SPARE: REFERENCE MANUAL

G A Venkatesh
Charles N. Fischer

Computer Sciences Technical Report #850

June 1989

SPARE : Reference Manual

G A Venkatesh and Charles N. Fischer
University of Wisconsin - Madison
venky@cs.wisc.edu, fischer@cs.wisc.edu

Abstract

The Structured Program Analysis Refinement Environment (SPARE) is a tool for rapid prototyping of program analysis algorithms through high-level specifications. An analysis algorithm is specified through denotational specifications. The specification language is based on the notation of lambda-calculus and the conventions used for writing denotational specifications for semantics of programming languages. Language features have been specially designed to express analysis algorithms in a clear and concise fashion.

SPARE is designed to be used in conjunction with the Synthesizer Generator [2]. Analysis specifications are translated into specifications in the Synthesizer Specification Language (SSL). The SSL translation, combined with the SSL specification for an editor for the language on which the analysis is defined, can be used to generate an editor for the language. The generated editor performs the specified analysis on programs input to the editor and the results can be displayed to verify the analysis.

This work was supported by National Science Foundation under grant CCR 87-06329.

Authors' address: Computer Sciences Department, University of Wisconsin, 1210 W.Dayton, Madison, WI 53706

Table of Contents

1. Introduction

The Structured Program Analysis Refinement Environment (SPARE) is a tool for rapid prototyping of program analysis algorithms through high-level specifications. It is designed to be used in conjunction with the Synthesizer Generator [2]. This manual assumes that the reader is familiar with the Synthesizer Generator.

An analysis algorithm is specified through denotational specifications. Readers unfamiliar with the denotational approach to language semantics may wish to read the tutorial paper in [5]. Other literature on denotational semantics include [3,4]. SPARE translates denotational specifications of algorithms into specifications in the Synthesizer Specification Language (SSL). The SSL translation, combined with the SSL specification for an editor for the language on which the algorithm is defined, can be used to generate an editor for the language. The generated editor performs the specified analysis on programs input to the editor and the results can be displayed to verify the analysis.

SPARE provides a syntax-directed editor to construct the denotational specifications. The editor is built using the Synthesizer Generator and follows the editing paradigm described in [2]. When editing is finished, the specification can be written to files in three formats.

(1) A textual representation of the specification.

(2) The abstract structure of the specification.

(3) A representation used by the translator to create a SSL specification.

Only (2) can be read back into the editor for further edits.

The language supported by SPARE is based on the notation of lambda-calculus and the conventions used for writing denotational specifications for semantics of programming languages. Language features have been specially designed to express analysis algorithms in a clear and concise fashion. It should be noted that unlike the abstract mathematical notation used to provide denotational semantics, the SPARE language is an applicative language with an underlying evaluation model specified through an axiomatic definition [6].

A specification consists of *domain declarations* and functions defined over these domains. There are two kinds of functions. *Semantic functions* define the analysis corresponding to various syntactic objects in the language. *Auxiliary functions* provide a way to abstract operations used in the specifications.

Every specification is required to be sufficiently typed to allow complete static type checking of operations. All functions must be defined statically. The editor performs the type checks and provides error messages. It also resolves the types of certain overloaded operations and literals.

The translation process produces SSL specifications that maintain *semantic functions* as attribute values and provide implementations for user-defined domains and operations on those domains. The translated specification can be evaluated using an interpreter provided as a pre-defined SSL function.

This manual describes Prototype 1.0 of SPARE. Section 2 describes the SPARE language. Section 3 provides details about using the editor. Section 4 describes the translation steps used to obtain SSL specifications and conventions used in the translated specifications.

2. SPARE Language

An algorithm specification consists of four sections. The *domain declaration* section defines the lattices (or cpos) that may be used in the analysis as well as any other domains that the functions in the specification are defined on. The optional *auxiliary function* section provides the definitions for functions to abstract operations required in the specifications. The *function declaration* section provides the functionalities of the semantic functions. The *function definition* section consists of a set of semantic equations. Example specifications are provided in Appendix C and D.

2.1. Syntax Notation

The syntax is defined using an extended BNF notation. Nonterminals are denoted by lower case words (possibly with embedded hyphens). Reserved words are denoted in bold. Square brackets enclose optional items and braces enclose a repeated item. A vertical bar separates alternatives. Square brackets and braces when enclosed in quotes denote themselves. The non-terminals with the suffix "-name" denote identifiers unless otherwise specified.

2.2. Lexical Description

As the analysis specifications are constructed using the syntax-directed SPARE editor, many of the syntactic rules are automatically enforced. The following description applies to the parts of the specification that are typed in by the user.

(1) The following character set is used:

> Letters : A .. Z , a .. z
> Digits : 0 .. 9
> Special Characters : " - < > [ ] ( ) $ * + , ’ = . ^ { } / _ : &
> White Space : space tab end_of_line

(2) An identifier is a string of letters, digits and underscores starting with a letter. Identifiers are used to name domains, functions, variables and productions. Upper and lower case letters are considered distinct characters. The following are reserved keywords and may not be used for other purposes.

add	and	argument	boolean	bottom	bound
by	cache	collect	div	do	external
false	fix	foreach	function	head	identity
in	integer	is	label	lambda	let
lifted	list	minus	mult	not	nil
null	of	or	ordered	powerset	store
strict	sub	syntactic	tail	top	topped
true					

The following identifiers are pre-defined and should not be re-defined:

bool	FALSE	id	IDENT	int	INTEGER	TRUE

In addition, the identifiers should not conflict with the reserved and pre-defined identifiers in the Synthesizer Generator [2]. A list of such identifiers is provided in Appendix A.

(3) Literals are either integers or character strings. Integers are sequences of digits. Character string literals begin and end with a double quote (") and may contain any sequence of characters in the character set. A double quote inside a string must be repeated.

(4) The following are compound symbols:

```
-->     [[     ]]     <=     ::
```

(5) Adjacent identifiers, integer literals and reserved words must be separated by either white space or any of the special characters that may not appear in those tokens.

2.3. Domain Declarations

Domain declarations associate names with explicit domain definitions.

```
domain-declaration-section   ::=   { domain-declaration ; }

domain-declaration           ::=   domain-name { , domain-name }
                                       = domain-definition [ ordering-option ]

domain-definition            ::=   primitive-domain
                               |   enumeration-domain
                               |   lifted-domain
                               |   topped-domain
                               |   product-domain
                               |   union-domain
                               |   powerset-domain
                               |   list-domain
                               |   store-domain
                               |   function-domain
```

A domain name must be defined before it is used except when it is used in a function domain constructor. Hence, recursive domains can be constructed only through function domains. A domain name cannot be re-defined within the the same specification. When two or more specifications are used together common domain names must be bound to identical declarations.

Partial orders within domain elements are either pre-defined, induced by domain constructors or explicitly specified by the user. All function domains are considered to be flat.

2.3.1. Primitive Domains

SPARE provides 4 primitive pre-defined domains: boolean, integer, syntactic and label. All the primitive domains are flat domains.

boolean:

The boolean domain contains elements from the set of booleans, **B**.

$$\mathbf{B} = \{ \text{false}, \text{true} \}$$

The reserved identifiers true and false denote values in this domain. The domain has the following pre-defined operations:

and : boolean × boolean —> boolean

or : boolean × boolean —> boolean

not : boolean —> boolean

corresponding to the standard boolean operations.

integer:

The integer domain contains elements from the set of integers, **I**.

$$\mathbf{I} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \}$$

The integer literals denote values in this domain. The domain has the following pre-defined operations:

add : integer × integer —> integer

sub : integer × integer —> integer

mult : integer × integer —> integer

div : integer × integer —> integer

minus : integer —> integer

corresponding to the standard integer operations.

syntactic and label:

The syntactic domain contains as elements the syntactic objects in the language for which the specification is constructed. The identifiers that name nonterminals in the abstract syntax of the language denote values in this domain.

Every syntactic object in the language is associated with a value from the label domain. There are no explicit denotations for values in the label domain. The operation

label : syntactic —> label

can be used to refer to elements in the label domain. The only other pre-defined operation on syntactic domains *cache* is described in Section 2.5.1.

The identifiers id, bool and int can only be declared as syntactic domains in a specification. These names denote syntactic objects as well as values in semantic domains and provide a mechanism to coerce primitive syntactic objects into semantic values without explicit semantic functions. bool and int are coerced into values in boolean and integer domains respectively. Details about use of these identifiers are described in Section 4.

2.3.2. Domain Constructors

The SPARE language contains nine domain constructors to create new domains. Each of the constructors induce partial orders (a flat ordering in some cases) in the new domains. The facilities to define alternative orderings are discussed in Section 2.3.4.

ENUMERATION DOMAIN

```
enumeration-domain    ::=    "{" string-literal { , string-literal } "}"
```

For example, the declaration for `Signs` in Appendix C is as follows:

```
Signs = { "+","-","0" } ...
```

The domain is ordered flat unless an explicit ordering is specified. String literals can be shared (i.e., overloaded) between different enumeration domains. The elements of an enumeration domain are denoted by the string literals with explicit type qualification (Section 2.5.1). Examples: `Signs "+"`, `Signs "-"`.

LIFTED DOMAIN

```
lifted-domain    ::=    lifted domain-name
```

The lifted domain contains all the elements of the base domain plus an additional element $\perp$. The overloaded identifier bottom denotes the element $\perp$ of all lifted domains. The induced ordering is defined as follows:

Let `A = lifted B`.

The partial order $\leq_A$ is defined as,

$$\forall\, a_1, a_2 \in A,\ a_1 \leq_A a_2 \text{ iff } a_1 = \perp \text{ or } a_1, a_2 \in B \text{ and } a_1 \leq_B a_2$$

Lifted domains inherit all the pre-defined operations on the base domain. If any of the arguments are $\perp$, they return the $\perp$ element of the result domain (i.e., they are strict). If the result domain does not have a $\perp$, the operations are undefined on $\perp$ arguments.

TOPPED DOMAIN

```
topped-domain    ::=    topped domain-name
```

The topped domain contains all the elements of the base domain plus an additional element $\top$. The overloaded identifier top denotes the element $\top$ of all topped domains. The induced ordering is defined as follows:

Let A = topped B.

The partial order $\leq_A$ is defined as,

$$\forall\, a_1, a_2 \in A,\ a_1 \leq_A a_2 \text{ iff } a_2 = \top \text{ or } a_1, a_2 \in B \text{ and } a_1 \leq_B a_2$$

Topped domains inherit all the pre-defined operations on the base domain. The operations are undefined if any of the arguments are $\top$.

Domains can be turned into lattices by using the lifted and topped constructors.

PRODUCT DOMAIN

```
product-domain    ::=    domain-name * domain-name { * domain-name }
```

The product domain contains tuples of elements from the component domains.

Let $A = A_1 * A_2 * \cdots * A_n$.

An element of A is denoted by $(x_1, x_2, \cdots, x_n)$ where x_i denotes an element of A_i. The select operator (denoted by ^) is pre-defined on all product domains.

$$(x_1, x_2, \cdots, x_n)\,\hat{}\,i = x_i \text{ if } 1 \leq i \leq n$$

The induced ordering $\leq_A$ is defined as follows:

$$(x_1, x_2, \cdots, x_n) \leq_A (y_1, y_2, \cdots, y_n) \text{ iff } x_i \leq_{A_i} y_i,\ 1 \leq i \leq n$$

A product domain has a least element if and only if all the component domains have least elements. The least element if it exists is denoted either by the tuple $(\perp_{A_1}, \perp_{A_2}, \cdots, \perp_{A_n})$ where $\perp_{A_i}$ denotes the least element in A_i or synonymously by the identifier bottom.

UNION DOMAIN

```
union-domain    ::=    domain-name + domain-name { + domain-name }
```

The disjoint union domain contains elements from its component domains with the elements implicitly labeled to mark their origins. The elements of an union domain are denoted by explicit type qualified denotations of elements of the component domains.

For example, let `A = integer + boolean`. `(A 1)` and `(A true)` denote elements of A. Note the possible ambiguity in the denotation of elements if the list of component domains contain the same domain name more than once. The resolution of the origin in such a case is left to the implementation. The specification should not depend on the resolution rule used.

The operator *is* is used to test the origin of an element of an union domain. In the above example, `(A 1)` *is* `integer` and `(A true)` *is* `boolean` evaluate to `true` while `(A 1)` *is* `boolean` evaluates to `false`.

Let $A = A_1 + A_2 + \cdots + A_n$.

The induced ordering $\leq_A$ is defined as follows:

$(A\ a_1) \leq_A (A\ a_2)$ iff $(A\ a_1)$ is A_i and $(A\ a_2)$ is A_i and $a_1 \leq_{A_i} a_2$ for some i, $1 \leq i \leq n$

Union domains have no least element.

POWERSET DOMAIN

powerset-domain	::=	**powerset of** domain-name

The powerset domain contains as elements sets of elements from the component domain. The elements of a powerset domain are denoted by $\{x_1, \cdots, x_n\}$ where x_i denote elements in the component domain. The induced ordering is the subset relation. { } and the identifier bottom synonymously denote the least element. The operators <= , + , * , – and in are pre-defined as the standard set operations subset, union, intersection, difference and membership respectively.

LIST DOMAIN

list-domain	::=	**list of** domain-name

The list domain consists of lists of zero or more elements from the base domain. The list with zero elements is denoted by the overloaded identifier nil. For all list domains `A = list of B`, the following operations are pre-defined:

$:: \ : B \times A \rightarrow A$	Concatenation operator
$\& \ : A \times A \rightarrow A$	Append operator
head : $A \rightarrow B$	
tail : $A \rightarrow A$	
null : $A \rightarrow$ boolean	

A list domain is considered to be flat unless explicitly ordered.

STORE DOMAIN

store-domain	::=	**store** domain-name --> domain-name

Store domains contain as elements specialized functions with the functionality specified by the declaration. The empty store, denoted by bottom[ ] (or just bottom), maps all elements in the first domain to the least element in the second domain if it exists. Otherwise, the function is undefined for all elements.

Let `A = store B --> C.`

The pre-defined operation *lookup* : $A \times B \longrightarrow C$, denoted by a[b] where a : A and b : B, provides the element of C to which b is mapped. An element of the store domain can be constructed from another element using the pre-defined operation *update* : $A \times B \times C \longrightarrow A$. The update operation uses the notation a[c/b] where a : A, b : B and c : C. There is also an expression to denote multiple updates and is described in Section 2.5.1.

Let a′ be the store denoted by a[c/b]. a′ is defined as follows:

$$a'[x] = \text{if } x = b \text{ then } c \text{ else } a[x]$$

The store domain is ordered pointwise.

$$\forall\, a_1, a_2 \in A, a_1 \leq_A a_2 \text{ iff } a_1[x] \leq_C a_2[x] \;\; \forall\, x \in B$$

If the codomain for the elements in the store domain contains the least element then all the elements in the store domain are total functions and the empty store is the least element of the store domain. Note that in *lifted* store domains, bottom denotes the least element of the domain while the empty store is denoted by bottom[ ]

FUNCTION DOMAIN

function-domain	::=	**function** [argument-list] --> domain-name
argument-list	::=	domain-name { --> domain-name }

The elements in the function domain are denoted using expressions described in Section 2.5.1. All function domains are considered to be flat. Domain names can be used in function domain declarations before they are defined.

2.3.3. The join operation

In addition to the pre-defined operations mentioned in the previous sections, the overloaded operator join is pre-defined on all domains.

JOIN

The operation join : $A \times A \longrightarrow A$ for any domain A is denoted by +. If a_1 and a_2 denote values in the same domain A, then $a_1 + a_2$ denotes the least upper bound of the values denoted by a_1 and a_2. If the least upper bound for the two values does not exist then the join is undefined.

UNARY JOIN

The binary operator join can be generalized to denote the least upper bound of a set of elements. The unary join is defined on powerset and store domains only.

Let `S = powerset of A`. If `s` denotes an element in `S` then `+s` denotes the element a in `A` (if it exists) where $a = \mathrm{lub}\{ x \in A \mid y \textit{ in } s, x = M[\![y]\!] \}$

Let `S = store I --> R`. If `s` denotes an element in `S` then `+s` denotes the element r in `R` (if it exists) where $r = \mathrm{lub}\{ x \in R \mid \exists\, i{:}I, s[i]=y, x = M[\![y]\!] \}$

$M[\![y]\!]$ is the semantic value denoted by the syntactic object `y`.

The unary join is undefined if the least upper bound does not exist. For the powerset domain `P` and the store domain `S`, the unary join operation is total if and only if `A` and `V` are complete join semi-lattices.

2.3.4. Ordering Specification

ordering-option	::=	**ordered by** ordering
ordering	::=	order-enumeration
	|	order-name
order-enumeration	::=	(order-tuple { , order-tuple })
order-tuple	::=	< string-literal , string-literal >

There are two ways to provide explicit ordering.

ENUMERATION

This method may be used for enumeration domains. The ordering is specified as a list of tuples <a , b> where a and b are string literals specified in the enumeration domain declaration. The tuple establishes the order a ≤ b. It is sufficient to provide all the "immediately less than" relations in the required ordering. The ordering through transitivity is implicitly established. The declaration for `Signs` in Appendix C is an example.

```
Signs = {"+","-","0"} ordered by (<"0","-">,<"0","+">);
```

EXTERNAL DEFINITION

This is an escape clause to specify complex orderings in SSL rather than in the SPARE language. It can also be used to provide a library of standard orderings (e.g. power-domain orderings). order–name is an identifier that stands for a collection of four SSL functions providing the join operation, the equality test, and the two constant functions that return the bottom and top elements if they exist.

<order–name>*_join* : $D \times D \rightarrow D \cup \{error\}$

<order–name>*_equal* : $D \times D \rightarrow$ boolean

<order–name>*_bottom* : $\rightarrow D \cup \{error\}$

<order–name>*_top* : $\rightarrow D \cup \{error\}$

SPARE will assume that the functions have been named as above and generate calls to these functions in the translation. Writing these functions requires information about the conventions used in translation of SPARE language to SSL. The translation details are provided in Section 4.

2.4. Function Declarations

The functions in SPARE language are of two kinds - *auxiliary functions* that abstract operations in the specifications and *semantic functions* that specify the analysis for the various syntactic objects in the language. *Auxiliary functions* are declared and defined in the *auxiliary function* section while *semantic functions* are declared in the *semantic function declaration* section and defined as *semantic equations* in the *semantic function definition* section.

2.4.1. Function Type Declaration

The function type declaration is similar to the function domain declaration except for the facility to specify *strictness*. The function type declaration is used to specify the types for both *semantic functions* as well as *auxiliary functions*.

```
function-type-declaration    ::=    [ funarg-list ] --> domain-name

funarg-list                  ::=    [ strict ] domain-name [ --> [ strict ] domain-name ]
```

The *strict* option forces the function to be strict in the arguments for which the option is used. If the codomain does not have a $\perp$ then the function will be undefined if any of the arguments with the strict option are $\perp$.

Although the strict option appears in the type declaration, it is not considered as part of the type description (functionality) of the function. It implicitly modifies the function body.

2.4.2. Auxiliary Functions

aux-function-section	::=	{aux-function}
aux-function	::=	function-name : function-type-declaration **is** expression ;

Auxiliary functions are defined using *expressions* described in Section 2.5.1. *Auxiliary function* names must be defined before they are used.

2.4.3. Semantic Function Declarations

semantic-func-decl-section	::=	semantic-func-decl { semantic-func-decl }
semantic-func-decl	::=	function-name "[[" domain-name "]]" : function-type-declaration [collect-option] [external-option] ;
external-option	::=	**external**

The semantic function declarations associate function names with syntactic objects and provide their type declarations. Semantic functions can only be associated with objects from syntactic domains. For example, in Appendix C, the declaration

```
E [[exp]] = Store --> Sign_Lattice;
```

associates the semantic function `E` with the syntactic object `exp`.

Semantic functions cannot be declared for the pre-defined syntactic objects id, int and bool. Functions declared as external are assumed to be defined in other specifications. These functions must be declared with identical type declarations in the specifications in which they are defined.

2.4.4. Collect Option

collect-option	::=	**collect** collect-domain
collect-domain	::=	[**powerset of**] selected-domain
selected-domain	::=	domain-name [^ integer]

The collect option specifies information that must be collected and associated with syntactic objects during the analysis. The domain–name in the collect option must appear in the function type declaration. If the name appears more than once in the type declaration, the occurrences must be differentiated by using suffixes of the form "$n" where n is an integer. For example, in the declaration

```
P [[ prog ]] : D$1 --> S --> D$2 collect D$1;
```

the value of the first argument rather than the result is collected. The suffix does not create new domains. Both `D$1` and `D$2` are the same as the domain `D`.

Whenever a semantic function with the collect option specified is evaluated, the value of the specified argument (or result) is included in the information associated with the instance of the syntactic object corresponding to the current evaluation. The pre-defined operation

cache : syntactic —> cache

can be used to access information associated with syntactic objects. cache is a store domain

cache = `store label --> collect-domain`

where `collect-domain` is determined from the declaration. The collect-domain will be one of the following:

(1) The domain domain–name

(2) A component domain of domain–name where domain–name is a product domain.

(3) The powerset of (1)

(4) The powerset of (2).

If the semantic function associated with an instance of a syntactic object is evaluated more than once, the information from successive evaluations is "joined" together using the *join*(+) operator in the collect-domain.

If the collect option is specified for more than one semantic function, then the cache domain maps label to an union domain of all the respective collect domains. Semantic functions for the same syntactic object must specify identical collect domains.

2.5. Semantic Function Definitions

semantic-func-defn-section	::=	semantic-equation { semantic-equation }
semantic-equation	::=	function-name "[[" (production) "]]" = expression
production	::=	production-name { syntactic-domain-name }

Semantic functions are defined through a list of *semantic equations*. A semantic equation defines the function for a single production in the abstract syntax for the syntactic object with which the function is associated. Semantic equations can be listed in any order. A semantic equation must be provided for each production in the abstract syntax. A production is specified as a list (possibly empty) of syntactic domains headed by a production name. If the same syntactic domain occurs more than once in the production, the occurrences must be differentiated by using suffixes of the form "$n" where n is an integer. For example, a semantic equation from Appendix C is given below:

```
E [[(Add exp$2 exp$3 )]] = lambda x. ( E [[exp$2]] + E [[exp$3]] ) x
```

The conventions used to relate the productions in the specification to an SSL description of the abstract syntax is discussed in Section 4. Appendix D provides the abstract syntax in SSL for a simple language and an analysis specification for that language.

2.5.1. Expressions

```
expression   ::=  constant
               |  variable
               |  primitive-expression
               |  lambda-expression
               |  function-application
               |  expression-composition
               |  expression-union
               |  conditional-expression
               |  fixed-point-expression
               |  let-expression
               |  typecast-expression
               |  ( expression )
```

CONSTANTS

Integer literals, string literals and boolean literals are constants. Integer and boolean literals denote values in integer and boolean domains respectively. String literals denote values in enumeration domains. Reserved identifiers bottom, top and nil are over-loaded and denote constant values from domains as described in Section 2.3. The reserved identifier identity denotes the constant function (see the section on lambda expressions).

VARIABLES

Variables can be used to denote values from various domains. They are either identifiers or semantic function instance denotations. Identifiers are introduced through lambda expressions, let expressions and fixed-point expressions. The pre-defined identifiers bool, int and id denote syntactic objects as well as corresponding values in boolean, integer and identifier–string domains respectively (these are described in more detail in Section 4). Semantic functions instances are denoted by a pair of identifiers consisting of a function name and a syntactic domain name.

PRIMITIVE EXPRESSIONS

```
primitive-expression    ::=   pre-defined-operation
                         |    set-construction
                         |    product-construction

set-construction        ::=   "{" [ expression-list ] "}"

product-construction    ::=   ( [ expression-list ] )

expression-list         ::=   expression { , expression }
```

The pre-defined operations were introduced in Section 2.3. They are summarized in this section. For operands to be compatible, they must denote values in the same domain (i.e., the domains must be name equivalent).

boolean operations:

Operation	Notation	Type	Usage
logical and	and	D × D —> boolean	binary infix
logical or	or	D × D —> boolean	binary infix
logical not	not	D —> boolean	unary prefix

D must be a boolean domain or a boolean domain that has been lifted and/or topped.

integer operations:

Operation	Notation	Type	Usage
addition	add	D × D —> D	binary prefix
subtraction	sub	D × D —> D	binary prefix
multiplication	mult	D × D —> D	binary prefix
division	div	D × D —> D	binary prefix
unary minus	minus	D —> D	unary prefix

D must be an integer domain or an integer domain that has been lifted and/or topped.

syntactic and label operations:

Operation	Notation	Type	Usage
label	d'label	syntactic —> label	unary postfix
cache	d'cache	syntactic —> cache	unary postfix

d must denote a syntactic object. The symbol '$' can be used in semantic equations to denote the instance of the syntactic object for which the semantic function is being evaluated. For example, in the semantic equation

```
        S [[ (Assign id exp) ]] = lambda x. x[exp'cache/$'label];
```

`$'label` refers to the label of the syntactic object `(Assign id exp).` `exp'cache` refers to the cache associated with the syntactic object `exp.`

product domain operation:

Operation	Notation	Type	Usage
selection	^	D × integer —> D1	binary infix

D must be a product domain, D1 is a component domain of D.

union domain operation:

Operation	Notation	Type	Usage
origin test	is	D × D1 —> boolean	binary infix

D must be an union domain, D1 is the set of names of component domains of D1.

powerset operations:

Operation	Notation	Type	Usage
subset	<=	D × D —> boolean	binary infix
union	+	D × D —> D	binary infix
intersection	*	D × D —> D	binary infix
difference	-	D × D —> D	binary infix
membership	in	D1 × D —> boolean	binary infix

D must be a powerset domain, D1 is the base domain of D.

list operations:

Operation	Notation	Type	Usage
concatenation	::	D1 × D —> D	binary infix
append	&	D × D —> D	binary infix
head	head	D —> D1	unary prefix
tail	tail	D —> D	unary prefix
nil test	null	D —> boolean	unary prefix

D must be a list domain, D1 is the base domain of D.

store operations:

Operation	Notation	Type	Usage
lookup	s[i]	D × D1 —> D2	binary mixfix
single update	s[v/i]	D × D2 × D1 —> D	ternary mixfix
multiple update	s[foreach x in e1 do e2/e3]	D × D3 × D2 × D1 —> D	quarternary mixfix

D must be a store domain, D1 the index domain of D and D2 the value domain of D. D3 must be a powerset domain. The multiple update consists of a sequence of single updates with the identifier x successively bound to each element of e1. The identifier is introduced with a scope covering e2 and e3.

overloaded operations:

Operation	Notation	Type	Usage
join	+	D0 × D0 —> D0	binary infix
unary join	+	D1 —> D2	unary prefix
equal	=	D3 × D3 —> boolean	binary infix

D0 is any domain. D1 is a powerset domain and D2 is the base domain of D, or D1 is a store domain and D2 is the value domain of D. D3 is any domain except a function or a syntactic domain.

LAMBDA EXPRESSIONS

```
lambda-expression   ::=   lambda identifier . expression
```

Lambda expressions are used to create functions. The evaluation of a lambda expression results in the representation of a function in some function domain. The expression in the lambda expression becomes the function body. The function body is evaluated in the context that is active at the point of definition of the lambda expression augmented with the binding of an actual parameter to the binding identifier. Static scoping is used to determine the context. The lambda expression creates a new scope for the inner expression in which the binding identifier is visible. The pre-defined identifier identity denotes the identity function. The pre-defined identifier bottom is over-loaded to include bottom functions ($\lambda x_1.\lambda x_2.\cdots\lambda x_n.\perp$) and the identifier top is over-loaded to include top functions ($\lambda x_1.\lambda x_2.\cdots\lambda x_n.\top$)

FUNCTION APPLICATION

function-application	::=	expression expression

The first expression must evaluate to a function. The evaluation of the second expression must denote a value in a domain compatible with (i.e., name equivalent to) the argument domain of the function. The result of evaluation of a function application is the result of applying the function to the result of the evaluation of the second expression.

EXPRESSION COMPOSITION

expression-composition	::=	expression . expression

Both the expressions must evaluate to functions. The result of the evaluation of an expression composition is a function that is a composition of the functions resulting from the evaluation of the two expressions. Let the first expression evaluate to the function A —> B for some domains A and B. The second expression must evaluate to a function of type X —> A for some domain X or to a constant function of type —> A.

EXPRESSION UNION

expression-union	::=	expression + expression

Both expressions must evaluate to denote values in the same domain. If the expressions evaluate to non-function domains, the result of evaluation of the expression union is the join (+) of the results of evaluation of the two expressions. If the expressions evaluate to functions, then the result of evaluation of the expression union is a *functional form* [1] defined as follows:

$$f1 \oplus f2 = \lambda x.\ f1\ x + f2\ x$$

CONDITIONAL EXPRESSION

conditional-expression	::=	expression --> expression , expression

The first expression must evaluate to denote a value in the boolean domain. The second and third expressions must evaluate to denote values in identical domains. The result of evaluating the conditional expression is the result of evaluating the second expression if the first expression evaluates to denote true. Otherwise, it is the result of evaluating the third expression.

FIXED-POINT EXPRESSION

fixed-point-expression	::=	**fix** [termination-option] function-name . expression
termination-option	::=	**argument** [bound-option]
	|	**cache** [bound-option]
bound-option	::=	**bound** (integer-literal)

Fixed-point expressions are used to define recursive functions. The function–name is bound to the fixed-point expression and can be used in the expression to refer to it. The termination–option is used to specify termination criterion for the fixed-point expression evaluation. The formal semantics of these options are provided in [6]. Operationally, the termination options control the evaluation of the fixed-point expression as follows:

If the fix argument option is specified, the recursive function is evaluated until the arguments to the recursive call reach a fixed-point. To detect the fixed-point, the arguments to a recursive call are compared with the arguments in the previous call. If they are the same (tested using the *equal* operation), then the recursive call is substituted with the $\perp$ function. Hence, the expression

```
fix argument F. lambda x1. ... lambda xn. E
```

denotes the function

$$\lambda x_1.\ \lambda x_2.\ \cdots\ \lambda x_n.\ \mathtt{E}\ [x_i / \mathtt{xi}]\ [G(\perp_1, \perp_2, \cdots, \perp_n) / \mathtt{F}] \text{ where}$$

$$G = \lambda p_1.\ \lambda p_2.\ \cdots\ \lambda p_n.\ \lambda y_1.\ \lambda y_2.\ \cdots\ y_n.\ \bigwedge_{i=1}^{n}(p_i = y_i) \longrightarrow \perp\ ,\ \mathtt{E}\ [y_i / \mathtt{xi}]\ [G(y_1, y_2, \cdots, y_n) / \mathtt{F}]$$

Note that the use of the termination option does not guarantee termination of the evaluation. The fix argument option is useful when the arguments to successive recursive calls form a strictly non-decreasing chain.

For example, the least fixed-point solution to equations of the form

`x = f(x)` when x is not a function

can be specified using the fix argument option as:

```
(fix argument F. lambda x. (F (f x)) + x) bottom
```

The cache option is similar to the argument option except that the recursive function is evaluated until the cache values computed by semantic functions during the recursive evaluation reach a fixed-point.

The bound option can be used to limit the depth of recursive calls. The integer provides an upper bound on the number of recursive calls. If the fixed-point is not reached within this number of recursive calls, the next recursive call is substituted with the constant function T.

LET EXPRESSION

```
let-expression   ::=   let let-clause { ; let-clause } in
                            expression

let-clause       ::=   identifier = expression
```

The expression in the let expression is evaluated in the context of the let expression augmented by the binding of each identifier to the result of evaluating the expression in the corresponding let clause. Static scoping is used to determine the context. The scope of an identifier extends from the let clause following its definition (if any) to the end of the let expression.

TYPECAST EXPRESSION

```
typecast-expression   ::=   domain-name expression
```

Type casting is used to coerce denotations of values from one domain to another (or possibly itself). Type casting is valid in only the following cases:

(1) The domain-name is the name of a lifted or topped domain and the expression denotes a value in the base domain.

(2) The domain-name is the name of an union domain and the expression denotes a value in one of its component domains.

(3) The domain-name is the name of the domain to which the value denoted by the expression belongs (i.e., the null or trivial coercion).

2.6. Type compatibility rules

(1) Two denotations that denote values in function domains are compatible if and only if belong to two domains that are structurally equivalent. The strict and collect options are not considered as part of the type structure (functionality).

(2) Two denotations that denote values in non-function domains are compatible if and only if they denote values in the same domain (name equivalence) with the exception in (3).

(3) The cache domain implicitly defined through the collect option is a store domain that maps labels to the specified collect domain (Section 2.4.4). The cache domain is compatible with any store domain that is structurally equivalent to it.

3. Using the SPARE Editor

The SPARE editor is constructed using the Synthesizer Generator and follows the editing paradigm described in [2]. It is assumed that the reader is familiar with the editing commands common to all editors generated through the Synthesizer Generator.

The editor is syntax-directed and provides warnings and error messages for type errors. In cases where the type information in the declarations is insufficient to derive the type of an expression, the user must explicitly typecast the expression. For example, consider the expression

```
let x = {1,2,3} in e
```

where `e` is an expression using `+x` (the unary join of `x`). To be type consistent, the set literal can denote a value in any powerset domain over an integer base domain. Such expressions must be explicitly typecast (Section 2.5.1). The editor will output a warning message for such expressions.

Syntactic object names in expressions are automatically enclosed in double square brackets.

The specifications can be written to files in three formats:

(1) The specification text: This is used to get a print copy of the specification and is created by using the write command with the *text* option.

(2) The specification structure: This allows specifications to be saved for future edits. It is created by using the write command with the *structure* option. Specifications saved in this format can be read back into the editor.

(3) The specification translation: This format is used by the translator to create SSL specifications. This is created using the write command with the *text* option on the alternate unparsing scheme for the specification.

SPARE does not perform any consistency checks between productions in the specifications and the abstract syntax of the language specified in SSL. The conventions to be followed are described in the next section. It also does not check the consistency of declarations for identical names in multiple specifications that may be used together.

4. Interfacing with SSL editor specifications

Although the specification translation output by the editor is very close to a SSL specification, it must be passed through a pre-processor that generates several SSL specification files to implement user-defined domains and to support operations on those domains. These files must be included with the editor specifications to generate an editor with the analysis built in.

In order to use SPARE specifications in editors generated through the Synthesizer Generator, it is necessary to know some details about the translation of SPARE specifications into SSL specifications. This section provides these details as well as some conventions that must be followed in creating SPARE specifications.

All the phylum names generated by SPARE in SSL translations are prefixed with the string "_ _" to prevent inadvertent collisions with pre-defined identifiers in SSL as well as identifiers declared by the user in editor specifications.

4.1. Conventions to be used in specifications

The abstract syntax of the language for which an analysis specification is written, is specified in SSL as part of the editor specification. The SPARE specification must follow certain conventions in order to be consistent with the abstract syntax. Appendix D provides an example SPARE specification along with the abstract syntax written in SSL.

4.1.1. Syntactic domain names

The names defined as syntactic domains in SPARE specifications must be production (phylum) names in the abstract syntax. In Appendix D, the the names `prog`, `stmtlist`, `stmt`, `exp` and `b_exp` are declared as syntactic objects and are phylum names in the abstract syntax.

The names `id`, `bool` and `int` are pre-defined phylums and can be used in the abstract syntax. Their SSL declarations are available in the file "SPARE.domains.ssl" which must be included with all editor specifications that use SPARE generated specifications. The user may change these definitions to suit the language for which the editor is to be constructed. The names `id`, `bool` and `int` can be used in SPARE specifications as syntactic objects. When they are used in expressions they are implicitly coerced to denote values in semantic domains. The domains that they denote values in are described in Section 4.6.

4.1.2. Productions and abstract syntax

When a semantic function is associated with a syntactic object, semantic equations must be provided for every production associated with the corresponding phylum in the abstract syntax (including any completing and/or placeholder productions). The production in each semantic equation must use the operator name in the abstract syntax as the production name and list the argument phylum names to this operator.

For example, in Appendix D, the phylum `stmt` has the following production:

```
                                Assign(id exp)
```

The semantic equation corresponding to this production for the semantic function `S` is defined as follows:

```
  S[[(Assign id exp )]] = lambda x. let n=E [[exp]] x in x[ n/[[id]] ];
```

If a phylum name occurs more than once in the argument list or occurs on both sides of the production, then the occurrences must be distinguished by adding a suffix of the form "$n" where n is an integer literal. The numbering begins with 1 and increases from left to right. This is the same convention as the one used in writing attribute equations in SSL specifications. For example, the occurrences of `exp` in productions with two different left hand sides in Appendix D are distinguished as follows:

production	semantic equation
`exp : Add(exp exp)`	`E[[(Add exp$2 exp$3)]] = ...`
`b_exp : Equal(exp exp)`	`Bt[[(Equal exp$1 exp$2)]] = ...`

Although the argument phylum names can be specified in any order (after the different occurrences have been distinguished) in the semantic equation, it is recommended (for readability) that they are listed in the same order as they appear in the abstract syntax.

4.2. Domain representations

Values of all domains used in the specifications are represented in the translation as terms belonging to the pre-defined phylum __userdomains. A unary operator is associated with this phylum for each domain defined in the specification. The single argument is the name of a phylum that implements the particular domain. The structure of __userdomains is not relevant to the user if the editor is to simply display the information provided by the analysis. Reasonable unparsing statements have been provided for these phylums in the file "SPARE.domains.ssl".

The next three sections provide the details necessary to generate editors that use this default display for the information obtained from the analysis. Section 4.6 provides some details about the internal structure of domain implementations to allow the user to process the analysis information in editor specifications.

4.3. Function representations

The semantic functions are defined as attribute values in attribute equations associated with nodes in the abstract syntax tree. The SPARE translation defines two attributes for every semantic function in the phylum associated with the semantic function. The first attribute is a string that labels the node. The label is generated using the SSL built-in function *gensym* and consists of a string with the prefix label– and is unique to each instance of the production. The second attribute is defined to be of type __fexpr which is a pre-defined phylum in "SPARE.domains.ssl". The phylum __fexpr defines the productions required to represent the semantic functions as attribute values. The internal structure of this phylum is not relevant to the user.

The label attribute is named __X_label while the semantic function attribute is named __X where X is the name of the semantic function. The expression corresponding to the semantic function is represented by the value of the second attribute.

4.4. The Interpreter

The interpreter for evaluating expressions in the SPARE language is provided as a pre-defined SSL function with the type:

```
__userdomains  __Interpreter (__fexpr f)
```

and can be used inside attribute equations in editor specifications.

4.5. Displaying the results from the analysis

In most situations the interpreter is used in an attribute equation for the root phylum. The semantic function attribute in the root phylum usually contains the expression corresponding to the analysis of the entire program. However, it is possible to use the interpreter in any phylum for which the corresponding semantic function expects no arguments.

To display the information obtained from the conditional constant propagation specification in Appendix D, the following schema may be used:

```
prog   : Prog   {
                local __userdomains output;
                output = __Interpret(__P);
                ...
                other attribute equations
                ...
                }
```

The attribute output can then be used in the unparsing statement for Prog to display the result of the analysis. The semantic function will be interpreted whenever a change in the program results in a different semantic function. The output will be presented using the default display.

It is possible to control the granularity of edits before successive analyses are carried out by making the attribute `output` a demand attribute and providing two unparsing specifications which are identical except that only one of them displays the attribute `output`. The semantic functions are then interpreted only when the output is to be displayed. All the edits can then be carried out when the principal unparsing scheme (without the output displayed) is in effect. After all the edits are completed one can switch to the alternate unparsing scheme to initiate the interpretation and observe the results of the analysis.

4.6. Domain implementations

All semantic values are represented as terms belonging to the pre-defined phylum __userdomains. It is necessary to know some details about the internal structure of this phylum, if the information obtained from the analysis is to be displayed in a format different from the default, or is to be further processed before display.

The operators (or productions) associated with __userdomains fall into two categories:

(1) Operators common to all specifications:

top and bottom operators:

The nullary operators __bottomrep and __toprep are used to create the terms __bottomrep() and __toprep() that represent $\perp$ and $\top$ respectively in any domain. There may be more than one representation for the top and bottom elements of some domains.

error operators:

These operators are used to catch run-time errors. The nullary operators __notop and __nobottom create terms that represent the non-existence of the least upper bound and $\perp$ respectively during the evaluation of expressions. For example, the join operation on two elements from some domain generates the term __notop() as the result, if the two elements have no least upper bound in the domain.

The third operator __errorvalue is used to cover errors other than the two above. Some of the operations defined in Section 2 are undefined for certain arguments. The __errorvalue operator is used to make these functions total, by generating the term __errorvalue() when the operations are undefined. All operations in the SPARE language propagate the error values.

These values are useful in debugging of analysis specifications.

primitive domain operators:

The unary operators __intdomain, __booldomain and __labeldomain are used to represent values from the primitive domains integer, boolean and label respectively. The single arguments are terms from phylums that implement these domains. The pre-defined phylums int and bool implement the integer and boolean values. The labels are implemented as the SSL-predefined string phylum STR.

The special syntactic objects int and bool used in specifications are implicitly coerced into semantic values by using the operators __intdomain and __booldomain respectively with the corresponding nodes of the abstract syntax tree provided as arguments.

Semantic functions must be defined to provide semantic values of syntactic objects. However, the special syntactic object id is implicitly coerced into the semantic domain by using the unary operator __syntacticdomain. The operator takes a term from the pre-defined phylum id as an argument. The node in the abstract syntax tree corresponding to an instance of id is used as an argument to this operator to obtain the corresponding semantic value.

Values in all function domains are constructed using the unary operator __functiondomain. The argument is a term from the SSL-predefined pointer phylum PTR. An expression is converted into a value in the function domain by using this operator and the pointer to the representation of the expression.

cache domain operator:

The cache domain that is implicitly defined through the collect option is implemented using the unary operator __cache. The argument is a term from the phylum for the implementation of a store domain.

(2) User defined domain operators:

A unary operator is created for each domain defined in the specification. The domain name is used as the name of the operator (i.e., production name). The single argument is the name of a phylum that implements the particular domain. The phylums that implement the various domain constructions are in the file "SPARE.domains.ssl". Appendix E provides the definition of __userdomains generated for the specification in Appendix D.

4.7. External domain order definitions

In Section 2.3.4., the facility to define domain orderings through SSL functions was described. The four functions required for the ordering must be defined as SSL functions with the types given below:

```
__userdomains  <order-name>_join ( __userdomains u1, __userdomains u2)
__userdomains  <order-name>_equal ( __userdomains u1, __userdomains u2)
__userdomains  <order-name>_bottom ()
__userdomains  <order-name>_top ()
```

If the domain for which the ordering is being defined contains other domains as components then the above operations can be defined in terms of the following pre-defined functions (in SSL) to perform the corresponding operations in the component domains:

```
__userdomains  __join ( __userdomains u1,  __userdomains u2)
__userdomains  __equal ( __userdomains u1,  __userdomains u2)
__userdomains  __bottom ()
__userdomains  __top ()
```

5. SPARE Prototype 1.0

This section lists certain features and limitations of the current prototype that are not implied by the description in the previous sections. These features and limitations are most likely to change in future versions.

(1) Functions can be "curried" only if they do not have the strictness and collect options specified in their type declarations. Currying of functions with either the strictness or collect option are not detected by the editor and will produce unreliable results on interpretation.

(2) The editor does not resolve types of all overloaded expressions and requires the user to provide explicit typecasting. Warnings are provided for expressions that require explicit typecasting. The warnings can be ignored. However, type errors in such specifications may result in unreliable results on interpretation.

(3) All domains are currently implemented through list structures. The speed of interpretation can be greatly improved by providing more efficient implementations.

(4) Auxiliary functions cannot be shared among multiple specifications.

(5) Identical domain names declared differently in multiple specifications may produce unreliable results on interpretation.

(6) The interpreter uses a cache to provide incremental evaluation of semantic functions. In the worst case, there is an overhead of roughly 10 percent compared to complete re-evaluation. The incremental evaluation scheme is not very effective for re-evaluation of expressions that involve fixed-point expressions.

Appendix A. Reserved words and pre-defined identifiers in SSL

RESERVED WORDS

and	as	class	default	demand
exported	ext_computers	false	foreign	in
inh	inherited	left	list	let
local	nil	nil_attr	nonassoc	on
optional	prec	repeated	right	root
store	style	syn	synthesized	transform
true	typedef	with		

PRE-DEFINED IDENTIFIERS

ASYNCHPAR	attr	Attr	boolean
Bottom	BOTTOM	Bool	character
Char	coll	COLLECTIONS	default_store
double	dreal	DontCare	DONTCARE
Dreal	Fcn	FCN	GetBottom
hashtable	integer	Int	LINEBREAK
mapping	NAME	NUMBER	PatVarDef
PATVARDEF	PatVarUse	PATVARUSE	ptr
Ptr	real	Real	str0
Str	SYNCHPAR	Table	TEXTLINE
Ytextfile	Ytextline		

Appendix B. Syntax of SPARE language

```
specification               ::=  domain-declaration-section
                                 aux-function-section
                                 semantic-func-decl-section
                                 semantic-func-defn-section

domain-declaration-section  ::=  { domain-declaration ; }

domain-declaration          ::=  domain-name { , domain-name }
                                    = domain-definition [ ordering-option ]

domain-definition           ::=  primitive-domain
                              |  enumeration-domain
                              |  lifted-domain
                              |  topped-domain
                              |  product-domain
                              |  union-domain
                              |  powerset-domain
                              |  list-domain
                              |  store-domain
                              |  function-domain

primitive-domain            ::=  boolean | integer | syntactic | label

enumeration-domain          ::=  "{" string-literal { , string-literal } "}"

lifted-domain               ::=  lifted domain-name

topped-domain               ::=  topped domain-name

product-domain              ::=  domain-name * domain-name { * domain-name }

union-domain                ::=  domain-name + domain-name { + domain-name }

powerset-domain             ::=  powerset of domain-name

list-domain                 ::=  list of domain-name
```

store-domain	::=	**store** domain-name --> domain-name
function-domain	::=	**function** [argument-list] --> domain-name
argument-list	::=	domain-name { --> domain-name }
ordering-option	::=	**ordered by** ordering
ordering	::=	order-enumeration
	|	order-name
order-enumeration	::=	(order-tuple { , order-tuple })
order-tuple	::=	< string-literal , string-literal >
aux-function-section	::=	{aux-function}
aux-function	::=	function-name : function-type-declaration **is** expression ;
semantic-func-decl-section	::=	semantic-func-decl { semantic-func-decl }
semantic-func-decl	::=	function-name "[[" domain-name "]]" :
		function-type-declaration [collect-option] [external-option] ;
function-type-declaration	::=	[funarg-list] --> domain-name
funarg-list	::=	[**strict**] domain-name [--> [**strict**] domain-name]
collect-option	::=	**collect** collect-domain
collect-domain	::=	[**powerset of**] selected-domain
selected-domain	::=	domain-name [^ integer]
semantic-func-defn-section	::=	semantic-equation { semantic-equation }
semantic-equation	::=	function-name "[[" (production) "]]" =
		expression
production	::=	production-name { syntactic-domain-name }

```
syntactic-domain-name   ::=  identifier
                          |  identifier$integer-literal

expression              ::=  constant
                          |  variable
                          |  primitive-expression
                          |  lambda-expression
                          |  function-application
                          |  expression-composition
                          |  expression-union
                          |  conditional-expression
                          |  fixed-point-expression
                          |  let-expression
                          |  typecast-expression
                          |  ( expression )

constant                ::=  integer-literal
                          |  boolean-literal
                          |  string-literal
                          |  nil
                          |  bottom
                          |  top

variable                ::=  $
                          |  identifier
                          |  "[[" syntactic-domain-name "]]"
                          |  function-name "[[" syntactic-domain-name "]]"

primitive-expression    ::=  pre-defined-operation
                          |  set-construction
                          |  product-construction

set-construction        ::=  "{" [ expression-list ] "}"

product-construction    ::=  ( [ expression-list ] )

expression-list         ::=  expression { , expression }

pre-defined-operation   ::=  unary-prefix-operation
```

```
                                |    binary-infix-operation
                                |    binary-prefix-operation
                                |    unary-postfix-operation
                                |    mixfix-operation

unary-prefix-operation         ::=   unary-prefix-op expression

binary-infix-operation         ::=   expression ^ integer-literal
                                |    expression is domain-name
                                |    expression binary-infix-op expression

binary-prefix-operation        ::=   binary-prefix-op expression expression

unary-postfix-operation        ::=   identifier ' unary-postfix-op

mix-fix-operation              ::=   expression "[" store-expression "]"

store-expression               ::=   expression
                                |    expression / expression
                                |    foreach identifier in expression do expression / expression

lambda-expression              ::=   lambda identifier . expression

function-application           ::=   expression expression

expression-composition         ::=   expression . expression

expression-union               ::=   expression + expression

conditional-expression         ::=   expression --> expression , expression

fixed-point-expression         ::=   fix [ termination-option ] function-name . expression

termination-option             ::=   argument [ bound-option ]
                                |    cache [ bound-option ]

bound-option                   ::=   bound ( integer-literal )

let-expression                 ::=   let let-clause { ; let-clause } in expression
```

let-clause	::=	identifier = expression
typecast-expression	::=	domain-name expression
unary-prefix-op	::=	**minus** | **not** | **head** | **tail** | **null** | +
binary-infix-op	::=	**and** | **or** | **in** | <= | + | * | - | :: | & | =
binary-prefix-op	::=	**add** | **sub** | **mult** | **div**
unary-postfix-op	::=	**label** | **cache**

Appendix C. Example 1 : Sign analysis of expressions

Domain Declaration:

```
id, exp = syntactic;
Signs = {"+","-","0"} ordered by (<"0","-">,<"0","+">);
Sign_SemiLattice = lifted Signs;
Sign_Lattice = topped Sign_SemiLattice;
Store = store id --> Sign_Lattice;
```

Semantic Function Declaration:

```
E [[exp]] : Store --> Sign_Lattice;
```

Semantic Function Definition:

```
E [[ (Id id) ]] = lambda x. x[ [[id]] ]

E [[ (Add exp$2 exp$3 ) ]] =
                  lambda x. ( E [[ exp$2 ]] + E [[ exp$3 ]] ) x
```

Appendix D. Example 2: Conditional Constant Propagation

(based on the algorithm by Wegbreit [7])

The abstract syntax in SSL:

```
root prog;
prog            :   Prog(stmtlist)
                ;


list stmtlist;
stmtlist        :   StmtListNil()
                |   StmtList(stmt stmtlist)
                ;


stmt            :   StmtNil()
                |   Assign(id exp)
                |   If(b_exp stmt stmt)
                |   While(b_exp stmt)
                |   Comp(stmtlist)
                ;


exp             :   ExpNil()
                |   ExpId(id)
                |   Int_Lit(int)
                |   Add(exp exp)
                ;


b_exp           :   B_ExpNil()
                |   Bool_Lit(bool)
                |   And(b_exp b_exp)
                |   Equal(exp exp)
                ;
```

Conditional Constant Propagation specification:

```
Domain Declarations:
  bool, int, id, exp, b_exp, stmt, stmtlist, prog = syntactic;
  con =  lifted integer;
  Con =  topped con;
  cstore =  store id --> Con;
  Cstore =  lifted cstore;

Semantic Function Declaration:
  P[[prog ]] :  --> Cstore;
  SL[[stmtlist ]] :  strict Cstore$1 --> Cstore$2 collect Cstore$2;
  S[[stmt ]] :  strict Cstore$1 --> Cstore$2 collect Cstore$2;
  E[[exp ]] :  strict Cstore --> Con;
  Bt[[b_exp ]] :  strict Cstore --> Cstore;
  Bf[[b_exp ]] :  strict Cstore --> Cstore;



Semantic Function Definitions:
  P[[(Prog stmtlist )]] =
      let c=SL [[stmtlist]] (bottom[]) in +([[stmtlist]]'cache);

  SL[[(StmtListNil )]] = identity;

  SL[[(StmtList stmt stmtlist$2 )]] = SL [[stmtlist$2]] . S [[stmt]];

  S[[(StmtNil )]] = identity;

  S[[(Assign id exp )]] = lambda x. let n=E [[exp]] x in x[ n/[[id]] ];

  S[[(If b_exp stmt$2 stmt$3 )]] =
      lambda x. let bt=Bt [[b_exp]] x; bf=Bf [[b_exp]] x in
            (S [[stmt$2]] bt + S [[stmt$3]] bf);

  S[[(While b_exp stmt$2 )]] =
      fix argument f. lambda x.
            let bt=Bt [[b_exp]] x; bf=Bf [[b_exp]] x in
```

```
                (f (S [[stmt$2]] bt + x) + bf);

E[[(ExpNil )]] = identity;

E[[(ExpId id )]] = lambda x. x[ [[id]] ];

E[[(Int_Lit int )]] = lambda x. Con [[int]];

E[[(Add exp$2 exp$3 )]] =
    lambda x. let n1=E [[exp$2]] x; n2=E [[exp$3]] x in
          (n1 = top) or (n2 = top) --> top ,add n1 n2;

Bt[[(B_ExpNil )]] = identity;

Bt[[(Bool_Lit bool )]] = lambda x. [[bool]] --> x ,bottom;

Bt[[(And b_exp$2 b_exp$3 )]] =
    lambda x. let bt1=Bt [[b_exp$2]] x; bt2=Bt [[b_exp$3]] x in
          (bt1 = bottom) or (bt2 = bottom) --> bottom ,x;

Bt[[(Equal exp$1 exp$2 )]] =
    lambda x. let n1=E [[exp$1]] x; n2=E [[exp$2]] x in
          (n1 = top) or (n2 = top) or (n1 = n2) --> x ,bottom;

Bf[[(B_ExpNil )]] = identity;

Bf[[(Bool_Lit bool )]] = lambda x. [[bool]] --> bottom ,x;

Bf[[(And b_exp$2 b_exp$3 )]] =
    lambda x. let bf1=Bf [[b_exp$2]] x; bf2=Bf [[b_exp$3]] x in
          (bf1 = bottom) and (bf2 = bottom) --> bottom ,x;

Bf[[(Equal exp$1 exp$2 )]] =
    lambda x. let n1=E [[exp$1]] x; n2=E [[exp$2]] x in
          (not ((n1 = top) or (n2 = top)) and (n1 = n2)) --> bottom ,x;
```

Appendix E. Domain definitions generated for the specification in Appendix D.

```
__userdomains  :  __bottomrep()
               |  __toprep()
               |  __notop()
               |  __nobottom()
               |  __errorvalue()
               |  __intdomain(int)
               |  __booldomain(bool)
               |  __labeldomain(STR)
               |  __syntacticdomain(id)
               |  __functiondomain(PTR)
               |  __cache(__store_impl)
               |  __cstore(__store_impl)
               |  __con(__lifted_domain_impl)
               |  __Cstore(__lifted_domain_impl)
               |  __Con(__topped_domain_impl)
               ;
```

References

1. J. Backus, "Can programming be liberated from the von Neuman style? A functional style and its algebra of programs," *Communications of the ACM* **21** pp. 613-641 (1978).

2. T. Reps and T. Teitelbaum, *The Synthesizer Generator Reference Manual,* Springer-Verlag, New York (Third Edition, 1988).

3. D. A. Schmidt, *Denotational Semantics - A methodology for language development,* Allyn and Bacon, Boston (1986).

4. Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* The MIT Press, Cambridge, Massachusetts, and London, England (1977).

5. R. D. Tennent, "The denotational semantics of programming languages," *Comm. of the ACM* **19** pp. 437-452 (1976).

6. G. A. Venkatesh and C. N. Fischer, "Formal Semantics of SPARE," Tech. Report in preparation, University of Wisconsin-Madison (1989).

7. B. Wegbreit, "Property extraction in well-founded property sets," *IEEE Trans. on Software Engineering* **SE-1**(3) pp. 270-285 (Sept 1975).