages such as SETL might be compiled. While SETL has variables, functions, maps, control flow and more, Sal problem definitions operate on *set registers*, *universe registers* and *predicate registers*. By providing a clear distinction between sets and universes, Sal simultaneously enables the association of sets with well-defined types, and operations which are only meaningful on sets in the presence of associated universes (such as set complement).

In addition to languages that permit computation on sets, there also exist formalisms or notational forms based on set theory. One prominent example is the Z notation [20]. Z and its offspring are not intended for creating executable programs (or problem definitions), but rather, for specifying program properties to enable formal analysis.

6.2 Logic programming languages

Logic programming languages enable the specification of logical predicates, which may be seen as "truths" input into a system by a user. These pieces of knowledge input into the system may subsequently be evaluated. A programming language implementing sets as such logical predicates is Prolog [25], for which the *Warren abstract machine* [24] was designed as an intermediate representation for its execution. Predicates in Prolog represent facts, or the truth value of a particular statement, and are equivalent to enumerated sets in Sal (but do not have associated types). In addition to truth statements, Prolog contains *rules*, which enable the expression of conditional relations between truth statements.

The process of evaluation of predicates in Prolog may be used to achieve computation as a side effect of logical evaluation, enabling the explicit implementation of algorithms. Logic programming languages such as Prolog are based on the fact that *computable functions* [23] can be represented with logic expressions and they achieve computation in the process of performing proofs of propositions in a goal-directed process. Characteristic functions, which are used to represent sets in this work, are *trees* of Boolean predicates, and their evaluation can not lead to iteration or recursion. Thus, Sal problem definitions have no procedural representation.

6.3 Database query languages

Database operations and the associated query languages may be seen as analogous to Svm and Sal. Just as the case in SETL and logic programming languages however, databases represent finite sets (tuples), which have no associated universe, precluding the implementation of the full extent of set theoretic expressions within them. There are however ideas from the domain of databases, such as query optimization, that may be applied to improve the performance of execution in Svm.

6.4 Set representation techniques

Complementary to the discussion of specific programming languages and software systems for processing sets, is the question of the way in which sets are represented in such systems. Finite sets (enumerated sets) may be represented with a variety of structures, including sorted lists and binary trees, when the elements are to be

represented directly in the data structure [4]. They may also be implemented as binary decision diagrams (BDDs) [5] and word-level binary moment diagrams (BMDs), when a prior encoding step is used to assign binary strings to elements of the set. While BDDs may be seen as a form of "symbolic" representation of the elements of a set, they are still an enumerative representation, albeit an *implicit enumeration* in a compact form. Taylor expansion diagrams (TEDs) [6] on the other hand permit true symbolic representations of expressions (and thus, possibly, of set predicates), *if the expressions to be represented are differentiable*.

6.5 Computer algebra languages

Computer algebra systems [3, 12, 14, 21, 29] provide general-purpose facilities for a wide variety of algebraic manipulations. Being often intended for symbolic algebra, calculus and visualization, their facilities focus on these domains, rather than on set theory. For example, while Mathematica® provides built-in facilities for the operations *union*, *join*, *intersection*, and *complement* (in all cases, given two sets), these operators act only on lists—static enumerated sets.

7 Summary and Future Directions

In addition to further development of the Sal/Svm system as presented in this paper, there are a number of directions of evolution which we are currently exploring.

Interpretation versus static and on-the-fly compilation of Sal programs: Since the operations in Sal correspond to complex high-level operations, the overhead of interpretation is low. The Sal assembler already performs all the expensive string processing and represents Sal programs in memory with a format that is efficient to process.

Possible hardware implementation strategies: While the foregoing potions of this article have discussed how Sal/Svm was implemented in software, it is a worthwhile exercise to consider ways in which it might be implemented in hardware. Such a thought experiment provides a different view of the possible applications of the ideas presented. Is there a basic abstraction that underlies all the set operations, that could be implemented in hardware?

Normal forms of predicate trees: Predicate trees are not necessarily in a normal form when constructed. Finding a normal form would be useful for many compile-time optimizations, but is however a difficult problem [11] in general. Barring an available normal form, one can only assume sets are identical, if, for the same universe, the predicate trees are identical; if they aren't, they may still be equivalent. It may still be possible to use techniques from interval analysis [10] to find equivalent subtrees; this does not however imply a normal form.

8 References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996.

[2] Apple Inc. NSSet Class Reference. `http://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSSet_Class/NSSet_Class.pdf`, Accessed April 2010.

[3] C. Batut, K. Belabas, D. Bernardi, H. Cohen, and M. Olivier. Pari-gp. Available from `ftp://megrez.math.u-bordeaux.fr/pub/pari`, Accessed April 2010.

[4] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 16–26, New York, NY, USA, 1998. ACM.

[5] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[6] M. Ciesielski, P. Kalla, and S. Askar. Taylor expansion diagrams: A canonical representation for verification of data flow designs. *IEEE Trans. Comput.*, 55(9):1188–1201, 2006.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[8] M. de Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Santa Clara, CA, USA, 2008.

[9] W. Franklin, P. Wu, S. Samaddar, and M. Nichols. Prolog and geometry projects. *IEEE Computer Graphics and Applications*, 6:46–55, 1986.

[10] M. A. Ghodrat, T. Givargis, and A. Nicolau. Expression equivalence checking using interval analysis. 2006.

[11] M. Karr. Canonical form for rational exponential expressions. In *EUROCAL '85: Research Contributions from the European Conference on Computer Algebra-Volume 2*, pages 585–594, London, UK, 1985. Springer-Verlag.

[12] Maplesoft Inc. Maple®. `http://www.maplesoft.com`, Accessed April 2010.

[13] H. W. Martin and B. J. Orr. A random binary tree generator. In *CSC '89: Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pages 33–38, New York, NY, USA, 1989. ACM.

[14] W. A. Martin and R. J. Fateman. The macsyma system. In *SYMSAC '71: Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 59–75, New York, NY, USA, 1971. ACM.

[15] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[16] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[17] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[18] H. Shojaei, T. Basten, M. Geilen, and P. Stanley-Marbell. Spac: a symbolic pareto calculator. In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software co-design and system synthesis*, pages 179–184, New York, NY, USA, 2008. ACM.

[19] J. Siltaneva and E. Mäkinen. A comparison of random binary tree generators. *The Computer Journal*, 45(6):653–660, 2002.

[20] J. Spivey and J. Abrial. *The Z notation*. Prentice-Hall, 1989.

[21] W. Stein. SAGE: Software for algebra and geometry exploration. Available from `http://sage.scipy.org/` and `http://sage.math.washington.edu/sage`, Accessed April 2010.

[22] Sun Microsystems Inc. java.util.set. `http://java.sun.com/javase/6/docs/api/java/util/Set.html`, Accessed April 2010.

[23] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, Series 2(42):230–265, 1936-1937.

[24] D. Warren. An abstract Prolog instruction set. *Technical note*, 309, 1983.

[25] D. Warren, L. M. Pereira, and F. Pereira. Prolog—the language and its implementation compared with lisp. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 109–115, New York, NY, USA, 1977. ACM.

[26] P. Winterbottom and R. Pike. The design of the Inferno virtual machine. In *Hot Chips IX*, 1997.

[27] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, 1977.

[28] N. Wirth. *Grundlagen und Techniken des Compilerbaus*. Addison-Wesley, Bonn, 1st edition, 1996.

[29] S. Wolfram. *The Mathematica® book*. Cambridge university press, 1999.

A Sal Grammar in EBNF Form

The complete Sal language grammar in EBNF [27] form is shown in Figure 15.

```
zeronine                = "0" .. "9" .
onenine                 = "1" .. "9" .
uimm                    ::= "0" | onenine {zeronine} .
intconst                ::= ["+" | "-"] uimm | ireg .
boolconst               ::= "true" | "false" | "maybe" .
drealconst              = ("0" | onenine {zeronine}) "." {zeronine} .
erealconst              = (drealconst | intconst) ("e" | "E") intconst .
realconst               ::= drealconst | erealconst | rreg .
strconst                ::= "\"" {Unicode character} "\"" | wreg .
basetype                = "integers" | "reals" | "strings" .
baseconst               = intconst | realconst | stringconst .
tuple                   ::= "(" baseconst {"," baseconst} ")" .
constsetexpr            ::= "{" tuple {"," tuple} "}"
                        | "{" baseconst {"," baseconst} "}" .
intrangelist            = "..." intconst ["delta" arithexpr] .
intconstlist            ::= (intconst | intrangelist) {"," (intconst
                              | intrangelist)} .
realrangelist           = "..." realconst "delta" arithexpr .
realconstlist           ::= (realconst | realrangelist) {"," (realconst
                              | realrangelist)} .
unorderableintconstdim  ::= "{" intconst {"," intconst} "}" .
orderableintconstdim    ::= "<" intconst {"," intconst} ">" | "integers" .
unorderablerealconstdim ::= "{" realconst {"," realconst} "}" .
orderablerealconstdim   ::= "<" realconst {"," realconst} ">" | "reals" .
unorderablestrconstdim  ::= "{" stringconst {"," stringconst} "}" .
orderablestrconstdim    ::= "<" stringconst {"," stringconst} ">"
                        | "strings" .
constdimexpr            ::= unorderableintconstdim | orderableintconstdim
                        | unorderablerealconstdim | orderablerealconstdim
                        | unorderablestrconstdim | orderablestrconstdim .

hprecbinboolop          ::= "&" | "^" .
lprecbinboolop          ::= "|" .
unaryboolop             ::= "!" .
arith2boolop            ::= "==" | "!=" | ">" | ">=" | "<" | "<=" .
hprecarith2arithop      ::= "*" | "/" | "%" | "pow" | "nrt" | "log" .
lprecarith2arithop      ::= "+" | "-" .
aggrop                  ::= hprecarith2arithop | lprecarith2arithop .
hprecboolsetop          ::= "#" | "><" .
lprecboolsetop          ::= "+" | "-" | "^" .
unarysetop              ::= "powerset" | "complement" .
quantifierop            ::= "forall" | "exists" .
uandop                  ::= "unionover" | "andover" .
setcmpop                ::= "sd" | "wd" .

program                 ::= {stmt} .
stmt                    ::= setinferdefn | predassign | iregassign
                        | wregassign | rregassign | unvdefn
                        | unvinferdefn | miscop .
filename                = strconst .
miscop                  ::= "print" ((printfmt (sreg|preg)) | ureg) [filename]
                        | "delete" (sreg|preg|ureg) | "lsregs" | "luregs"
                        | "lpregs" | "liregs" | "lrregs" | "lwregs"
                        | "load" filename .
printfmt                = "enum" | "randenum" | "prewalk" | "postwalk"
                        | "dot" | "info" .
setinferdefn            ::= sreg "=" setexpr .
unvdefn                 ::= ureg ":" unvexpr "=" constdimexpr .
unvinferdefn            ::= ureg "=" unvexpr .
predassign              ::= preg "=" predexpr .
sreg                    ::= "S"uimm{uimm} .
ureg                    ::= "U"uimm{uimm} .
preg                    ::= "P"uimm{uimm} .
ireg                    ::= "I"uimm{uimm} .
wreg                    ::= "W"uimm{uimm} .
rreg                    ::= "R"uimm{uimm} .
type                    ::= basetype | ureg | "(" "sreg2type" sreg ")" .
unvfactor               ::= type | "(" unvexpr ")" .
unvterm                 ::= unvfactor {hprecboolsetop unvfactor}
                        | unarysetop unvfactor .
unvexpr                 ::= unvterm {lprecboolsetop unvterm} .
aggrexpr                ::= "aggregate" sreg aggrop uimm uimm .
abstrexpr               ::= "abstract" sreg uimm .
uandoverexpr            ::= "(" uandop varintro predexpr setexpr ")" .

identifier              ::= strconst .
varintro                ::= identifier ":" ureg "[" uimm "]" .
vartuple                ::= "(" identifier {"," identifier} ")" .
arithconst              ::= intconst | realconst .
arithfactor             ::= arithconst | varintro | identifier
                        | "(" arithexpr ")" .
arithterm               ::= arithfactor {hprecarith2arithop arithfactor} .
arithexpr               ::= arithterm {lprecartih2arithop arithterm} .
quantboolterm           ::= quantifierop varintro predexpr .
regfullterm             ::= "full" (ireg|rreg|wreg) .
setcmpterm              ::= setexpr setcmpop setexpr .
predfactor              ::= boolconst | preg | "(" predexpr ")" .
predterm                ::= predfactor {hprecbinboolop predfactor}
                        | arithexpr arith2boolop ["@" (intconst|realconst)]
                              arithexpr
                        | quantboolterm | regfullterm | setcmpterm
                        | vartuple "in" ["@" (intconst | realconst)] setexpr
                        | unaryboolop predfactor .
predexpr                ::= predterm {lprecbinboolop predterm} .
minexpr                 ::= "min" sreg .
setfactor               ::= constsetexpr : unvexpr | "{" "}" | "omega"
                        | sreg | "(" setexpr ")" | "(" predexpr ":" unvexpr ")" .
setterm                 ::= setfactor {hprecboolsetop setfactor}
                        | unarysetop setfactor | minexpr
                        | aggrexpr | abstrexpr | uandoverexpr .
setexpr                 ::= setterm {lprecboolsetop setterm} .
```

Figure 15. EBNF grammar for input language of the virtual machine, with both single and multi-character tokens enclosed in double quotes.