SPARE: FORMAL SEMANTICS

by

G A Venkatesh
Charles N. Fischer

Computer Sciences Technical Report #873

August 1989

SPARE : Formal Semantics

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) [9] is a tool for rapid prototyping of program analysis algorithms through high-level specifications. An analysis algorithm is specified through denotational specifications. This report provides the formal semantics for the specification language. 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.

The semantics is presented using a formalism based on Natural Semantics [5]. The semantics specification consists of a set of logical inference rules and facilitates the derivation of proofs for the correctness of translations for SPARE as well as the correctness of analysis algorithms specified in SPARE.

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) [9] 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. Unlike the abstract mathematical notation used to provide denotational semantics, the SPARE language is an applicative language with an underlying evaluation model.

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 specification language is described in [9].

The semantics is presented using a formalism similar to that of Natural Semantics [5]. Natural Semantics has its origin in Plotkin's structural semantics [6] but uses a natural deduction [7] approach rather than an operational approach. Natural Semantics has been used to provide formal semantics for ML [1,4]. Natural Semantics specifications can be used to prove the correctness of translations [2] or used as executable specifications [3].

The formal specification in this report is entirely for descriptive purposes. The notation has been designed for clarity and conciseness rather than for execution. The goal is to express the semantics in a formalism that would facilitate the derivation of correctness proofs for translations for the SPARE language as well as correctness proofs for the analysis algorithms expressed in the SPARE language. There is no claim made as to the completeness of the specification. We have formalized the language definition to an extent that is deemed sufficient for the derivation of correctness proofs mentioned above.

1.1. The structure of the specification

The semantic description is divided into two parts, corresponding to the two phases in the use of the SPARE system. First, the specifications are constructed using the SPARE editor. The editor is syntax-directed and performs static type checking. The output of the editor is a translation that associates functions with productions in the abstract syntax of the language for which the analysis specification is written. The first part of the semantic description deals with *static* semantics and specifies the rules followed by the editor.

The second phase consists of evaluation of functions obtained in the first phase using an interpreter supplied by SPARE. The second part of the semantic description deals with *dynamic* semantics and specifies the evaluation rules.

2. Notation

For readability, we will use a variation of the concrete syntax (Appendix 1) rather than an abstract syntax representation. Some of the terminal symbols have been omitted from the syntax when they do not affect readability. We will also use the same name to denote a nonterminal (in `typewriter font`) as well as the syntactic category (in *italics*) that it belongs to. Hence each nonterminal name in Appendix 1 is also the name of a syntactic category. Multiple occurrences of the same non-terminal in a production will be distinguished with subscripts.

Both static and dynamic semantic descriptions consist of a set of rules of the form

$$\frac{\text{hypothesis}}{\text{conclusion}}$$

The conclusion is a sentence (called a *judgment*) of the form

$$C \vdash \text{phrase} \Rightarrow C'$$

where C (called a *context*) and C′ are semantic objects. phrase is usually a syntactic object. The *judgment* should be read as - "The phrase phrase *implies* C′ in the *context* of C". The operator *implies* is overloaded.

The hypothesis is either empty, in which case the conclusion is an axiom, or it consists of a list of conditions - $c_1, c_2, \cdots, c_n$. The conclusion is a theorem if and only if all the conditions c_i can be derived as theorems. A condition c_i is either a *judgment* or a predicate in first-order logic.

A *judgment* is defined for each nonterminal in the syntax. Corresponding to each production for a nonterminal, a rule is provided with a *judgment* for the right hand side of the production as the *conclusion*. The rules for a *judgment* numbered N will be numbered with the prefix "N.". The *context* for the *conclusion* is the same as the *context* in the judgment for the nonterminal. For convenience, we will use ellipses ($\cdots$) to denote components of the context in a rule if those components are not used in the rule. If some of the components are missing and ellipses are not used, the missing components are assumed to be empty sets.

All variables that occur within a rule must be bound to the same value. Positions of variables whose values are not relevant to the rule are marked by a '_'.

An object (or a class of objects) O with *arity* k will be denoted by $O^{(k)}$ while O^k denotes an object from a k-ary product domain.

$\{E\}_i^j$, $i-1 \le j$ denotes a list $E_i, \cdots, E_j$ where E_i is E with all the unbound subscripts occurring within it bound to i. If $j = i-1$ then the list is empty.

If A is a set then $\mathbf{P}_{fin}(A)$ denotes the set of all finite subsets of A. If A and B are sets then $A \Rightarrow_{fin} B$ denotes the set of finite maps from A to B. Finite maps can be explicitly denoted by

$$\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\}.$$

$\{\}$ denotes the empty map. **Dom** f and **Ran** f denote the domain and range respectively of a finite map f. If f_1 and f_2 are finite maps then f_1 / f_2 denotes a finite map with domain **Dom** $f_1 \cup$ **Dom** f_2 such that

$$f_1/f_2(a) = \begin{cases} f_1(a) & \text{if } a \in \textbf{Dom } f_1 \\ f_2(a) & \text{otherwise} \end{cases}$$

The / operator can be generalized to more than two maps. $\bigcup_{i=1}^{k} f_i$ denotes the finite map $f_k/\cdots/(f_2/f_1)$.

For a union domain $B \cup C$, the injection functions are denoted by $\mathbf{in}B : B \to B \cup C$ and $\mathbf{in}C : C \to B \cup C$. When clear from context, these injection functions will be omitted. If $\alpha \in B \cup C$, the boolean predicates $\mathbf{is}B(\alpha)$ and $\mathbf{is}C(\alpha)$ are used to test for the component to which α belongs.

For an element of a product domain $\alpha \in B \times C$, the components of α can be selected in two ways:

- Named selection : e.g. B **of** α, C **of** α
- Positional selection : e.g. $\alpha\downarrow 1$ and $\alpha\downarrow 2$

Optional objects in syntactic phrases are enclosed in square brackets. Corresponding to these optional objects, a rule may have square brackets over semantic objects and/or predicates. Such a rule stands for several rule instances in which the optional objects (syntactic and semantic) are all present or all absent.

Some of the semantic objects are defined to be isomorphic to (or derived from) syntactic categories. The same names are used in the rules to denote both the syntactic and semantic objects isomorphic to them. The coercion from the syntactic object to the corresponding object is indicated by a change of font (from `typewriter font` to sans-serif).

3. Static Semantics

The collection of rules in this section allows the derivation of a *judgment* of the form

$$DE_{init} \vdash \text{specification} \Rightarrow (SD, FD)$$

as a theorem for all "valid" specifications. DE_{init} is a semantic object that incorporates information about pre-defined objects in the language. SD maps a class of semantic objects called *ASNode* to a class of semantic objects called *TypeCorExpr*. *TypeCorExpr* is derived from the syntactic category of `expression` by resolving overloaded operators. These semantic objects are defined in the next section. The goal of static semantics is to provide the rules necessary to associate abstract syntax nodes with semantic

functions and to ensure that the expressions used in semantic function definitions are type correct.

3.1. Semantic Domains

		Str		String
		NonVar		Nonterminal variables (identifiers)
		VarName		Variable names (identifiers)
β	$\in$	*Bound*		Integers
π	$\in$	*ProdVar*		Production variables (identifiers)
κ	$\in$	*ConName*		Domain constructor names
δ	$\in$	*DomName*		Domain names (identifiers)
ϕ	$\in$	*FuncName*		Function names (identifiers)
ε	$\in$	*TypeCorExpr*		Derived from `expression`
		ColDescr		Isomorphic to `collect-option`
		CurName	=	$\{\$\}$
		AnyType	=	$\{\Gamma\}$
ψ	$\in$	*StrictOpt*	=	$\{\Sigma\} \cup \{\}$
χ	$\in$	*CollectOpt*	=	$\{\Xi\} \cup \{\}$
C	$\in$	*ColDescr*	=	$ColDescr \cup \{\}$
ζ	$\in$	*StrSet*	=	$\mathbf{P}_{fin}(Str)$
V	$\in$	*VarSet*	=	$\mathbf{P}_{fin}(VarName)$
D	$\in$	*DomNameSet*	=	$\mathbf{P}_{fin}(DomName)$
ω	$\in$	*Order*	=	$\mathbf{P}_{fin}(Str \times Str)$
		ArgTest	=	*VarSet*
		CacheTest	=	$\{\Phi\}$
TT	$\in$	*TermTest*	=	$ArgTest \cup CacheTest \cup \{\}$
TO	$\in$	*TermOpt*	=	$TermTest \times bound$
FI	$\in$	*FixInfo*	=	$(FuncName \times TermOpt) \cup \{\}$
$(\delta_1, \cdots, \delta_k)$ or δ^k	$\in$	$DomName^k$		
(κ, δ^k)	$\in$	$CompDom^{(k)}$	=	$\bigcup_{k \geq 0} ConName^{(k)} \times DomName^k$

		ASNode	=	*NonVar* × *ProdVar* × *FuncName*
		EnumDom	=	*StrSet* × *Order*
σ	∈	*DomStruct*	=	*EnumDom* ∪ *CompDom*
τ	∈	*Type*	=	*DomName* ∪ *DomStruct* ∪ *AnyType*
DE	∈	*DomainEnv*	=	*DomName* $\Rightarrow_{fin}$ *DomStruct*
CE	∈	*CollectEnv*	=	*DomName* $\Rightarrow_{fin}$ *Type*
FE	∈	*FuncEnv*	=	*FuncName* $\Rightarrow_{fin}$ *DomStruct*
VE	∈	*VarEnv*	=	(*VarName* ∪ *CurName*) $\Rightarrow_{fin}$ *Type*
FD	∈	*FuncDefn*	=	*FuncName* $\Rightarrow_{fin}$ *TypeCorExpr*
FA	∈	*FuncAttr*	=	*FuncName* $\Rightarrow_{fin}$ *DomNameSet* × *StrictOpt*^{k} × *CollectOpt*^{k} × *ColSet*
SD	∈	*SemEqn*	=	*ASNode* $\Rightarrow_{fin}$ *TypeCorExpr*

$, Σ, Ξ and Γ denote distinguished semantic objects.

The domain *ConName* contains the following elements:

$$\begin{aligned} \mathit{ConName} = {} & \{\text{integer, boolean, syntactic, label}\} \\ & \cup \{\text{lifted, topped, powerset, list, store, cache}\} \\ & \cup \bigcup_{k\geq 1} \{\text{function}^{(k)}\} \cup \bigcup_{k\geq 2} \{\text{product}^{(k)}\} \cup \bigcup_{k\geq 2} \{\text{union}^{(k)}\} \end{aligned}$$

Each element in *ConName* has an arity k as follows:

Arity 0:

integer, boolean, syntactic, label

Arity 1:

lifted, topped, powerset, list

Arity 2:

store, cache

function$^{(k)}$, product$^{(k)}$ and union$^{(k)}$ denote domain constructor names of arity k.

The semantic domain *TypeCorExpr* is derived from `expression` using the following modifications to the syntax description in Appendix 1:

constant ::= integer-literal
| boolean-literal
| string-literal
| **nil**
| **bottom** *Type*
| **top** *Type*
| **bottom**[ ] *Type*

unary-prefix-op ::= **minus** | **not** | **head** | **tail** | **null**
| unary-plus

unary-plus ::= ($+_{set}$ *Type*) | ($+_{store}$ *Type*)

binary-infix-op ::= **and** | **or** | **in** | <= | * | - | :: | &
| (+ *Type*) | (= *Type*)

expression ::= constant
| variable
. . .
| collect-expression
| fix-test-expression

collect-expression ::= collect-descriptor expression

fix-test-expression ::= **fixtest** *FuncName TermTest*
bounded *Type* expression

lambda-expression ::= **lambda** [strict-descriptor]
[collect-descriptor]
identifier . expression

strict-descriptor ::= **strict** $Type_1$ $Type_2$

collect-descriptor ::= **collect** *ColDescr Type*

expression-union ::= expression (⊕ *Type*) expression

fixed-point-expression ::= **fix** function-name . expression

Since *TypeCorExpr* is identical to `expression` except for the above modifications, we will provide explicit coercion rules for only those inference rules that are affected by the above modifications. When an element in `expression` is implicitly coerced in inference rules, it is assumed that any embedded components of the element affected by the above modifications are coerced according to the explicit coercion rules.

3.2. Pre-defined objects

All inferences in the system are made in the context of an initial environment that describes the pre-defined objects.

The primitive domain names are mapped to domain constructors as follows:

DE_{Init} = {integer $\mapsto$ integer, boolean $\mapsto$ boolean,
syntactic $\mapsto$ syntactic, label $\mapsto$ label}

For convenience, we use the same names to denote both the domain names as well as the corresponding domain constructor names.

3.3. Domain declaration section

The domain declaration section is used to associate names with explicit domain definitions. The rules in this section are used to infer a *judgment* of the form:

(J1). $DE_{Init} \vdash$ `domain-declaration-section` $\Rightarrow DE'$

Rule 1.1:

$$\frac{DE_0 = DE_{Init}, \{DE_{i-1} \vdash \texttt{domain-declaration} \Rightarrow DE_i\}_1^k}{DE_{Init} \vdash \{\texttt{domain-declaration}_i\ ;\}_1^k \Rightarrow DE_k}$$

The list of declarations is to be processed sequentially. Each declaration is processed in an environment that includes the declarations occurring before it.

(J2). $DE \vdash$ `domain-declaration` $\Rightarrow DE'$

Rule 2.1:

$$\frac{C_1, C_2, DE \vdash \texttt{domain-definition} \Rightarrow \sigma,\ \sigma \vdash [\ \texttt{ordering-option}\ ] \Rightarrow \sigma'}{DE \vdash \{\texttt{domain-name}_i\}_1^k = \texttt{domain-definition}\ [\ \texttt{ordering-option}\ ] \Rightarrow \{\text{domain-name}_i \mapsto \sigma'\}_1^k / DE}$$

The conditions C_1 and C_2 preclude multiple definitions:

$C_1 \equiv \{\text{domain-name}_i \notin \textbf{Dom}\ DE\}_1^k$

$C_2 \equiv \forall i, j : 1..k.\ i = j \vee \text{domain-name}_i \neq \text{domain-name}_j$

A domain name is not defined until the end of the definition and cannot be used inside the definition recursively.

(J3). $\sigma \vdash [$ `ordering-option` $] \Rightarrow \sigma'$

The domain structure is updated to include the explicit ordering (if any) specified for the domain. The option to specify the ordering through external functions [9] will not be formalized here.

Rule 3.1:

$$\frac{}{\sigma \vdash\ \ \Rightarrow \sigma}$$

Null ordering option.

Rule 3.2:

$$\frac{\textbf{is}\mathit{EnumDom}(\sigma),\ \sigma = (\zeta, \_),\ \{\zeta \vdash \texttt{order-tuple}_i \Rightarrow \omega_i\}_1^k}{\sigma \vdash \textbf{ordered by}\ \{\texttt{order-tuple}_i\}_1^k \Rightarrow \textbf{in}\mathit{EnumDom}(\zeta, \bigcup_1^k \omega_i)}$$

Explicit ordering can be specified for enumeration domains only.

Rule 3.3:

$$\frac{\{\text{string-literal}_i \in \zeta\}_1^2}{\zeta \vdash \texttt{<string-literal}_1\texttt{,string-literal}_2\texttt{>} \Rightarrow \{(\text{string-literal}_1, \text{string-literal}_2)\}}$$

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 $a \leq b$. It is sufficient to provide all the "immediately less than" relations in the required ordering. The ordering through transitivity is implicitly established. There are no checks made to verify that the ordering establishes a partial order in the domain.

(J4). DE $\vdash$ `domain-definition` $\Rightarrow \sigma$

The domain structure is obtained from the definition.

PRIMITIVE DOMAINS:

Rule 4.1:

$$\frac{}{\text{DE} \vdash \textbf{boolean} \Rightarrow \textbf{in}\mathit{CompDom}(\text{boolean})}$$

Rule 4.2:

$$\frac{}{\text{DE} \vdash \textbf{integer} \Rightarrow \textbf{in}\mathit{CompDom}(\text{integer})}$$

Rule 4.3:

$$\frac{}{\text{DE} \vdash \textbf{syntactic} \Rightarrow \textbf{in}\mathit{CompDom}(\text{syntactic})}$$

Rule 4.4:

$$\frac{}{\text{DE} \vdash \textbf{label} \Rightarrow \textbf{in}\mathit{CompDom}(\text{label})}$$

ENUMERATION DOMAIN:

Rule 4.5:

$$\frac{}{\text{DE} \vdash \{\texttt{string-literal}_i\}_1^k \Rightarrow \textbf{in}\mathit{EnumDom}(\bigcup_{i=1}^{k} \{\text{string-literal}_i\}, \{\})}$$

The ordering is assumed to be flat unless the declaration has an explicit ordering specified.

LIFTED DOMAIN:

Rule 4.6:

$$\frac{\text{domain-name} \in \textbf{Dom DE}}{\text{DE} \vdash \textbf{lifted } \texttt{domain-name} \Rightarrow \textbf{in}\mathit{CompDom}(\text{lifted, domain-name})}$$

TOPPED DOMAIN:

Rule 4.7:

$$\frac{\text{domain-name} \in \textbf{Dom DE}}{\text{DE} \vdash \textbf{topped } \texttt{domain-name} \Rightarrow \textbf{in}\mathit{CompDom}(\text{topped, domain-name})}$$

PRODUCT DOMAIN:

Rule 4.8:

$$\frac{\{\text{domain-name}_i \in \textbf{Dom DE}\}_1^k}{\text{DE} \vdash \texttt{domain-name}_1\ \{\ *\ \texttt{domain-name}_i\}_2^k \Rightarrow \textbf{in}\mathit{CompDom}(\text{product}^{(k)}, \{\text{domain-name}_i\}_1^k)}$$

UNION DOMAIN:

Rule 4.9:

$$\frac{\{\text{domain-name}_i \in \textbf{Dom DE}\}_1^k}{\text{DE} \vdash \texttt{domain-name}_1\ \{\ +\ \texttt{domain-name}_i\}_2^k \Rightarrow \textbf{in}\mathit{CompDom}(\text{union}^{(k)}, \{\text{domain-name}_i\}_1^k)}$$

POWERSET DOMAIN:

Rule 4.10:

$$\frac{\text{domain-name} \in \textbf{Dom DE}}{\text{DE} \vdash \textbf{powerset of}\ \texttt{domain-name} \Rightarrow \textbf{in}\mathit{CompDom}(\text{powerset, domain-name})}$$

LIST DOMAIN:

Rule 4.11:

$$\frac{\text{domain-name} \in \textbf{Dom DE}}{\text{DE} \vdash \textbf{list of}\ \texttt{domain-name} \Rightarrow \textbf{in}\mathit{CompDom}(\text{list, domain-name})}$$

STORE DOMAIN:

Rule 4.12:

$$\frac{\{\text{domain-name}_i \in \textbf{Dom DE}\}_1^2}{\text{DE} \vdash \textbf{store}\ \texttt{domain-name}_1\ \texttt{-->}\ \texttt{domain-name}_2 \Rightarrow \textbf{in}\mathit{CompDom}(\text{store, (domain-name}_1\text{,domain-name}_2\text{))}}$$

FUNCTION DOMAIN:

Rule 4.13:

$$\frac{}{\text{DE} \vdash \textbf{function}\ \{\texttt{domain-name}_i\}_1^k \Rightarrow \textbf{in}\mathit{CompDom}(\text{function}^{(k)}, (\{\text{domain-name}_i\}_1^k))}$$

The editor/compiler must ensure that the domain names used in a function declaration are eventually declared in the domain declaration section. The rules given above can be extended to include this requirement. However, for simplicity, we will leave it stated informally.

3.4. Auxiliary functions

Auxiliary functions are used to abstract operations used in expressions. The rules in this section are used to infer the following kind of *judgment*:

(J5). DE $\vdash$ `aux-function-section` $\Rightarrow$ (FE, FD)

Function names are mapped to function type descriptors in FE and to type checked expressions that form the body of the functions in FD.

Rule 5.1:

$$\frac{FE_0 = \{\},\ \{(DE, FE_{i-1}) \vdash \texttt{aux-function}_i \Rightarrow (FE_i, FD_i)\}_1^k}{DE \vdash \{\texttt{aux-function}_i\}_1^k \Rightarrow (FE_k, \bigcup_{i=1}^{k} FD_i)}$$

The list of auxiliary functions are processed sequentially.

(J6). (DE, FE) $\vdash$ `aux-function` $\Rightarrow$ (FE′, FD)

Rule 6.1:

$$\frac{\begin{array}{c} C_1,\ (DE, \{\}) \vdash \texttt{function-type-declaration} \Rightarrow (\sigma, \psi^k, \chi^k), \\ (DE, FE, \psi^k, \chi^k, \textbf{in}\mathit{DomStruct}(\sigma)) \vdash \texttt{expression} \Rightarrow \tau \end{array}}{\begin{array}{r} (DE, FE) \vdash \texttt{function-name : function-type-declaration} \\ \textbf{is expression ;} \Rightarrow (\{\textbf{function-name}\} \mapsto \sigma\}/FE,\ \{\text{function-name} \mapsto \text{expression}\}) \end{array}}$$

The condition C_1 precludes re-definition of names.

$C_1 \equiv$ function-name $\notin$ (**Dom** DE $\cup$ **Dom** FE)

The rules for `function-type-declaration` is in Section 3.6. The function type declaration provides the domain structure (functionality), the strictness descriptor and the collect option descriptor for the function. The collect option cannot be used in an auxiliary function type declaration. This restriction is enforced syntactically. The rules for `expression` are in Section 3.8.

The expression is processed to ensure that it is type compatible with the function type declaration. The function name is not defined until the end of the expression. Recursive functions can be defined using the fixed point expressions.

3.5. Semantic function declarations

The semantic function declaration section associates function names with syntactic objects and declares their functionality. The *judgment* to be inferred is of the form:

(J7). (DE, FE) $\vdash$ `semantic-func-decl-section` $\Rightarrow$ (FA, FE′, CE)

FA maps function names to syntactic domain names, strictness description and collect information (if the collect option is used). The function environment FE′ is FE augmented with the mapping between semantic function names and their function type descriptors. CE maps the syntactic domain name to the type of the collect domain.

Rule 7.1:

$$\frac{\begin{array}{c}FE_0 = FE,\ CE_0 = CE,\\ \{(DE, FE_{l-1}, CE_{l-1}) \vdash \texttt{semantic-func-decl}_l \Rightarrow (FA_l, FE_l, CE_l)\}_1^k\end{array}}{(DE, FE) \vdash \{\texttt{semantic-func-decl}_l\}_1^k \Rightarrow (\bigcup_{l=1}^{k} FA_l, FE_k, CE_k)}$$

The semantic function declarations are processed sequentially.

(J8). (DE, FE, CE) $\vdash$ `semantic-func-decl` $\Rightarrow$ (FA, FE′, CE′)

Rule 8.1:

$$\frac{\begin{array}{c}C_1, C_2, C_3, DE \vdash \texttt{[collect-option]} \Rightarrow (D, \tau),\\ (DE, D) \vdash \texttt{function-type-declaration} \Rightarrow (\sigma, \psi^k, \chi^k)\end{array}}{\begin{array}{r}(DE, FE, CE) \vdash \texttt{function-name[[domain-name]]} :\\ \texttt{function-type-declaration [collect-option];} \Rightarrow\\ (\{\text{function-name} \mapsto (\{\text{domain-name}\}, \psi^k, \chi^k, [\text{collect-option}])\},\\ \{\text{function-name} \mapsto \sigma\}/FE, \{\text{domain-name} \mapsto \tau\})\end{array}}$$

$C_1 \equiv DE(\text{domain-name}) = \text{syntactic}$

$C_2 \equiv \text{function-name} \notin (\textbf{Dom}\ DE \cup \textbf{Dom}\ FE)$

$C_3 \equiv (\text{domain-name} \notin \textbf{Dom}\ CE) \vee (DE \vdash CE(\text{domain-name}) \leftrightarrow \tau \Rightarrow \tau')$

The condition C_1 ensures that the semantic function is associated with a syntactic domain. The condition C_2 precludes re-definition of the identifier used for the function name. The condition C_3 ensures that collect options on multiple functions associated with the same syntactic object specify collection of values from type compatible domains. The *judgment* for type compatibility used in C_3 is defined in Section 3.8.1. The external option has been left out since the static semantics is not affected by its use.

(J9). DE $\vdash$ `[collect-option]` $\Rightarrow$ (D, τ)

The collect option is processed to provide the domain name from which the value for collection is to be obtained as well as the type of the value that will be collected.

Rule 9.1:

$$\frac{}{\text{DE} \vdash \quad \Rightarrow (\{\}, \textbf{in}\mathit{AnyType}(\Gamma))}$$

Rule 9.2:

$$\frac{\text{DE} \vdash \texttt{selected-domain} \Rightarrow (\text{D}, \delta)}{\text{DE} \vdash \texttt{selected-domain} \Rightarrow (\text{D}, \textbf{in}\mathit{DomName}(\delta))}$$

Rule 9.3:

$$\frac{\text{DE} \vdash \texttt{selected-domain} \Rightarrow (\text{D}, \delta)}{\text{DE} \vdash \textbf{powerset of } \texttt{selected-domain} \Rightarrow (\text{D}, \textbf{in}\mathit{DomStruct}(\text{powerset}, \delta))}$$

(J10). $\text{DE} \vdash \texttt{selected-domain} \Rightarrow (\text{D}, \delta)$

Rule 10.1:

$$\frac{\text{domain-name} \in \text{DE}}{\text{DE} \vdash \texttt{domain-name} \Rightarrow (\{\text{domain-name}\}, \text{domain-name})}$$

Rule 10.2:

$$\frac{\text{domain-name} \in \text{DE},\ \text{DE}(\text{domain-name}) = (\text{product}^{(k)}, \delta^k),\ 1 \le \text{integer} \le k}{\text{DE} \vdash \texttt{domain-name ^ integer} \Rightarrow (\{\text{domain-name}\}, \delta^k \downarrow \text{integer})}$$

3.6. Function type declaration

The rules in this section are used to derive function descriptors from function type declarations. The *judgment* to be inferred is of the form:

(J11). $(\text{DE,D}) \vdash \texttt{function-type-declaration} \Rightarrow (\sigma, \psi^k, \chi^k)$

D is a singleton set containing a domain name from possible use of the collect option. If the collect option was not used or the function type declaration was for an auxiliary function, then D is empty.

Rule 11.1:

$$\frac{\begin{array}{c}C_1,\ C_2,\ \{\ \text{D} \vdash [\textbf{strict}]\ \texttt{domain-name}_i \Rightarrow (\psi_i, \chi_i)\}_1^k, \\ \text{D} \vdash \texttt{domain-name}_{k+1} \Rightarrow (\psi_{k+1}, \chi_{k+1})\end{array}}{\begin{array}{l}(\text{DE, D}) \vdash \{[\textbf{strict}]\ \texttt{domain-name}_i\}_1^k \texttt{ --> domain-name}_{k+1} \Rightarrow \\ (\textbf{in}\mathit{CompDom}(\text{function}^{(k+1)}, \{\text{domain-name}_i\}_1^{k+1}), \{\psi_i\}_1^{k+1}, \{\psi_i\}_1^{k+1})\end{array}}$$

$C_1 \equiv \{\text{domain-name}_i \in \textbf{Dom DE}\}_1^{k+1}$

$C_2 \equiv \exists i : 1..k+1.\ \text{domain-name}_i \in \text{D}$

The condition C_1 ensures that the argument domains are defined. The condition C_2 ensures that the the domain name in the collect option is the name of one of the argument domains (or the result domain).

The editor/compiler must ensure that multiple occurrences of the same domain name are distinguished with suitable suffixes [9]. We will not formalize this condition but assume that multiple occurrences have already been distinguished.

(J12). $D \vdash$ **[strict]** `domain-name` $\Rightarrow (\psi, \chi)$

This *judgment* is used to mark positions of arguments that are designated to be strict and/or specified in the collect option.

Rule 12.1:

$$\frac{\text{domain-name} \notin D}{D \vdash \textbf{[strict]}\ \texttt{domain-name} \Rightarrow ([\Sigma], \{\})}$$

Rule 12.2:

$$\frac{\text{domain-name} \in D}{D \vdash \textbf{[strict]}\ \texttt{domain-name} \Rightarrow ([\Sigma], \{\Xi\})}$$

3.7. Semantic function definitions

Semantic functions are defined through a list of *semantic equations.* A semantic equation declares the function for a single production in the syntax for the syntactic object with which the function is associated. The rules in this section are used to infer a *judgment* of the form:

(J13). $(DE, FE, FA, CE) \vdash$ `semantic-func-defn-section` $\Rightarrow SD$

Rule 13.1:

$$\frac{SD_0 = \{\},\ \{(DE, FE, FA, SD_{i-1}, CE) \vdash \texttt{semantic-equation}_i \Rightarrow SD_i\}_1^k}{(DE, FE, FA, CE) \vdash \{\texttt{semantic-equation}_i\}_1^k \Rightarrow SD_k}$$

The semantic function definitions are processed sequentially.

(J14). $(DE, FE, FA, SD, CE) \vdash$ `semantic-equation` $\Rightarrow SD'$

Rule 14.1:

$$\frac{\begin{array}{c} C_1,\ C_2,\ FA(\text{function-name}) = (D, \psi^k, \chi^k, C),\ DE \vdash \texttt{production} \Rightarrow (\pi, D'), \\ \delta \in D,\ VE = \{\$ \mapsto \textbf{in}\mathit{DomName}(\delta)\},\ \tau' = \textbf{in}\mathit{DomStruct}(FE(\text{function-name})), \\ (DE, FE, VE, CE, C, D', \psi^k, \chi^k, \tau') \vdash \texttt{expression} \Rightarrow \tau \end{array}}{\begin{array}{l} ((DE, FE, FA, SD) \vdash \texttt{function-name [[(production)]] = expression ;} \\ \qquad \Rightarrow \{(\delta, \pi, \text{function-name}) \mapsto \text{expression}\}/SD \end{array}}$$

$C_1 \equiv$ function-name $\in$ **Dom** FE

$C_2 \equiv (\delta, \pi, \text{function-name}) \notin$ **Dom** SD

The condition C_1 ensures that the semantic function has been declared in the semantic function declaration section. The condition C_2 ensures that there are no multiple definitions for the same production. D′ contains the names of syntactic objects listed in the production. The expression that forms the body of the semantic function can use only these syntactic object names.

(J15). DE $\vdash$ `production` $\Rightarrow$ (π, D)

Rule 15.1:

$$\frac{\{\text{syntactic-domain-name}_i \in \textbf{Dom DE}\}_1^k,\ \{\text{DE}(\text{syntactic-domain-name}_i) = \text{syntactic}\}_1^k}{\text{DE} \vdash \texttt{production-name}\ \{\texttt{syntactic-domain-name}_i\}_1^k \Rightarrow (\text{production-name}, \bigcup_{i=1}^{k}\{\text{syntactic-domain-name}_i\})}$$

3.8. Expressions

This section provides the rules for static type checking of expressions. The rules can be used to infer a *judgment* of the form:

(J16). (DE, FE, VE, CE, FI, C, D, ψ^k, χ^k, τ) $\vdash$ expression $\Rightarrow$ τ'

The rules ensure that the type of the expression τ' is equivalent to the expected type τ. τ is determined from the context in which the expression occurs. Name equivalence is used for all domains except function domains. Structural equivalence is used for function domains. In addition, a cache domain is equivalent to all store domains that are structurally equivalent to it.

The following coercion rules must be used for all expressions:

Coercion rule:

$$\frac{\text{DE} \vdash \tau \Rightarrow (\text{function}^{(1)}, \delta),\ \text{FI} = (\phi, (\text{TT}, \beta))}{(\text{DE}, \text{FI}, \cdots) \vdash \texttt{expression} \Rightarrow \textbf{fixtest}\ \phi\ \text{TT}\ \beta\ \textbf{in}\textit{DomName}(\delta)\ \text{expression}}$$

This rule is used when the expression is the body of a fixed-point expression after the λs corresponding to the arguments to the function have been stripped. The **fixtest** prefix is inserted so that during evaluation the termination tests can be made before the expression is evaluated. We will assume that the body of a fixed point expression is in a normal form where all the λs corresponding to the arguments of the function are to the left of the body.

Coercion rule:

$$\frac{\mathsf{DE} \vdash \tau \Rightarrow (\mathsf{function}^{(1)}, \delta), \chi^k\downarrow 1 = \Xi}{(\mathsf{DE}, \mathsf{VE}, \mathsf{CE}, \mathsf{C}, \chi^k, \cdots) \vdash \texttt{expression} \Rightarrow \textbf{collect C CE(VE(\$))} \text{ expression}}$$

This rule is used when the expression forms the body of a semantic function and its value is to be collected.

If both coercion rules apply, then the collect coercion must be applied first.

3.8.1. Type equivalence

To formalize the type equivalence rules, we use the following variant form of a *judgment*:

(J17). $\mathsf{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3$

The rules ensure that if τ_1 and τ_2 are equivalent types, then τ_3 is equivalent to both τ_1 and τ_2.

Rule 17.1:

$$\frac{}{\mathsf{DE} \vdash \delta \leftrightarrow \delta \Rightarrow \textbf{in}\mathit{DomName}(\delta)}$$

The domains are name equivalent.

Rule 17.2:

$$\frac{\mathsf{DE} \vdash \textbf{in}\mathit{DomName}(\mathsf{DE}(\delta_1)) \leftrightarrow \textbf{in}\mathit{DomName}(\mathsf{DE}(\delta_2)) \Rightarrow \tau}{\mathsf{DE} \vdash \delta_1 \leftrightarrow \delta_2 \Rightarrow \tau}$$

Two domains are type equivalent if their structures are type equivalent. Rules are provided below for structure equivalences of function and cache domains. As the structure equivalence of other domains are not defined, the structural equivalence is used for only function and cache domains.

Rule 17.3:

$$\frac{\mathsf{DE} \vdash \textbf{in}\mathit{DomStruct}(\mathsf{DE}(\delta)) \leftrightarrow \textbf{in}\mathit{DomStruct}(\sigma) \Rightarrow \tau}{\mathsf{DE} \vdash \delta \leftrightarrow \sigma \Rightarrow \tau}$$

Rule 17.4:

$$\frac{\mathsf{DE} \vdash \textbf{in}\mathit{DomName}(\delta) \leftrightarrow \textbf{in}\mathit{DomStruct}(\sigma) \Rightarrow \tau}{\mathsf{DE} \vdash \sigma \leftrightarrow \delta \Rightarrow \tau}$$

Rule 17.5:

$$\frac{\sigma = (\mathsf{function}^{(1)}, \delta'), \mathsf{DE} \vdash \textbf{in}\mathit{DomName}(\delta) \leftrightarrow \textbf{in}\mathit{DomName}(\delta') \Rightarrow \tau}{\mathsf{DE} \vdash \delta \leftrightarrow \sigma \Rightarrow \tau}$$

Constant function domains are compatible with their range domains.

Rule 17.6:

$$\frac{\tau \neq \Gamma}{\mathsf{DE} \vdash \mathbf{in}\mathit{AnyType}(\Gamma) \leftrightarrow \tau \Rightarrow \tau}$$

Rule 17.7:

$$\frac{\tau \neq \Gamma}{\mathsf{DE} \vdash \tau \leftrightarrow \mathbf{in}\mathit{AnyType}(\Gamma) \Rightarrow \tau}$$

Γ is used to denote a type that is compatible with any type. However, as all expressions in the language must be statically typed with unique types, the above rules specify that Γ is compatible with all types except itself. Hence in type compatibility checks at least one of the types must always be uniquely determined.

Rule 17.8:

$$\frac{\{\sigma_i = (\mathsf{function}^{(k)}, \delta_i^k)\}_1^2,\ \delta_1^k = \delta_2^k}{\mathsf{DE} \vdash \sigma_1 \leftrightarrow \sigma_2 \Rightarrow \mathbf{in}\mathit{DomStruct}(\sigma_1)}$$

Function domains are equivalent if their functionalities are the same.

Rule 17.9:

$$\frac{\sigma_1 = (\mathsf{cache}, \delta_1^2),\ \sigma_2 = (\mathsf{store}, \delta_2^2),\ \delta_1^2 = \delta_2^2}{\mathsf{DE} \vdash \sigma_1 \leftrightarrow \sigma_2 \Rightarrow \mathbf{in}\mathit{DomStruct}(\sigma_2)}$$

Rule 17.10:

$$\frac{\sigma_2 = (\mathsf{cache}, \delta_2^2),\ \sigma_1 = (\mathsf{store}, \delta_1^2),\ \delta_1^2 = \delta_2^2}{\mathsf{DE} \vdash \sigma_1 \leftrightarrow \sigma_2 \Rightarrow \mathbf{in}\mathit{DomStruct}(\sigma_1)}$$

A cache domain is equivalent to any store domain with the same structure.

The following auxiliary *judgments* are used in type rules for expressions.

(J18). $\mathsf{DE} \vdash \tau_1 \leftarrow \tau_2 \Rightarrow \tau_3$

The rules ensure that if τ_1 is a domain obtained by using lifted and topped domain constructors (zero or more times) on τ_2, then τ_3 is equivalent to the domain τ_1.

Rule 18.1:

$$\frac{\mathsf{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \_}{\mathsf{DE} \vdash \tau_1 \leftarrow \tau_2 \Rightarrow \tau_1}$$

Rule 18.2:

$$\frac{\mathsf{DE} \vdash \mathbf{in}\mathit{DomStruct}(\mathsf{DE}(\delta_1)) \leftarrow \mathbf{in}\mathit{DomStruct}(\mathsf{DE}(\delta_2)) \Rightarrow \_}{\mathsf{DE} \vdash \delta_1 \leftarrow \delta_2 \Rightarrow \mathbf{in}\mathit{DomName}(\delta_1)}$$

Rule 18.3:

$$\frac{\mathsf{DE} \vdash \mathbf{in}\mathit{DomStruct}(\mathsf{DE}(\delta)) \leftarrow \mathbf{in}\mathit{DomStruct}(\sigma) \Rightarrow \_}{\mathsf{DE} \vdash \delta \leftarrow \sigma \Rightarrow \mathbf{in}\mathit{DomName}(\delta)}$$

Rule 18.4:

$$\frac{\mathsf{DE} \vdash \mathbf{in}\mathit{DomStruct}(\sigma) \leftarrow \mathbf{in}\mathit{DomStruct}(\mathsf{DE}(\delta)) \Rightarrow \_}{\mathsf{DE} \vdash \sigma \leftarrow \delta \Rightarrow \mathbf{in}\mathit{DomStruct}(\sigma)}$$

Rule 18.5:

$$\frac{\sigma_1 = (\mathsf{lifted},\delta),\ \mathsf{DE} \vdash \mathbf{in}\mathit{DomName}(\delta) \leftarrow \mathbf{in}\mathit{DomStruct}(\sigma_2) \Rightarrow \_}{\mathsf{DE} \vdash \sigma_1 \leftrightarrow \sigma_2 \Rightarrow \mathbf{in}\mathit{DomStruct}(\sigma_1)}$$

Rule 18.6:

$$\frac{\sigma_1 = (\mathsf{topped},\delta),\ \mathsf{DE} \vdash \mathbf{in}\mathit{DomName}(\delta) \leftarrow \mathbf{in}\mathit{DomStruct}(\sigma_2) \Rightarrow \_}{\mathsf{DE} \vdash \sigma_1 \leftrightarrow \sigma_2 \Rightarrow \mathbf{in}\mathit{DomStruct}(\sigma_1)}$$

(J19). $\mathsf{DE} \vdash \tau \Rightarrow \sigma$

σ is the domain structure of τ.

Rule 19.1:

$$\frac{}{\mathsf{DE} \vdash \sigma \Rightarrow \sigma}$$

Rule 19.2:

$$\frac{\delta \in \mathbf{Dom}\ \mathsf{DE},\ \mathsf{DE}(\delta) = \sigma}{\mathsf{DE} \vdash \delta \Rightarrow \sigma}$$

3.8.2. Constants

Rule 16.3:

$$\frac{\mathsf{DE} \vdash \tau \leftrightarrow \mathbf{in}\mathit{CompDom}(\mathsf{integer}) \Rightarrow \tau'}{(\mathsf{DE}, \tau, \cdots) \vdash \texttt{integer-literal} \Rightarrow \tau'}$$

Rule 16.4:

$$\frac{\mathsf{DE} \vdash \tau \leftrightarrow \mathbf{in}\mathit{CompDom}(\mathsf{boolean}) \Rightarrow \tau'}{(\mathsf{DE}, \tau, \cdots) \vdash \texttt{boolean-literal} \Rightarrow \tau'}$$

Rule 16.5:

$$\frac{\mathsf{DE} \vdash \tau \Rightarrow \sigma,\ \mathsf{is}\mathit{EnumDom}(\sigma),\ \mathsf{string\text{-}literal} \in \mathit{StrSet}\ \mathbf{of}\ \sigma}{(\mathsf{DE}, \tau, \cdots) \vdash \texttt{string-literal} \Rightarrow \tau}$$

Rule 16.6:

$$\frac{\mathsf{DE} \vdash \tau \Rightarrow \sigma,\ \mathsf{is}\mathit{CompDom}(\sigma),\ \mathit{ConName}\ \mathbf{of}\ \sigma = \mathsf{list}}{(\mathsf{DE}, \tau, \cdots) \vdash \mathbf{nil} \Rightarrow \tau}$$

Rule 16.7:

$$\frac{\tau \neq \Gamma}{(\tau, \cdots) \vdash \mathbf{bottom} \Rightarrow \tau}$$

Coercion rule:

$$\frac{}{(\tau, \cdots) \vdash \mathbf{bottom} \Rightarrow \mathbf{bottom}\ \tau}$$

Rule 16.8:

$$\frac{\tau \neq \Gamma}{(\tau, \cdots) \vdash \mathbf{top} \Rightarrow \tau}$$

Coercion rule:

$$\frac{}{(\tau, \cdots) \vdash \mathbf{top} \Rightarrow \mathbf{top}\ \tau}$$

Rule 16.9:

$$\frac{\vdash \tau \leftarrow (\mathsf{powerset}, \delta)}{(\tau, \cdots) \vdash \mathbf{bottom}[\,] \Rightarrow \tau}$$

Coercion rule:

$$\frac{}{(\cdots) \vdash \mathbf{bottom}[\,] \Rightarrow \mathbf{bottom}\ \mathbf{in}\mathit{DomName}(\delta)}$$

bottom and **top** must occur in only those contexts in which the expected type can be uniquely determined.

3.8.3. Variables

Rule 16.10:

$$\frac{\$ \in \mathbf{Dom}\ \mathsf{VE},\ \mathsf{DE} \vdash \mathsf{VE}(\$) \leftrightarrow \tau \Rightarrow \tau'}{(\mathsf{DE}, \mathsf{VE}, \tau, \cdots) \vdash \$ \Rightarrow \tau'}$$

As $ is defined in VE for only semantic function definitions (Rule 14.1), this rule ensures that $ is used in expressions that define semantic functions.

Rule 16.11:

$$\frac{\text{identifier} \in \textbf{Dom VE, DE} \vdash \text{VE(identifier)} \leftrightarrow \tau \Rightarrow \tau'}{(\text{DE, VE}, \tau, \cdots) \vdash \texttt{identifier} \Rightarrow \tau'}$$

Rule 16.12:

$$\frac{\text{identifier} \in \textbf{Dom FE, DE} \vdash \text{FE(identifier)} \leftrightarrow \tau \Rightarrow \tau', \psi^k = \{\}^k, \chi^k = \{\}^k}{(\text{DE, FE}, \psi^k, \chi^k, \tau, \cdots) \vdash \texttt{identifier} \Rightarrow \tau'}$$

Named functions can appear in only those contexts in which a function with no strict and/or collect options specified is expected. Named functions can be used in other contexts by introducing lambda bindings surrounding them.

Rule 16.13:

$$\frac{\text{syntactic–domain–name} \in \text{D, DE} \vdash \textbf{in}\textit{CompDom}\text{(syntactic)} \leftrightarrow \tau \Rightarrow \tau'}{(\text{DE, D}, \tau, \cdots) \vdash [\![\ \texttt{syntactic–domain–name}\ ]\!] \Rightarrow \tau'}$$

Rule 16.14:

$$\frac{\begin{array}{c}\text{function–name} \in \textbf{Dom FE}, \text{syntactic–domain–name} \in \text{D},\\ \text{syntactic–domain–name} \in \textit{DomNameSet} \textbf{ of FA(function–name)},\\ \text{DE} \vdash \text{FE(function–name)} \leftrightarrow \tau \Rightarrow \tau', \psi^k = \{\}^k, \chi^k = \{\}^k\end{array}}{\begin{array}{l}(\text{DE, FE, D}, \psi^k, \chi^k, \tau, \cdots) \vdash \texttt{function–name[[syntactic–domain–name]]}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \tau'\end{array}}$$

3.8.4. Primitive expressions

SET CONSTRUCTION:

Rule 16.15:

$$\frac{\begin{array}{c}\text{DE} \vdash \tau \Rightarrow \text{(powerset,}\delta\text{)},\\ \{(\text{DE, }\textbf{in}\textit{DomName}(\delta), \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i, \tau_i \neq \Gamma\}_1^k\end{array}}{(\text{DE}, \tau, \cdots) \vdash \{\ \{\texttt{expression}_i\}_1^k\ \} \Rightarrow \tau}$$

Set construction can only occur in places where the type can be inferred from the context as a powerset domain (possibly through explicit type casting). The type of each expression in the set construction must be uniquely determinable and must be compatible with the base domain of the expected powerset domain.

PRODUCT CONSTRUCTION:

Rule 16.16:

$$\frac{\mathsf{DE} \vdash \tau \Rightarrow (\mathsf{product}^{(n)}, \delta^n),\quad \{(\mathsf{DE}, \mathbf{in}\mathit{DomName}(\delta^n{\downarrow}i), \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i, \tau_i \neq \Gamma\}_1^k}{(\mathsf{DE}, \tau, \cdots) \vdash (\{\texttt{expression}_i\}_1^k) \Rightarrow \tau}$$

The product construction can only occur in places where the type can be inferred from the context as a product domain. The type of each expression in the product construction must be uniquely determinable and must be compatible with the corresponding component domain of the expected product domain.

UNARY PREFIX OPERATIONS:

Rule 16.17:

$$\frac{(\mathsf{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression} \Rightarrow \tau_2,\quad \mathsf{DE} \vdash \tau_2 \leftarrow \mathbf{in}\mathit{CompDom}(\mathsf{integer}) \Rightarrow \tau_3,\ \mathsf{DE} \vdash \tau_1 \leftrightarrow \tau_3 \Rightarrow \tau}{(\mathsf{DE}, \tau_1, \cdots) \vdash \mathbf{minus}\ \texttt{expression} \Rightarrow \tau}$$

Rule 16.18:

$$\frac{(\mathsf{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression} \Rightarrow \tau_2,\quad \mathsf{DE} \vdash \tau_2 \leftarrow \mathbf{in}\mathit{CompDom}(\mathsf{boolean}) \Rightarrow \tau_3,\ \mathsf{DE} \vdash \tau_1 \leftrightarrow \tau_3 \Rightarrow \tau}{(\mathsf{DE}, \tau_1, \cdots) \vdash \mathbf{not}\ \texttt{expression} \Rightarrow \tau}$$

Rule 16.19:

$$\frac{(\mathsf{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression} \Rightarrow \tau_2,\quad \mathsf{DE} \vdash \tau_2 \Rightarrow (\mathsf{list}, \delta),\ \mathsf{DE} \vdash \tau_1 \leftrightarrow \mathbf{in}\mathit{DomName}(\delta) \Rightarrow \tau}{(\mathsf{DE}, \tau_1, \cdots) \vdash \mathbf{head}\ \texttt{expression} \Rightarrow \tau}$$

Rule 16.20:

$$\frac{(\mathsf{DE}, \tau_1, \cdots) \vdash \texttt{expression} \Rightarrow \tau_2,\ \mathsf{DE} \vdash \tau_2 \Rightarrow (\mathsf{list}, \_)}{(\mathsf{DE}, \tau_1, \cdots) \vdash \mathbf{tail}\ \texttt{expression} \Rightarrow \tau_2}$$

Rule 16.21:

$$\frac{(\mathsf{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression} \Rightarrow \tau_2,\ \mathsf{DE} \vdash \tau_2 \Rightarrow (\mathsf{list}, \_),\quad \mathsf{DE} \vdash \tau_1 \leftrightarrow \mathbf{in}\mathit{CompDom}(\mathsf{boolean}) \Rightarrow \tau_2}{(\mathsf{DE}, \tau_1, \cdots) \vdash \mathbf{null}\ \texttt{expression} \Rightarrow \tau_2}$$

Rule 16.22:

$$\frac{(\mathsf{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression} \Rightarrow \tau_2,\quad \mathsf{DE} \vdash \tau_2 \Rightarrow (\mathsf{powerset}, \delta),\ \mathsf{DE} \vdash \tau_1 \leftrightarrow \mathbf{in}\mathit{DomName}(\delta) \Rightarrow \tau}{(\mathsf{DE}, \tau_1, \cdots) \vdash +\ \texttt{expression} \Rightarrow \tau}$$

Coercion rule:

$$\frac{}{(\mathrm{DE}, \tau, \cdots) \vdash + \ \texttt{expression} \Rightarrow (+_{\mathrm{set}}\, \tau)\ \texttt{expression}}$$

Rule 16.23:

$$\frac{(\mathrm{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression} \Rightarrow \tau_2,\quad \mathrm{DE} \vdash \tau_2 \Rightarrow (\mathrm{store}, (\_, \delta_2)),\ \mathrm{DE} \vdash \tau_1 \leftrightarrow \mathbf{in}\mathit{DomName}(\delta_2) \Rightarrow \tau}{(\mathrm{DE}, \tau_1, \cdots) \vdash + \ \texttt{expression} \Rightarrow \tau}$$

Coercion rule:

$$\frac{}{(\mathrm{DE}, \tau, \cdots) \vdash + \ \texttt{expression} \Rightarrow (+_{\mathrm{store}}\, \tau)\ \texttt{expression}}$$

BINARY INFIX OPERATIONS:

Rule 16.24:

$$\frac{(\mathrm{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression} \Rightarrow \tau_2,\ \mathrm{DE} \vdash \tau_2 \Rightarrow (\mathrm{product}^{(k)}, \delta^k),\quad \mathrm{DE} \vdash \tau_1 \leftrightarrow \mathbf{in}\mathit{DomName}(\delta^k{\downarrow}\mathrm{integer\text{-}literal}) \Rightarrow \tau}{(\mathrm{DE}, \tau_1, \cdots) \vdash \texttt{expression ^ integer-literal} \Rightarrow \tau}$$

Rule 16.25:

$$\frac{(\mathrm{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression} \Rightarrow \tau_2,\ \mathrm{DE} \vdash \tau_2 \Rightarrow (\mathrm{union}^{(k)}, \delta^k),\quad \exists i.\ \delta^k{\downarrow}i = \mathrm{domain\text{-}name},\ \mathrm{DE} \vdash \tau_1 \leftrightarrow \mathbf{in}\mathit{CompDom}(\mathrm{boolean}) \Rightarrow \tau}{(\mathrm{DE}, \tau_1, \cdots) \vdash \texttt{expression is domain-name} \Rightarrow \tau}$$

Rule 16.26:

$$\frac{\{(\mathrm{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathrm{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\quad \mathrm{DE} \vdash \tau_3 \leftarrow \mathbf{in}\mathit{CompDom}(\mathrm{boolean}) \Rightarrow \tau_4,\ \mathrm{DE} \vdash \tau_4 \leftrightarrow \tau \Rightarrow \tau_5}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1\ \mathbf{and}\ \texttt{expression}_2 \Rightarrow \tau_5}$$

Rule 16.27:

$$\frac{\{(\mathrm{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathrm{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\quad \mathrm{DE} \vdash \tau_3 \leftarrow \mathbf{in}\mathit{CompDom}(\mathrm{boolean}) \Rightarrow \tau_4,\ \mathrm{DE} \vdash \tau_4 \leftrightarrow \tau \Rightarrow \tau_5}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1\ \mathbf{or}\ \texttt{expression}_2 \Rightarrow \tau_5}$$

Rule 16.28:

$$\frac{\{(\mathrm{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathrm{DE} \vdash \tau_2 \Rightarrow (\mathrm{powerset}, \delta),\quad \mathrm{DE} \vdash \tau_1 \leftrightarrow \mathbf{in}\mathit{DomName}(\delta) \Rightarrow \_,\ \mathrm{DE} \vdash \tau \leftrightarrow \mathbf{in}\mathit{CompDom}(\mathrm{boolean}) \Rightarrow \tau_4}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1\ \mathbf{in}\ \texttt{expression}_2 \Rightarrow \tau_4}$$

Rule 16.29:

$$\frac{\{(\mathrm{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathrm{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\ \mathrm{DE} \vdash \tau_3 \Rightarrow (\mathrm{powerset}, \_),\ \mathrm{DE} \vdash \tau \leftrightarrow \mathbf{in}\mathit{CompDom}(\mathrm{boolean}) \Rightarrow \tau_4}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1 \texttt{ <= } \texttt{expression}_2 \Rightarrow \tau_4}$$

Rule 16.30:

$$\frac{\{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathrm{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\ \mathrm{DE} \vdash \tau_3 \Rightarrow \sigma,\ \mathit{ConName}\ \mathbf{of}\ \sigma \neq \mathrm{function}^k}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1 \texttt{ + } \texttt{expression}_2 \Rightarrow \tau_3}$$

The join over function expressions is defined in Rule 16.5.

Coercion rule:

$$\frac{}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1 \texttt{ + } \texttt{expression}_2 \Rightarrow \texttt{expression}_1 \texttt{ (+ } \tau \texttt{) } \texttt{expression}_2}$$

Rule 16.31:

$$\frac{\{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathrm{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\ \mathrm{DE} \vdash \tau_3 \Rightarrow \sigma,\ \mathit{ConName}\ \mathbf{of}\ \sigma = \mathrm{powerset}}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1 \texttt{ * } \texttt{expression}_2 \Rightarrow \tau_3}$$

Rule 16.32:

$$\frac{\{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathrm{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\ \mathrm{DE} \vdash \tau_3 \Rightarrow \sigma,\ \mathit{ConName}\ \mathbf{of}\ \sigma = \mathrm{powerset}}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1 \texttt{ - } \texttt{expression}_2 \Rightarrow \tau_3}$$

Rule 16.33:

$$\frac{(\mathrm{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression}_1 \Rightarrow \tau_1,\ (\mathrm{DE}, \tau) \vdash \texttt{expression}_2 \Rightarrow \tau_2,\ \mathrm{DE} \vdash \tau_2 \Rightarrow (\mathrm{list}, \delta),\ \mathrm{DE} \vdash \tau_1 \leftrightarrow \mathbf{in}\mathit{DomName}(\delta) \Rightarrow \tau_3}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1 \texttt{ :: } \texttt{expression}_2 \Rightarrow \tau_2}$$

Rule 16.34:

$$\frac{\{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathrm{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\ \mathrm{DE} \vdash \tau_3 \Rightarrow \sigma,\ \mathit{ConName}\ \mathbf{of}\ \sigma = \mathrm{list}}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1 \texttt{ \& } \texttt{expression}_2 \Rightarrow \tau_3}$$

Rule 16.35:

$$\frac{\{(\mathsf{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathsf{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\ \mathsf{DE} \vdash \tau_3 \Rightarrow \sigma,\ \mathit{ConName}\ \mathbf{of}\ \sigma \neq \text{function},\ \mathsf{DE} \vdash \tau \leftrightarrow \mathbf{in}\mathit{CompDom}(\text{boolean}) \Rightarrow \tau_4}{(\mathsf{DE}, \tau, \cdots) \vdash \texttt{expression}_1 = \texttt{expression}_2 \Rightarrow \tau_4}$$

The equality operator is not defined for function and syntactic domains.

Coercion rule:

$$\frac{}{(\cdots) \vdash \texttt{expression}_1 = \texttt{expression}_2 \Rightarrow \texttt{expression}_1\ (=\tau_3)\ \texttt{expression}_2}$$

BINARY PREFIX OPERATIONS:

Rule 16.36:

$$\frac{\{(\mathsf{DE}, \tau, \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathsf{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\ \mathsf{DE} \vdash \tau_3 \leftarrow \mathbf{in}\mathit{CompDom}(\text{integer}) \Rightarrow \tau_4}{(\mathsf{DE}, \tau, \cdots) \vdash \mathbf{add}\ \texttt{expression}_1\ \texttt{expression}_2 \Rightarrow \tau_4}$$

Rule 16.37:

$$\frac{\{(\mathsf{DE}, \tau, \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathsf{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\ \mathsf{DE} \vdash \tau_3 \leftarrow \mathbf{in}\mathit{CompDom}(\text{integer}) \Rightarrow \tau_4}{(\mathsf{DE}, \tau, \cdots) \vdash \mathbf{sub}\ \texttt{expression}_1\ \texttt{expression}_2 \Rightarrow \tau_4}$$

Rule 16.38:

$$\frac{\{(\mathsf{DE}, \tau, \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathsf{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\ \mathsf{DE} \vdash \tau_3 \leftarrow \mathbf{in}\mathit{CompDom}(\text{integer}) \Rightarrow \tau_4}{(\mathsf{DE}, \tau, \cdots) \vdash \mathbf{mult}\ \texttt{expression}_1\ \texttt{expression}_2 \Rightarrow \tau_4}$$

Rule 16.39:

$$\frac{\{(\mathsf{DE}, \tau, \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2,\ \mathsf{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3,\ \mathsf{DE} \vdash \tau_3 \leftarrow \mathbf{in}\mathit{CompDom}(\text{integer}) \Rightarrow \tau_4}{(\mathsf{DE}, \tau, \cdots) \vdash \mathbf{div}\ \texttt{expression}_1\ \texttt{expression}_2 \Rightarrow \tau_4}$$

UNARY POSTFIX OPERATIONS:

Rule 16.40:

$$\frac{(\mathsf{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{variable} \Rightarrow \tau_1,\ \mathsf{DE} \vdash \tau_1 \leftrightarrow \mathbf{in}\mathit{CompDom}(\text{syntactic}) \Rightarrow \_,\ \mathsf{DE} \vdash \tau \leftrightarrow \mathbf{in}\mathit{CompDom}(\text{label}) \Rightarrow \tau_3}{(\mathsf{DE}, \tau, \cdots) \vdash \texttt{variable'}\mathbf{label} \Rightarrow \tau_3}$$

Rule 16.41:

$$\frac{\$ \in \textbf{Dom}\ \mathrm{VE},\ \mathrm{VE}(\$) = \tau_1,\ \tau_1 \in \textbf{Dom}\ \mathrm{CE},\ \mathrm{CE}(\tau_1) = \tau_2,\ \mathrm{DE} \vdash \tau \leftrightarrow \tau_2 \Rightarrow \tau_3}{(\mathrm{DE}, \mathrm{VE}, \mathrm{CE}, \tau, \cdots) \vdash \$\text{'}\textbf{cache} \Rightarrow \tau_3}$$

Rule 16.42:

$$\frac{\begin{array}{c}\mathrm{identifier} \in \textbf{Dom}\ \mathrm{VE},\ \mathrm{VE}(\mathrm{identifier}) = \tau_1,\ \tau_1 \in \textbf{Dom}\ \mathrm{CE},\\ \mathrm{CE}(\tau_1) = \tau_2,\ \mathrm{DE} \vdash \tau \leftrightarrow \tau_2 \Rightarrow \tau_3\end{array}}{(\mathrm{DE}, \mathrm{VE}, \mathrm{CE}, \tau, \cdots) \vdash \mathtt{identifier}\text{'}\textbf{cache} \Rightarrow \tau_3}$$

Rule 16.43:

$$\frac{\begin{array}{c}\mathrm{syntactic\text{–}domain\text{–}name} \in \mathrm{D},\ \mathrm{syntactic\text{–}domain\text{–}name} \in \textbf{Dom}\ \mathrm{CE},\\ \mathrm{CE}(\mathrm{syntactic\text{–}domain\text{–}name}) = \tau_2,\ \mathrm{DE} \vdash \tau \leftrightarrow \tau_2 \Rightarrow \tau_3\end{array}}{(\mathrm{DE}, \mathrm{CE}, \mathrm{D}, \tau, \cdots) \vdash [\![\mathtt{syntactic\text{–}domain\text{–}name}]\!]\text{'}\textbf{cache} \Rightarrow \tau_3}$$

MIXFIX OPERATIONS:

Rule 16.44:

$$\frac{\begin{array}{c}(\mathrm{DE}, \textbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \mathtt{expression}_1 \Rightarrow \tau_1,\ \mathrm{DE} \vdash \tau_1 \Rightarrow (\mathrm{store}, (\delta_1, \delta_2)),\\ (\mathrm{DE}, \textbf{in}\mathit{DomName}(\delta_1)) \vdash \mathtt{expression}_2 \Rightarrow \_,\ \mathrm{DE} \vdash \tau \leftrightarrow \textbf{in}\mathit{DomName}(\delta_2) \Rightarrow \tau_3\end{array}}{(\mathrm{DE}, \tau, \cdots) \vdash \mathtt{expression}_1\,[\,\mathtt{expression}_2\,] \Rightarrow \tau_3}$$

Rule 16.45:

$$\frac{\begin{array}{c}(\mathrm{DE}, \tau, \cdots) \vdash \mathtt{expression}_1 \Rightarrow \tau_1,\ \mathrm{DE} \vdash \tau_1 \Rightarrow (\mathrm{store}, (\delta_1, \_)),\\ (\mathrm{DE}, \textbf{in}\mathit{DomName}(\delta_1), \cdots) \vdash \mathtt{expression}_3 \Rightarrow \_,\\ (\mathrm{DE}, \textbf{in}\mathit{DomName}(\delta_2), \cdots) \vdash \mathtt{expression}_2 \Rightarrow \tau_2\end{array}}{(\mathrm{DE}, \tau, \cdots) \vdash \mathtt{expression}_1\,[\,\mathtt{expression}_2/\mathtt{expression}_3\,] \Rightarrow \tau_1}$$

Rule 16.46:

$$\frac{\begin{array}{c}(\mathrm{DE}, \tau, \cdots) \vdash \mathtt{expression}_1 \Rightarrow \tau_1,\ \mathrm{DE} \vdash \tau_1 \Rightarrow (\mathrm{store}, (\delta_1, \delta_2)),\\ (\mathrm{DE}, \textbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \mathtt{expression}_2 \Rightarrow \tau_2,\ \mathrm{DE} \vdash \tau_2 \Rightarrow (\mathrm{powerset}, \delta_3),\\ \mathrm{VE}' = \{\mathrm{identifier} \mapsto \textbf{in}\mathit{DomName}(\delta_3)\}/\mathrm{VE},\\ (\mathrm{DE}, \mathrm{VE}'\ \textbf{in}\mathit{DomName}(\delta_1), \cdots) \vdash \mathtt{expression}_4 \Rightarrow \_,\\ (\mathrm{DE}, \mathrm{VE}', \textbf{in}\mathit{DomName}(\delta_2), \cdots) \vdash \mathtt{expression}_3 \Rightarrow \_\end{array}}{\begin{array}{l}(\mathrm{DE}, \mathrm{VE}, \tau, \cdots) \vdash \mathtt{expression}_1\,[\,\textbf{foreach}\ \mathtt{identifier}\ \textbf{in}\ \mathtt{expression}_2\ \textbf{do}\\ \qquad\qquad \mathtt{expression}_3/\mathtt{expression}_4\,] \Rightarrow \tau_1\end{array}}$$

3.8.5. Lambda expression

Rule 16.47:

$$\frac{\begin{array}{c}DE \vdash \tau \Rightarrow (\text{function}^{(k)}, (\delta_1, \cdots, \delta_k)),\ k \geq 2,\ VE' = \{\text{identifier} \mapsto \textbf{in}\mathit{DomName}(\delta_1)\}/VE,\\ FI \vdash \texttt{identifier} \Rightarrow FI',\ \tau' = \textbf{in}\mathit{CompDom}(\text{function}^{(k-1)}, (\delta_2, \cdots, \delta_k))\\ (DE, VE', FI', (\psi^k\downarrow 2, \cdots, \psi^k\downarrow k), (\chi^k\downarrow 2, \cdots, \chi^k\downarrow k), \tau', \cdots) \vdash \texttt{expression} \Rightarrow \tau\end{array}}{(DE, VE, FI, \psi^k, \chi^k, \tau, \cdots) \vdash \textbf{lambda}\ \texttt{identifier . expression} \Rightarrow \tau}$$

A lambda expression must occur in only those places where the type can be inferred from context. The expected type must be a function. FI is used to gather lambda variable names for possible use in a fixed-point termination test inside the expression.

Coercion rule:

$$\frac{\psi^k\downarrow 1 = \Sigma,\ [\chi^k\downarrow 1 = \Xi],\ \bar{\tau} = \textbf{in}\mathit{DomName}(\delta_1)}{\begin{array}{l}(\cdots) \vdash \textbf{lambda}\ \texttt{identifier . expression} \Rightarrow\\ \textbf{lambda strict}\ \bar{\tau}\ \tilde{\tau}\ [\textbf{collect}\ C\ CE(VE(\$))]\ \text{identifier . expression}\end{array}}$$

(J20). $FI \vdash \texttt{identifier} \Rightarrow FI'$

Rule 20.1:

$$\frac{(FI = \{\}) \vee ((FI = (\_, (TT, \_))) \wedge \neg\, \textbf{is}\mathit{ArgTest}(T))}{FI \vdash \texttt{identifier} \Rightarrow FI}$$

The lambda expression is not part of a fixed point expression or the termination is not on argument values. There is no need to collect the lambda variable.

Rule 20.2:

$$\frac{FI = (\phi, (TT, \beta)),\ \textbf{is}\mathit{ArgTest}(T)}{FI \vdash \texttt{identifier} \Rightarrow (\phi, (\textbf{in}\mathit{ArgTest}(TT \cup \{\text{identifier}\}), \beta)}$$

3.8.6. Function application

Rule 16.3:

$$\frac{\begin{array}{c}(DE, \textbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression}_1 \Rightarrow \tau_1,\\ DE \vdash \tau_1 \Rightarrow (\text{function}^{(k)}, (\delta_1, \cdots, \delta_k)),\ k \geq 2,\\ (DE, \textbf{in}\mathit{DomName}(\delta_1), \cdots) \vdash \texttt{expression}_2 \Rightarrow \_,\\ DE \vdash \tau \leftrightarrow \textbf{in}\mathit{CompDom}(\text{function}^{(k-1)}, (\delta_2, \cdots, \delta_k)) \Rightarrow \tau_3\end{array}}{(DE, \tau, \cdots) \vdash \texttt{expression}_1\ \texttt{expression}_2 \Rightarrow \tau_3}$$

Function is curried once.

3.8.7. Expression composition

Rule 16.4:

$$\frac{\begin{array}{c}\{(\mathrm{DE}, \mathbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2, \\ \mathrm{DE} \vdash \tau_1 \Rightarrow (\mathrm{function}^{(k)}, (\delta_1, \cdots, \delta_k)), \mathrm{DE} \vdash \tau_2 \Rightarrow (\mathrm{function}^{(n)}, (\delta'_1, \cdots, \delta'_n)), \\ k \geq 2, n \geq 2, \mathrm{DE} \vdash \tau \leftrightarrow \mathbf{in}\mathit{CompDom}(\mathrm{function}^{(k-1)}, (\delta_2, \cdots, \delta_k)) \Rightarrow \tau_4, \\ \mathrm{DE} \vdash \mathbf{in}\mathit{DomName}(\delta_1) \leftrightarrow \mathbf{in}\mathit{DomStruct}((\mathrm{function}^{(n-1)}, (\delta'_2, \cdots, \delta'_n)) \Rightarrow \_\end{array}}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1\texttt{ . expression}_2 \Rightarrow \tau_4}$$

Both expressions must evaluate to functions. The type of the value returned by a single application of the second expression must be equivalent to the type of the first argument of the first function.

3.8.8. Expression union

Rule 16.5:

$$\frac{\{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_1^2, \mathrm{DE} \vdash \tau_1 \leftrightarrow \tau_2 \Rightarrow \tau_3, \quad \mathrm{DE} \vdash \tau_3 \Rightarrow (\mathrm{function}^{(k)}, \_)}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1\texttt{ + expression}_2 \Rightarrow \tau_3}$$

Coercion rule:

$$\frac{\mathrm{DE} \vdash \tau \Rightarrow (\mathrm{function}^{(k)}, (\delta_1, \cdots, \delta_k)), \quad \tau' = \mathbf{in}\mathit{CompDom}(\mathrm{function}^{(k-1)}, (\delta_2, \cdots, \delta_k))}{\begin{array}{r}(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1\texttt{ + expression}_2 \Rightarrow \\ \texttt{expression}_1\texttt{ (} \oplus \tau' \texttt{) expression}_2\end{array}}$$

3.8.9. Conditional expression

Rule 16.6:

$$\frac{\begin{array}{c}(\mathrm{DE}, \mathbf{in}\mathit{CompDom}(\mathrm{boolean}), \cdots) \vdash \texttt{expression}_1 \Rightarrow \_, \\ \{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_i \Rightarrow \tau_i\}_2^3, \mathrm{DE} \vdash \tau_2 \leftrightarrow \tau_3 \Rightarrow \tau_4\end{array}}{(\mathrm{DE}, \tau, \cdots) \vdash \texttt{expression}_1\texttt{ --> expression}_2\texttt{ , expression}_3 \Rightarrow \tau_4}$$

3.8.10. Fixed point expression

Rule 16.7:

$$\frac{\begin{array}{c}\tau \neq \Gamma, \vdash \texttt{termination-option function-name} \Rightarrow \mathrm{FI}, \\ (\mathrm{DE}, \{\texttt{function-name} \mapsto \tau\}/\mathrm{VE}, \mathrm{FI}, \tau, \cdots) \vdash \texttt{expression} \Rightarrow \tau_1\end{array}}{\begin{array}{r}(\mathrm{DE}, \mathrm{VE}, \tau, \cdots) \vdash \mathbf{fix}\texttt{ termination-option function-name .} \\ \texttt{expression} \Rightarrow \tau_1\end{array}}$$

We will assume that the expression corresponding to the body of the fixed point expression is in a normal form where all the λs corresponding to the functionality of the expression must be grouped at the beginning of the expression. We will not provide rules to transform expression to this form. The editor/compiler must either enforce this convention or perform required transformations.

Coercion rule:

$$\frac{}{(\cdots) \vdash \textbf{fix}\ \texttt{termination-option function-name . expression} \Rightarrow \textbf{fix}\ \texttt{function-name . expression}}$$

A fixed point termination test will be inserted in the expression after all the lambda bindings (See coercion rule at the beginning of Section 3.8). The termination information is captured in FI. Hence the termination option is no longer required in the coerced expression. For example, the expression

fix argument f . λ x. λ y.expr

will be coerced to

fix f . λ x. λ y. **fixtest** ··· expr

(J21). ⊢ `termination-option function-name` ⇒ FI

Rule 21.1:

$$\frac{}{\vdash \texttt{function-name} \Rightarrow \{\}}$$

No termination criterion specified.

Rule 21.2:

$$\frac{\vdash \texttt{bound-option} \Rightarrow \beta}{\vdash \texttt{bound-option function-name} \Rightarrow (\text{function-name}, (\{\}, \beta))}$$

Only bounded termination specified.

Rule 21.3:

$$\frac{\vdash \texttt{test-option} \Rightarrow TT}{\vdash \texttt{test-option function-name} \Rightarrow (\text{function-name}, (TT, -1))}$$

Only argument or cache fixed point specified.

Rule 21.4:

$$\frac{\vdash \texttt{test-option} \Rightarrow TT,\ \vdash \texttt{bound-option} \Rightarrow \beta}{\vdash \texttt{test-option bound-option function-name} \Rightarrow (\text{function-name}, (TT, \beta))}$$

Both bounded termination as well as argument or cache fixed point specified.

(J22). $\vdash$ `bound-option` $\Rightarrow \beta$

Rule 22.1:

$$\frac{}{\vdash \textbf{bounded} \text{ (} \texttt{integer-literal} \text{)} \Rightarrow \text{integer-literal}}$$

(J23). $\vdash$ `test-option` $\Rightarrow$ TT

Rule 23.1:

$$\frac{}{\vdash \textbf{argument} \Rightarrow \textbf{is}\mathit{ArgTest}(\{\})}$$

Rule 23.2:

$$\frac{}{\vdash \textbf{cache} \Rightarrow \textbf{is}\mathit{CacheTest}(\{\Phi\})}$$

3.8.11. Let expression

Rule 16.3:

$$\frac{\begin{array}{c}(\text{DE, FE, VE, CE, D}) \vdash \{\texttt{let-clause}_i\}_1^k \Rightarrow \text{VE}', \\ (\text{DE, FE, VE}', \text{CE, D, } \cdots) \vdash \texttt{expression} \Rightarrow \tau_1\end{array}}{(\text{DE, FE, VE, CE, D, } \cdots) \vdash \textbf{let} \{\texttt{let-clause}_i\}_1^k \textbf{ in } \texttt{expression} \Rightarrow \tau_1}$$

The expression is processed in the environment VE′ created by the processing of the let clauses.

Rule 16.4:

$$\frac{\text{VE}_0 = \text{VE}, \{(\text{DE, FE, VE}_{i-1}, \text{CE, D}) \vdash \texttt{let-clause}_i \Rightarrow \text{VE}_i\}_1^k}{(\text{DE, FE, VE, CE, D}) \vdash \{\texttt{let-clause}_i\}_1^k \Rightarrow \text{VE}_k}$$

The let clauses are processed sequentially. Each let clause is processed in an environment created by any previous let clauses.

(J24). (DE, FE, VE, CE, D) $\vdash$ `let-clause` $\Rightarrow$ VE′

Rule 24.1:

$$\frac{(\text{DE, FE, VE, CE, D, } \textbf{in}\mathit{AnyType}(\Gamma)) \vdash \texttt{expression} \Rightarrow \tau}{(\text{DE, FE, VE, CE, D}) \vdash \texttt{identifier = expression} \Rightarrow \{\text{identifier} \mapsto \tau\}/\text{VE}}$$

The identifier is not defined until the end of the expression and cannot be used recursively in the expression. The identifier masks any previous bindings of the same identifier.

3.8.12. Typecast expression

Rule 16.2:

$$\frac{\text{domain-name} \in \textbf{Dom DE}, \text{DE} \vdash \textbf{in}\mathit{DomName}(\text{domain-name}) \leftrightarrow \tau \Rightarrow \tau_1, \quad (\text{DE}, \tau_1, \cdots) \vdash \texttt{expression} \Rightarrow \tau_2}{(\text{DE}, \tau, \cdots) \vdash \texttt{domain-name expression} \Rightarrow \tau_2}$$

The type of the expression is either the same as the specified domain name (redundant typecasting) or the type casting can be done statically. Hence the expression is coerced as follows:

Coercion rule:

$$\frac{}{(\cdots) \vdash \texttt{domain-name expression} \Rightarrow \text{expression}}$$

Rule 16.3:

$$\frac{\text{domain-name} \in \textbf{Dom DE}, \text{DE} \vdash \textbf{in}\mathit{DomName}(\text{domain-name}) \leftrightarrow \tau \Rightarrow \tau_1, \quad (\text{DE}, \textbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression} \Rightarrow \tau_2, \text{DE} \vdash \tau_1 \leftarrow \tau_2 \Rightarrow \tau_3}{(\text{DE}, \tau, \cdots) \vdash \texttt{domain-name expression} \Rightarrow \tau_3}$$

The expression is coerced to a lifted or topped domain. Again this is a static typecasting and the following type coercion is used:

Coercion rule:

$$\frac{}{(\cdots) \vdash \texttt{domain-name expression} \Rightarrow \text{expression}}$$

Rule 16.4:

$$\frac{\text{domain-name} \in \textbf{Dom DE}, \text{DE} \vdash \textbf{in}\mathit{DomName}(\text{domain-name}) \leftrightarrow \tau \Rightarrow \tau_1, \quad \text{DE} \vdash \tau_1 \Rightarrow (\text{union}^{(k)}, \delta^k), \quad (\text{DE}, \textbf{in}\mathit{AnyType}(\Gamma), \cdots) \vdash \texttt{expression} \Rightarrow \bar{\delta}, \exists i{:}1..k.\ \delta^k{\downarrow}i = \bar{\delta}}{(\text{DE}, \tau, \cdots) \vdash \texttt{domain-name expression} \Rightarrow \tau_1}$$

The expression is injected into a union domain. As the values from a union domain will be represented by tagged elements in the dynamic semantics, the following coercion rule is used:

Coercion rule:

$$\frac{}{(\cdots) \vdash \texttt{domain-name expression} \Rightarrow \bar{\delta}\ \text{expression}}$$

3.8.13. Parenthesized expression

Rule 16.5:

$$\frac{(\mathsf{DE}, \tau, \cdots) \vdash \texttt{expression} \Rightarrow \tau_1}{(\mathsf{DE}, \tau, \cdots) \vdash \texttt{(expression)} \Rightarrow \tau_1}$$

4. Dynamic Semantics

A "program" in SPARE is assumed to have the following syntax:

```
program  ::=  specification;
              Interpret(nonterminal,instance,
                    function-name)
```

`nonterminal` and `function-name` are identifiers while `instance` is an integer. The collection of rules that constitute the dynamic semantics allows the derivation of a *judgment* of the form

$$\mathsf{AST} \vdash \texttt{program} \Rightarrow \mathsf{v}$$

as a theorem for all "valid programs". AST is a semantic object that denotes an abstract syntax tree for a program in the language for which the specifications are written. v is a semantic object that denotes the information obtained by performing the analysis on the program represented by the abstract syntax tree. These semantic objects are defined in the next section.

4.1. Semantic Domains

Several domains defined in Section 3.1 are used here and will not be redefined:

		Int		Integers
		Bool		Booleans
ι	∈	*Instance*		Integers
		NodeInst	=	$NonVar \times Instance$
FX	∈	*FixInst*	=	$(NodeInst \cup \{\}) \times FuncName$
IM	∈	*InstMap*	=	$NonVar \Rightarrow_{fin} Instance$
AST	∈	*ASTree*	=	$NodeInst \Rightarrow_{fin} ProdVar \times InstMap$
v	∈	*Value*	=	$Int \cup Bool \cup Str \cup \{\perp, \top\} \times Type \cup$ $SetValue \cup TupleValue \cup UnionValue \cup$ $StoreValue \cup NodeInst$ $FuncComp \cup FuncUnion \cup$ $LambClosure \cup StrLambClosure$
		SetValue	=	$\mathbf{P}_{fin}(Value)$
		TupleValue	=	$\bigcup_{k \geq 1} Value^k \cup \mathbf{nil}$

		UnionValue	=	*DomName* × *Value*
		StoreValue	=	*Value* $\Rightarrow_{fin}$ *Value*
		FuncComp	=	*Value* × *Value*
		FuncUnion	=	*Value* × *Value* × *Type*
		LambClosure	=	*TypeCorExpr* × *VarName* × *VarStore* × *InstMap* × *CollectInfo*
		StrLambClosure	=	*TypeCorExpr* × *VarName* × *VarStore* × *InstMap* × *Type* × *Type* × *CollectInfo*
CI	∈	*CollectInfo*	=	(*ColDescr* × *Type*) ∪ {}
VS	∈	*VarStore*	=	(*VarName* ∪ *FuncName*) $\Rightarrow_{fin}$ *Value*
AS	∈	*ArgStore*	=	*VarName* $\Rightarrow_{fin}$ *Value*
CS	∈	*CacheStore*	=	*NodeInst* $\Rightarrow_{fin}$ *Value*
FS	∈	*FixStore*	=	*FixInst* $\Rightarrow_{fin}$ (*ArgStore* ∪ *NodeInst*) × *Bound*

nil denotes the tuple with zero elements.

4.2. Standard operators

Most of the primitive operations in the SPARE language correspond to the standard operators commonly used in domain theory [8]. The following standard operators are assumed to have the usual definitions:

Integer operators:

– (unary and binary), +, ×, /

Boolean operators:

∧, ∨, ¬

Set operators:

∈, ⊆, ∪, ∩, --

List operators:

hd, tl, null, cons, append

Product operator:

↓

Lifted, topped and union domain inherit operations from their base/component domains. The following rules define the standard operators mentioned above on such domains. The inherited operations are also strict in lifted domains. The rules to ensure strictness are trivial and will not be provided here.

unary prefix operators

Rule 24.6:

$$\frac{v = (\delta, v'),\ \vdash -v' \Rightarrow v_1}{\vdash -v \Rightarrow (\delta, v_1)}$$

Rule 24.7:

$$\frac{\mathbf{is}\mathit{Int}(v)}{\vdash -v \Rightarrow \mathbf{in}\mathit{Int}(-v)}$$

Rule 24.8:

$$\frac{v = (\delta, v'),\ \vdash \neg v' \Rightarrow v_1}{\vdash \neg v \Rightarrow v_1}$$

Rule 24.9:

$$\frac{\mathbf{is}\mathit{Bool}(v)}{\vdash \neg v \Rightarrow \mathbf{in}\mathit{Bool}(\neg v)}$$

Rule 24.10:

$$\frac{v = (\delta, v'),\ \vdash \mathrm{hd}\ v' \Rightarrow v_1}{\vdash \mathrm{hd}\ v \Rightarrow v_1}$$

Rule 24.11:

$$\frac{\mathbf{is}\mathit{TupleValue}(v),\ v \neq \mathbf{nil}}{\vdash \mathrm{hd}\ v \Rightarrow \mathbf{in}\mathit{TupleValue}(\mathrm{hd}\ v)}$$

Rule 24.12:

$$\frac{v = (\delta, v'),\ \vdash \mathrm{tl}\ v' \Rightarrow v_1}{\vdash \mathrm{tl}\ v \Rightarrow (\delta, v_1)}$$

Rule 24.13:

$$\frac{\mathbf{is}\mathit{TupleValue}(v)}{\vdash \mathrm{tl}\ v \Rightarrow \mathbf{in}\mathit{TupleValue}(\mathrm{tl}\ v)}$$

Rule 24.14:

$$\frac{v = (\delta, v'),\ \vdash \mathrm{null}\ v' \Rightarrow v_1}{\vdash \mathrm{null}\ v \Rightarrow v_1}$$

Rule 24.15:

$$\frac{\mathbf{is}\mathit{TupleValue}(\nu)}{\vdash \mathsf{null}\ \nu \Rightarrow \mathbf{in}\mathit{Bool}(\mathsf{null}\ \nu)}$$

binary infix operators

Rule 24.16:

$$\frac{\nu = (\delta, \nu'),\ \vdash \nu'\downarrow i \Rightarrow \nu_1}{\vdash \nu\downarrow i \Rightarrow \nu_1}$$

Rule 24.17:

$$\frac{\mathbf{is}\mathit{TupleValue}(\nu)}{\vdash \nu\downarrow i \Rightarrow \mathbf{in}\mathit{TupleValue}(\nu\downarrow i)}$$

Rule 24.18:

$$\frac{\nu = (\delta_1, \nu'),\ \vdash \nu'\ \mathbf{is}\ \delta' \Rightarrow \nu_1}{\vdash \nu\ \mathbf{is}\ \delta \Rightarrow \nu_1}$$

Rule 24.19:

$$\frac{\nu = (\delta_1, \nu')}{\vdash \nu\ \mathbf{is}\ \delta \Rightarrow \mathbf{in}\mathit{Bool}(\delta_1 = \delta)}$$

Rule 24.20:

$$\frac{\{\nu_i = (\delta, \nu_i')\}_1^2,\ \vdash \nu_1' \wedge \nu_2' \Rightarrow \nu_3}{\vdash \nu_1 \wedge \nu_2 \Rightarrow \nu_3}$$

Rule 24.21:

$$\frac{\{\mathbf{is}\mathit{Bool}(\nu_i)\}_1^2}{\vdash \nu_1 \wedge \nu_2 \Rightarrow \mathbf{in}\mathit{Bool}(\nu_1 \wedge \nu_2)}$$

Rule 24.22:

$$\frac{\{\nu_i = (\delta, \nu_i')\}_1^2,\ \vdash \nu_1' \vee \nu_2' \Rightarrow \nu_3}{\vdash \nu_1 \vee \nu_2 \Rightarrow \nu_3}$$

Rule 24.23:

$$\frac{\{\mathbf{is}\mathit{Bool}(\nu_i)\}_1^2}{\vdash \nu_1 \vee \nu_2 \Rightarrow \mathbf{in}\mathit{Bool}(\nu_1 \vee \nu_2)}$$

Rule 24.24:

$$\frac{v_2 = (\delta, v_2'),\ \vdash v_1 \in v_2' \Rightarrow v_3}{\vdash v_1 \in v_2 \Rightarrow v_3}$$

Rule 24.25:

$$\frac{\mathbf{is}\mathit{SetValue}(v_2)}{\vdash v_1 \in v_2 \Rightarrow \mathbf{in}\mathit{Bool}(v_1 \in v_2)}$$

Rule 24.26:

$$\frac{\{v_i = (\delta, v_i')\}_1^2,\ \vdash v_1' \subseteq v_2' \Rightarrow v_3}{\vdash v_1 \subseteq v_2 \Rightarrow v_3}$$

Rule 24.27:

$$\frac{\{\mathbf{is}\mathit{SetValue}(v_i)\}_1^2}{\vdash v_1 \subseteq v_2 \Rightarrow \mathbf{in}\mathit{Bool}(v_1 \subseteq v_2)}$$

Rule 24.28:

$$\frac{\{v_i = (\delta, v_i')\}_1^2,\ \vdash v_1' \cup v_2' \Rightarrow v_3}{\vdash v_1 \cup v_2 \Rightarrow (\delta, v_3)}$$

Rule 24.29:

$$\frac{\{\mathbf{is}\mathit{SetValue}(v_i)\}_1^2}{\vdash v_1 \cup v_2 \Rightarrow \mathbf{in}\mathit{SetValue}(v_1 \cup v_2)}$$

Rule 24.30:

$$\frac{\{v_i = (\delta, v_i')\}_1^2,\ \vdash v_1' \cap v_2' \Rightarrow v_3}{\vdash v_1 \cap v_2 \Rightarrow (\delta, v_3)}$$

Rule 24.31:

$$\frac{\{\mathbf{is}\mathit{SetValue}(v_i)\}_1^2}{\vdash v_1 \cap v_2 \Rightarrow \mathbf{in}\mathit{SetValue}(v_1 \cap v_2)}$$

Rule 24.32:

$$\frac{\{v_i = (\delta, v_i')\}_1^2,\ \vdash v_1' - v_2' \Rightarrow v_3}{\vdash v_1 - v_2 \Rightarrow (\delta, v_3)}$$

Rule 24.33:

$$\frac{\{\mathbf{is}\mathit{SetValue}(v_i)\}_1^2}{\vdash v_1 - v_2 \Rightarrow \mathbf{in}\mathit{SetValue}(v_1 - v_2)}$$

Rule 24.34:

$$\frac{\{\mathbf{is}\mathit{Int}(v_i)\}_1^2}{\vdash v_1 - v_2 \Rightarrow \mathbf{in}\mathit{Int}(v_1 - v_2)}$$

Rule 24.35:

$$\frac{v_2 = (\delta, v_2'),\ \vdash \mathsf{Cons}(v_1, v_2') \Rightarrow v_3}{\vdash \mathsf{Cons}(v_1, v_2) \Rightarrow (\delta, v_3)}$$

Rule 24.36:

$$\frac{\mathbf{is}\mathit{TupleValue}(v_2)}{\vdash \mathsf{Cons}(v_1, v_2) \Rightarrow \mathbf{in}\mathit{TupleValue}(\mathsf{Cons}(v_1, v_2))}$$

Rule 24.37:

$$\frac{\{v_i = (\delta, v_i')\}_1^2,\ \vdash \mathsf{Append}(v_1', v_2') \Rightarrow v_3}{\vdash \mathsf{Append}(v_1, v_2) \Rightarrow (\delta, v_3)}$$

Rule 24.38:

$$\frac{\{\mathbf{is}\mathit{TupleValue}(v_i)\}_1^2}{\vdash \mathsf{Append}(v_1, v_2) \Rightarrow \mathbf{in}\mathit{TupleValue}(\mathsf{Append}(v_1, v_2))}$$

Rule 24.39:

$$\frac{\{v_i = (\delta, v_i')\}_1^2,\ \vdash v_1' + v_2' \Rightarrow v_3}{\vdash v_1 + v_2 \Rightarrow (\delta, v_3)}$$

Rule 24.40:

$$\frac{\{\mathbf{is}\mathit{Int}(v_i)\}_1^2}{\vdash v_1 + v_2 \Rightarrow \mathbf{in}\mathit{Int}(v_1 + v_2)}$$

Rule 24.41:

$$\frac{\{v_i = (\delta, v_i')\}_1^2,\ \vdash v_1' \times v_2' \Rightarrow v_3}{\vdash v_1 \times v_2 \Rightarrow (\delta, v_3)}$$

Rule 24.42:

$$\frac{\{\mathbf{is}\mathit{Int}(v_i)\}_1^2}{\vdash v_1 \times v_2 \Rightarrow \mathbf{in}\mathit{Int}(v_1 \times v_2)}$$

Rule 24.43:

$$\frac{\{v_i = (\delta, v_i')\}_1^2,\ \vdash v_1' / v_2' \Rightarrow v_3}{\vdash v_1 / v_2 \Rightarrow (\delta, v_3)}$$

Rule 24.44:

$$\frac{\{\mathbf{is}\mathit{Int}(v_i)\}_1^2}{\vdash v_1 / v_2 \Rightarrow \mathbf{in}\mathit{Int}(v_1 / v_2)}$$

4.3. Overloaded operators

The standard operators join (∇), equal (=), bottom (Bottom) and top (Top) are defined with respect to the partial ordering specified on the domains. The ordering is either implicitly induced by domain constructors or explicitly defined in the specifications. The rules below constitute the definition of these operators for domains in which the ordering is not specified through the external option [9]. When the external option is used, external functions must be provided for these operations and the compiler/editor must ensure that these functions are used at the appropriate places. We will not formalize the use of the external option.

EQUAL OPERATOR

(J25). $DE \vdash v_1 =_\tau v_2$

Rule 25.1:

$$\frac{v_1 = v_2}{DE \vdash v_1 =_\tau v_2}$$

Identical denotations denote the same value.

Rule 25.2:

$$\frac{DE \vdash \mathrm{Bottom}_\tau \Rightarrow \tilde{v},\ DE \vdash \tilde{v} =_\tau v_2,\ DE \vdash \tilde{v} =_\tau v_1}{DE \vdash v_1 =_\tau v_2}$$

Both denote $\perp$.

Rule 25.3:

$$\frac{DE \vdash Top_\tau \Rightarrow \tilde{v},\ DE \vdash \tilde{v} =_\tau v_2,\ DE \vdash \tilde{v} =_\tau v_1}{DE \vdash v_1 =_\tau v_2}$$

Both denote $\top$.

Rule 25.4:

$$\frac{DE \vdash \tau \Rightarrow (\text{lifted}, \delta),\ \tilde{\tau} = \textbf{in}\mathit{DomName}(\delta),\ DE \vdash v_1 =_{\tilde{\tau}} v_2}{DE \vdash v_1 =_\tau v_2}$$

Values from a lifted domain are equal if they are equal in the base domain.

Rule 25.5:

$$\frac{DE \vdash \tau \Rightarrow (\text{topped}, \delta),\ \tilde{\tau} = \textbf{in}\mathit{DomName}(\delta),\ DE \vdash v_1 =_{\tilde{\tau}} v_2}{DE \vdash v_1 =_\tau v_2}$$

Values from a topped domain are equal if they are equal in the base domain.

Rule 25.6:

$$\frac{\begin{array}{c} DE \vdash \tau \Rightarrow (\text{product}^{(k)}, \delta^k), \\ \forall i{:}1 \,..\, k.\ \tau i = \textbf{in}\mathit{DomName}(\delta^k{\downarrow}i),\ DE \vdash v_1{\downarrow}i =_{\tau i} v_2{\downarrow}i \end{array}}{DE \vdash v_1 =_\tau v_2}$$

Values from a product domain are equal if each of the corresponding components are equal.

Rule 25.7:

$$\frac{\begin{array}{c} DE \vdash \tau \Rightarrow (\text{list}, \delta),\ \tilde{\tau} = \textbf{in}\mathit{DomName}(\delta), \\ DE \vdash \text{hd}\ v_1 =_{\tilde{\tau}} \text{hd}\ v_2,\ \ DE \vdash \text{tl}\ v_1 =_{\tilde{\tau}} \text{tl}\ v_2 \end{array}}{DE \vdash v_1 =_\tau v_2}$$

Values from a list domain are equal if each of the corresponding elements of the list are equal.

Rule 25.8:

$$\frac{\begin{array}{c} DE \vdash \tau \Rightarrow (\text{store}, (\_, \delta)),\ \textbf{Dom}\ v_1 = \textbf{Dom}\ v_2, \\ \forall \tilde{v} \in \textbf{Dom}\ v_1.DE \vdash v_1(\tilde{v}) =_\delta v_2(\tilde{v}) \end{array}}{DE \vdash v_1 =_\tau v_2}$$

Values from a store domain are equal if they are identical mappings.

Rule 25.9:

$$\frac{\begin{array}{c} DE \vdash \tau \Rightarrow (\text{union}^{(k)}, \_),\ \{\mathit{DomName}\ \textbf{of}\ v_i = \delta'\}_1^2, \\ \tilde{\tau} = \textbf{in}\mathit{DomName}(\delta'),\ DE \vdash \mathit{Value}\ \textbf{of}\ v_1 =_{\tilde{\tau}} \mathit{Value}\ \textbf{of}\ v_2 \end{array}}{DE \vdash v_1 =_\tau v_2}$$

Values from a union domain are equal if they are from the same component domain and equal in that domain.

Rule 25.10:

$$\frac{DE \vdash \tau \Rightarrow (\text{powerset}, \delta),\ \forall \bar{v}_1 \in v_1.\exists \bar{v}_2 \in v_2.DE \vdash \bar{v}_1 =_\delta \bar{v}_2,\ \forall \bar{v}_2 \in v_2.\exists \bar{v}_1 \in v_1.DE \vdash \bar{v}_1 =_\delta \bar{v}_2}{DE \vdash v_1 =_\tau v_2}$$

Values from a powerset domain are equal if they have the same elements.

JOIN OPERATOR

(J26). $DE \vdash v_1 \nabla_\tau v_2 \Rightarrow v_3$

Rule 26.1:

$$\frac{DE \vdash v_1 =_\tau v_2}{DE \vdash v_1 \nabla_\tau v_2 \Rightarrow v_1}$$

Join between same values.

Rule 26.2:

$$\frac{DE \vdash \text{Bottom}_\tau \Rightarrow \bar{v},\ DE \vdash v_1 =_\tau \bar{v}}{DE \vdash v_1 \nabla_\tau v_2 \Rightarrow v_2}$$

Rule 26.3:

$$\frac{DE \vdash \text{Bottom}_\tau \Rightarrow \bar{v},\ DE \vdash v_2 =_\tau \bar{v}}{DE \vdash v_1 \nabla_\tau v_2 \Rightarrow v_1}$$

Join with $\perp$.

Rule 26.4:

$$\frac{DE \vdash \text{Top}_\tau \Rightarrow \bar{v},\ DE \vdash v_1 =_\tau \bar{v}}{DE \vdash v_1 \nabla_\tau v_2 \Rightarrow v_1}$$

Rule 26.5:

$$\frac{DE \vdash \text{Top}_\tau \Rightarrow \bar{v},\ DE \vdash v_2 =_\tau \bar{v}}{DE \vdash v_1 \nabla_\tau v_2 \Rightarrow v_2}$$

Join with $\top$.

Rule 26.6:

$$\frac{DE \vdash \tau \Rightarrow (\text{lifted}, \delta),\ DE \vdash v_1\ \nabla_\delta\ v_2 \Rightarrow v}{DE \vdash v_1\ \nabla_\tau\ v_2 \Rightarrow v}$$

Join in a lifted domain.

Rule 26.7:

$$\frac{DE \vdash \tau \Rightarrow (\text{topped}, \delta),\ DE \vdash v_1\ \nabla_\delta\ v_2 \Rightarrow v}{DE \vdash v_1\ \nabla_\tau\ v_2 \Rightarrow v}$$

Join in a topped domain.

Rule 26.8:

$$\frac{DE \vdash \tau \Rightarrow (\text{product}^{(k)}, \delta^k),\ \tilde{\tau} = \textbf{in}\mathit{DomName}(\delta^k{\downarrow}i),\ \{DE \vdash v_1{\downarrow}i\ \nabla_{\tau'}\ v_2{\downarrow}i \Rightarrow v'_i\}_1^k}{DE \vdash v_1\ \nabla_\tau\ v_2 \Rightarrow \textbf{in}\mathit{TupleValue}(v'_1, \cdots, v'_k)}$$

Join in a product domain.

Rule 26.9:

$$\frac{DE \vdash \tau \Rightarrow (\text{union}^{(k)}, \_),\ \{\mathit{DomName}\ \textbf{of}\ v_i = \delta'\}_1^2,\ DE \vdash \mathit{Value}\ \textbf{of}\ v_1\ \nabla_{\delta'}\ \mathit{Value}\ \textbf{of}\ v_2 \Rightarrow v'}{DE \vdash v_1\ \nabla_\tau\ v_2 \Rightarrow \textbf{in}\mathit{UnionValue}(\delta, v')}$$

Join in a union domain.

Rule 26.10:

$$\frac{DE \vdash \tau \Rightarrow (\text{powerset}, \_)}{DE \vdash v_1\ \nabla_\tau\ v_2 \Rightarrow \textbf{in}\mathit{SetValue}(v_1 \cup v_2)}$$

Join in a powerset domain.

Rule 26.11:

$$\frac{DE \vdash \tau \Rightarrow (\text{list}, \delta),\ v = \text{hd}\ v_1\ \nabla_\delta\ v_2, v' = \text{tl}\ v_1\ \nabla_\tau\ v_2}{DE \vdash v_1\ \nabla_\tau\ v_2 \Rightarrow \textbf{in}\mathit{TupleValue}(\text{Cons}(v, v'))}$$

Join in a list domain.

Rule 26.12:

$$\frac{\begin{array}{r} DE \vdash \tau \Rightarrow (\text{store}, (\_, \delta)),\\ v = \{v'_1 \mapsto v'_2 \mid (v'_1 \in (\textbf{Dom}\ v_1 - \textbf{Dom}\ v_2) \wedge v'_2 = v_1(v'_1)) \vee\\ (v'_1 \in (\textbf{Dom}\ v_2 - \textbf{Dom}\ v_1) \wedge v'_2 = v_2(v'_1)) \vee\\ (v'_1 \in (\textbf{Dom}\ v_1 \cup \textbf{Dom}\ v_2) \wedge DE \vdash v_1(v'_1)\ \nabla_\delta\ v_2(v'_1) \Rightarrow v'_2\} \end{array}}{DE \vdash v_1\ \nabla_\tau\ v_2 \Rightarrow \textbf{in}\mathit{StoreValue}(v)}$$

Join in a store domain. The join of two store domains A and B is a pointwise join of the two maps A/B and B/A.

Rule 26.13:

$$\frac{\mathsf{DE} \vdash \tau \Rightarrow (\zeta, \omega),\ V = \{v \in \zeta \mid (\zeta, \omega) \vdash v_1 \le v \wedge (\zeta, \omega) \vdash v_2 \le v\},\ \exists v' \in V.\ \forall \tilde{v} \in V.\ (\zeta, \omega) \vdash v' \le \tilde{v}}{\mathsf{DE} \vdash v_1\ \nabla_\tau\ v_2 \Rightarrow \mathbf{in}Str(v')}$$

Join in an enumeration domain with ordering specified by ω.

Ordering relation for enumeration domains:

(J27). $(\zeta, \omega) \vdash v_1 \le v_2$

Rule 27.1:

$$\frac{v_1 = v_2}{(\zeta, \omega) \vdash v_1 \le v_2}$$

The ordering is reflexive.

Rule 27.2:

$$\frac{(v_1, v_2) \in \omega}{(\zeta, \omega) \vdash v_1 \le v_2}$$

The ordering is specified explicitly as a tuple.

Rule 27.3:

$$\frac{\exists \tilde{v} \in \zeta.\ (\zeta, \omega) \vdash v_1 \le \tilde{v} \wedge (\zeta, \omega) \vdash \tilde{v} \le v_2}{(\zeta, \omega) \vdash v_1 \le v_2}$$

The ordering is transitive.

BOTTOM OPERATOR:

(J28). $\mathsf{DE} \vdash \mathsf{Bottom}_\tau \Rightarrow v$

Rule 28.1:

$$\frac{\mathsf{DE} \vdash \tau \Rightarrow (\mathsf{lifted}, \_)}{\mathsf{DE} \vdash \mathsf{Bottom}_\tau \Rightarrow (\bot, \tau)}$$

⊥ of a lifted domain.

Rule 28.2:

$$\frac{DE \vdash \tau \Rightarrow (\text{topped}, \delta),\ DE \vdash \text{Bottom}_{\delta} \Rightarrow \nu}{DE \vdash \text{Bottom}_{\tau} \Rightarrow \nu}$$

$\perp$ of a topped domain.

Rule 28.3:

$$\frac{DE \vdash \tau \Rightarrow (\text{product}^{(k)}, \delta^k),\ \{\tau i = \textbf{in}\mathit{DomName}(\delta^k{\downarrow}i),\ DE \vdash \text{Bottom}_{\tau i} \Rightarrow \nu_i\}_1^k}{DE \vdash \text{Bottom}_{\tau} \Rightarrow \textbf{in}\mathit{TupleValue}(\{\nu_i\}_1^k)}$$

$\perp$ of a product domain.

Rule 28.4:

$$\frac{DE \vdash \tau \Rightarrow (\text{powerset}, \_)}{DE \vdash \text{Bottom}_{\tau} \Rightarrow \textbf{in}\mathit{SetValue}(\{\})}$$

$\perp$ of a powerset domain.

Rule 28.5:

$$\frac{DE \vdash \tau \Rightarrow (\text{store}, \_)}{DE \vdash \text{Bottom}_{\tau} \Rightarrow \textbf{in}\mathit{StoreValue}(\{\})}$$

$\perp$ of a store domain.

Rule 28.6:

$$\frac{DE \vdash \tau \Rightarrow (\zeta, \omega),\ \exists \nu \in \zeta.\ \forall \tilde{\nu} \in \zeta.(\zeta, \omega) \vdash \nu \leq \tilde{\nu}}{DE \vdash \text{Bottom}_{\tau} \Rightarrow \textbf{in}\mathit{Str}(\nu)}$$

$\perp$ of an enumeration domain.

TOP OPERATOR:

(J29). $DE \vdash \text{Top}_{\tau} \Rightarrow \nu$

Rule 29.1:

$$\frac{DE \vdash \tau \Rightarrow (\text{lifted}, \delta),\ DE \vdash \text{Top}_{\delta} \Rightarrow \nu}{DE \vdash \text{Top}_{\tau} \Rightarrow \nu}$$

$\top$ of a lifted domain.

Rule 29.2:

$$\frac{DE \vdash \tau \Rightarrow (\text{topped}, \_)}{DE \vdash \text{Top}_{\tau} \Rightarrow (\top, \tau)}$$

$\top$ of a topped domain.

Rule 29.3:

$$\frac{DE \vdash \tau \Rightarrow (product^{(k)}, \delta^k), \{\tau i = \mathbf{in}DomName(\delta^k \downarrow i), DE \vdash Top_{\tau i} \Rightarrow v_i\}_1^k}{DE \vdash Top_\tau \Rightarrow \mathbf{in}TupleValue(\{v_i\}_1^k)}$$

$\top$ of a product domain.

Rule 29.4:

$$\frac{DE \vdash \tau \Rightarrow (\zeta, \omega), \exists v \in \zeta.\forall \tilde{v} \in \zeta.(\zeta, \omega) \vdash \tilde{v} \leq v}{DE \vdash Top_\tau \Rightarrow \mathbf{in}Str(v)}$$

$\top$ of an enumeration domain.

4.4. Meaning of a SPARE "program"

The meaning of a SPARE "program" is obtained by deriving the following *judgment*:

(J30). $AST \vdash \texttt{program} \Rightarrow v$

Rule 30.1:

$$\frac{\begin{array}{c} DE_{init} \vdash \texttt{specification} \Rightarrow (SD, FD), AST(\text{nonterminal,instance}) = (\pi, IM, SD), \\ VS = \{\$ \mapsto \mathbf{in}NodeInst(\text{nonterminal, instance})\}, \\ (VS, IM, SD, FD, AST) \vdash SD(\text{nonterminal}, \pi, \text{function-name}) \Rightarrow v \end{array}}{\begin{array}{l} (DE_{init}, AST) \vdash \texttt{specification;} \\ \texttt{Interpret(nonterminal,instance,function-name)} \Rightarrow v \end{array}}$$

A SPARE "program" consists of a specification and a call to the pre-defined function `Interpret` [9]. The "program" is executed(or interpreted) in the context of an abstract syntax tree for a program in the target language (i.e. the language in which the programs to be analyzed are written). The specification is first "evaluated" using the static semantic rules. The dynamic semantics provides the rules necessary to evaluate the expression which defines the analysis on the program represented by the abstract syntax tree.

4.5. Meaning of expressions

(J31). $(CS, VS, IM, SD, FD, FS, AST) \vdash \varepsilon \Rightarrow (v, CS')$

The above *judgment* is used to provide the meaning of elements from *TypeCorExpr*. As these elements are obtained through elaboration of the specifications using the rules in the static semantics, they contain only type consistent operations.

As CS is used in only a couple of rules, CS and CS′ are omitted from most of the rules. The convention for restoring them is as follows:

A rule of the form

$$\frac{C_1 \vdash \varepsilon \Rightarrow v_1, \cdots, C_n \vdash \varepsilon \Rightarrow v_n}{C \vdash \varepsilon \Rightarrow v}$$

must be assumed to represent the full form

$$\frac{(CS_0, C_1) \vdash \varepsilon \Rightarrow (v_1, CS_1), \cdots, (CS_{n-1}, C_n) \vdash \varepsilon \Rightarrow (v_n, CS_n)}{(CS_0, C) \vdash \varepsilon \Rightarrow (v, CS_n)}$$

Other predicates (if any) are left unaltered. This convention is to be followed for all judgments that use CS.

4.5.1. Constants

Rule 31.1:

$$\frac{}{(\cdots) \vdash \text{integer–literal} \Rightarrow \textbf{in}\mathit{Int}(\mathit{integer–literal})}$$

Rule 31.2:

$$\frac{}{(\cdots) \vdash \text{boolean–literal} \Rightarrow \textbf{in}\mathit{Bool}(\mathit{boolean–literal})}$$

Rule 31.3:

$$\frac{}{(\cdots) \vdash \text{string–literal} \Rightarrow \textbf{in}\mathit{Str}(\mathit{string–literal})}$$

Rule 31.4:

$$\frac{}{(\cdots) \vdash \textbf{nil} \Rightarrow \textbf{in}\mathit{TupleValue}(\textbf{nil})}$$

Rule 31.5:

$$\frac{}{(\cdots) \vdash \textbf{bottom}\ \tau \Rightarrow \text{Bottom}_\tau}$$

Rule 31.6:

$$\frac{}{(\cdots) \vdash \textbf{top}\ \tau \Rightarrow \text{Top}_\tau}$$

4.5.2. Variables

Rule 31.7:

$$\frac{\$ \in \textbf{Dom}\ VS}{(VS, \cdots) \vdash \$ \Rightarrow VS(\$)}$$

$ is included in the initial environment VS for the evaluation of semantic functions.

Rule 31.8:

$$\frac{\text{identifier} \in \textbf{Dom VS}}{(\text{VS}, \cdots) \vdash \text{identifier} \Rightarrow \text{VS(identifier)}}$$

Rule 31.9:

$$\frac{\text{identifier} \in \textbf{Dom FD}, (\text{FD}, \text{AST}) \vdash \text{FD(identifier)} \Rightarrow \text{v}}{(\text{FD}, \text{AST}, \cdots) \vdash \text{identifier} \Rightarrow \text{v}}$$

identifier is a function name.

Rule 31.10:

$$\frac{}{\begin{array}{l}(\text{IM}, \cdots) \vdash [\![\text{ syntactic–domain–name }]\!] \Rightarrow \\ \textbf{in}\mathit{NodeInst}(\text{syntactic–domain–name, IM(syntactic–domain–name)})\end{array}}$$

The current instance of the syntactic object is obtained from IM.

Rule 31.11:

$$\frac{\begin{array}{c}\text{IM(syntactic–domain–name)} = \iota, \text{AST(syntactic–domain–name, } \iota) = (\pi, \text{IM}'), \\ \text{VS}' = \{\$ \mapsto \textbf{in}\mathit{NodeInst}(\text{syntactic–domain–name, } \iota)\}, \\ (\text{VS}', \text{IM}', \text{SD}, \text{AST}, \cdots) \vdash \text{SD(syntactic–domain–name, } \pi, \text{function–name}) \Rightarrow \text{v}\end{array}}{(\text{VS}, \text{IM}, \text{SD}, \text{AST}, \cdots) \vdash \text{function–name } [\![\text{ syntactic–domain–name }]\!] \Rightarrow \text{v}}$$

IM provides the instance of the syntactic object to be evaluated. AST provides the production for this instance of the syntactic object. The function is evaluated in the new variable environment VS'

SET CONSTRUCTION:

Rule 31.12:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow \text{v}_i\}_1^k}{(\cdots) \vdash \{\,\{\text{expression}_i\}_1^k\,\} \Rightarrow \textbf{in}\mathit{SetValue}(\{\text{v}_1, \cdots, \text{v}_k\})}$$

PRODUCT CONSTRUCTION:

Rule 31.13:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow \text{v}_i\}_1^k}{(\cdots) \vdash (\,\{\text{expression}_i\}_1^k\,) \Rightarrow \textbf{in}\mathit{TupleValue}(\text{v}_1, \cdots, \text{v}_k)}$$

UNARY PREFIX OPERATIONS:

Rule 31.14:

$$\frac{(\cdots) \vdash \text{expression} \Rightarrow v,\ \vdash -v \Rightarrow v'}{(\cdots) \vdash \textbf{minus}\ \text{expression} \Rightarrow v'}$$

Rule 31.15:

$$\frac{(\cdots) \vdash \text{expression} \Rightarrow v,\ \vdash \neg v \Rightarrow v'}{(\cdots) \vdash \textbf{not}\ \text{expression} \Rightarrow v'}$$

Rule 31.16:

$$\frac{(\cdots) \vdash \text{expression} \Rightarrow v,\ \vdash \text{hd}\ v \Rightarrow v'}{(\cdots) \vdash \textbf{head}\ \text{expression} \Rightarrow v'}$$

Rule 31.17:

$$\frac{(\cdots) \vdash \text{expression} \Rightarrow v,\ \vdash \text{tl}\ v \Rightarrow v'}{(\cdots) \vdash \textbf{tail}\ \text{expression} \Rightarrow v'}$$

Rule 31.18:

$$\frac{(\cdots) \vdash \text{expression} \Rightarrow v,\ \vdash \text{null}\ v \Rightarrow v'}{(\cdots) \vdash \textbf{null}\ \text{expression} \Rightarrow v'}$$

Rule 31.19:

$$\frac{(\cdots) \vdash \text{expression} \Rightarrow v}{(\cdots) \vdash (+_{\text{set}}\ \tau)\ \text{expression} \Rightarrow \nabla_\tau\ v}$$

Rule 31.20:

$$\frac{(\cdots) \vdash \text{expression} \Rightarrow v,\ v' = \{v_1 \mid \exists v_2 \in \textbf{Dom}v,\ v(v_2) = v_1\}}{(\cdots) \vdash (+_{\text{store}}\ \tau)\ \text{expression} \Rightarrow \nabla_\tau\ v'}$$

The notation $\nabla_\tau\ v$ denotes the value $v_1\ \nabla_\tau\ v_2\ \nabla_\tau\ \cdots\ \nabla_\tau\ v_k$ where v is the finite set $\{v_1, v_2, \cdots, v_k\}$.

BINARY INFIX OPERATIONS:

Rule 31.21:

$$\frac{(\cdots) \vdash \text{expression} \Rightarrow v,\ \vdash v\downarrow\text{integer–literal} \Rightarrow v'}{(\cdots) \vdash \text{expression}\ \hat{}\ \text{integer–literal} \Rightarrow v'}$$

Rule 31.22:

$$\frac{(\cdots) \vdash \text{expression} \Rightarrow v,\ \vdash v\ \textbf{is}\ \text{domain–name} \Rightarrow v'}{(\cdots) \vdash \text{expression}\ \textbf{is}\ \text{domain–name} \Rightarrow v'}$$

Rule 31.23:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash v_1 \wedge v_2 \Rightarrow v'}{(\cdots) \vdash \text{expression}_1 \textbf{ and } \text{expression}_2 \Rightarrow v'}$$

Rule 31.24:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash v_1 \vee v_2 \Rightarrow v'}{(\cdots) \vdash \text{expression}_1 \textbf{ or } \text{expression}_2 \Rightarrow v'}$$

Rule 31.25:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash v_1 \in v_2 \Rightarrow v'}{(\cdots) \vdash \text{expression}_1 \textbf{ in } \text{expression}_2 \Rightarrow v'}$$

Rule 31.26:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash v_1 \subseteq v_2 \Rightarrow v'}{(\cdots) \vdash \text{expression}_1 <= \text{expression}_2 \Rightarrow v'}$$

Rule 31.27:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2}{(\cdots) \vdash \text{expression}_1\ (+\tau)\ \text{expression}_2 \Rightarrow v_1\ \nabla_\tau\ v_2}$$

Rule 31.28:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash v_1 \cap v_2 \Rightarrow v'}{(\cdots) \vdash \text{expression}_1 * \text{expression}_2 \Rightarrow v'}$$

Rule 31.29:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash v_1 - v_2 \Rightarrow v'}{(\cdots) \vdash \text{expression}_1 - \text{expression}_2 \Rightarrow v'}$$

Rule 31.30:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash \text{Cons}(v_1, v_2) \Rightarrow v'}{(\cdots) \vdash \text{expression}_1 :: \text{expression}_2 \Rightarrow v'}$$

Rule 31.31:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash \text{Append}(v_1, v_2) \Rightarrow v'}{(\cdots) \vdash \text{expression}_1 \,\&\, \text{expression}_2 \Rightarrow v'}$$

Rule 31.32:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2}{(\cdots) \vdash \text{expression}_1\ (=\tau)\ \text{expression}_2 \Rightarrow \textbf{in}\mathit{Bool}(v_1 =_\tau v_2)}$$

The equality operation may be redefined on domains that have explicit specification of ordering. We will assume that the implementation uses the definition provided by the ordering in such cases.

BINARY PREFIX OPERATIONS:

Rule 31.33:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash v_1 + v_2 \Rightarrow v'}{(\cdots) \vdash \textbf{add}\ \text{expression}_1\ \text{expression}_2 \Rightarrow v'}$$

Rule 31.34:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash v_1 - v_2 \Rightarrow v'}{(\cdots) \vdash \textbf{sub}\ \text{expression}_1\ \text{expression}_2 \Rightarrow v'}$$

Rule 31.35:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash v_1 \times v_2 \Rightarrow v'}{(\cdots) \vdash \textbf{mult}\ \text{expression}_1\ \text{expression}_2 \Rightarrow v'}$$

Rule 31.36:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ \vdash v_1 / v_2 \Rightarrow v'}{(\cdots) \vdash \textbf{div}\ \text{expression}_1\ \text{expression}_2 \Rightarrow v'}$$

UNARY POSTFIX OPERATIONS:

Rule 31.37:

$$\frac{(\cdots) \vdash \text{variable} \Rightarrow v}{(\cdots) \vdash \text{variable}'\textbf{label} \Rightarrow v}$$

The label domain is mapped to the domain of *NodeInst*.

Rule 31.38:

$$\frac{(CS, \cdots) \vdash \text{variable} \Rightarrow (v, CS'),\ v \in \textbf{Dom}\ CS',\ CS'(v) = v'}{(CS, \cdots) \vdash \text{variable}'\textbf{cache} \Rightarrow (v', CS')}$$

MIXFIX OPERATIONS:

Rule 31.39:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ v_2 \notin \textbf{Dom}\ v_1}{(\cdots) \vdash \text{expression}_1[\ \text{expression}_2\ ] \Rightarrow \text{Bottom}_\tau}$$

Rule 31.40:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\ v_2 \in \textbf{Dom}\ v_1}{(\cdots) \vdash \text{expression}_1[\ \text{expression}_2\ ] \Rightarrow v_1(v_2)}$$

Rule 31.41:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^3}{(\cdots) \vdash \text{expression}_1[\ \text{expression}_2/\text{expression}_3\ ] \Rightarrow \{v_3 \mapsto v_2\}/v_1}$$

Rule 31.42:

$$\frac{\begin{array}{c}\{(\text{VS}, \cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2,\\ v = \{v_4 \mapsto v_3 \mid \exists v' \in v_2, \{(\{\text{identifier} \mapsto v'\}/\text{VS}, \cdots) \vdash \text{expression}_i \Rightarrow v_i\}_3^4\}\end{array}}{\begin{array}{l}(\text{VS}, \cdots) \vdash \text{expression}_1[\ \textbf{foreach}\ \text{identifier}\ \textbf{in}\ \text{expression}_2\ \textbf{do}\\ \qquad\qquad \text{expression}_3/\text{expression}_4\ ] \Rightarrow v/v_1\end{array}}$$

The multiple update consists of a sequence of single updates with the identifier successively bound to each element of the set that expression_2 evaluates to.

4.5.3. Lambda expression

Rule 31.43:

$$\frac{}{\begin{array}{l}(\text{VS}, \text{IM}, \cdots) \vdash \textbf{lambda}\ [\textbf{collect}\ \text{C}\ \tau]\ \text{identifier}\ .\ \text{expression} \Rightarrow\\ \qquad \textbf{in}\mathit{LambClosure}(\text{expression}, \text{identifier}, \text{VS}, \text{IM}, [(\text{c}, \tau)])\end{array}}$$

A lambda expression evaluates to a closure.

Rule 31.44:

$$\frac{}{\begin{array}{l}(\text{VS}, \text{IM}, \cdots) \vdash \textbf{lambda strict}\ \tau_1\ \tau_2\ [\textbf{collect}\ \text{C}\ \tau]\ \text{identifier}\ .\ \text{expression} \Rightarrow\\ \qquad \textbf{in}\mathit{StrLambClosure}(\text{expression}, \text{identifier}, \text{VS}, \text{IM}, \tau_1, \tau_2, [(\text{c}, \tau)])\end{array}}$$

Closure for a strict lambda expression. τ_1 is the type of the argument and τ_2 is the type of the expression.

4.5.4. Function application

Rule 31.45:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2, (\cdots) \vdash \textbf{apply}\ (v_1, v_2) \Rightarrow v}{(\cdots) \vdash \text{expression}_1\ \ \text{expression}_2 \Rightarrow v}$$

The **Apply** function is defined in Section 4.4.14.

4.5.5. Expression composition

Rule 31.46:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2}{(\cdots) \vdash \text{expression}_1\ .\ \text{expression}_2 \Rightarrow \textbf{in}\mathit{FuncComp}(v_1, v_2)}$$

4.5.6. Expression union

Rule 31.47:

$$\frac{\{(\cdots) \vdash \text{expression}_i \Rightarrow v_i\}_1^2}{(\cdots) \vdash \text{expression}_1\ (\oplus \tau)\ \text{expression}_2 \Rightarrow \textbf{in}\mathit{FuncUnion}(v_1, v_2, \tau)}$$

4.5.7. Conditional expression

Rule 31.48:

$$\frac{(\cdots) \vdash \text{expression}_1 \Rightarrow \text{true}, (\cdots) \vdash \text{expression}_2 \Rightarrow v_2}{(\cdots) \vdash \text{expression}_1 \text{ --> } \text{expression}_2\ ,\ \text{expression}_3 \Rightarrow v_2}$$

Rule 31.49:

$$\frac{(\cdots) \vdash \text{expression}_1 \Rightarrow \text{false}, (\cdots) \vdash \text{expression}_3 \Rightarrow v_3}{(\cdots) \vdash \text{expression}_1 \text{ --> } \text{expression}_2\ ,\ \text{expression}_3 \Rightarrow v_3}$$

4.5.8. Fixed point expression

Rule 31.50:

$$\frac{(\{\text{function–name} \mapsto \text{expression}\}/\text{FD}, \cdots) \vdash \text{expression} \Rightarrow v}{(\text{FD}, \cdots) \vdash \textbf{fix}\ \text{function–name}\ .\ \text{expression} \Rightarrow v}$$

4.5.9. Let expression

Rule 31.51:

$$\frac{(VS, \cdots) \vdash \{\text{let-clause}_i\}_1^k \Rightarrow VS', (VS', \cdots) \vdash \text{expression} \Rightarrow v}{(VS, \cdots) \vdash \textbf{let}\ \{\text{let-clause}_i\}_1^k\ \textbf{in}\ \text{expression} \Rightarrow v}$$

The body of the let expression is evaluated in an environment augmented by the let clauses.

Rule 31.52:

$$\frac{VS_0 = VS, \{(VS_{i-1}, \cdots) \vdash \text{let-clause}_i \Rightarrow VS_i\}_1^k}{(VS, \cdots) \vdash \{\text{let-clause}_i\}_1^k \Rightarrow VS_k}$$

The let clauses are evaluated sequentially. Each let clause is evaluated in an environment augmented by the previous (if any) let clauses.

(J32). $(CS, FS, VS, IM, SD, FD, AST) \vdash \text{let-clause} \Rightarrow (VS', CS')$

Rule 32.1:

$$\frac{(VS, \cdots) \vdash \text{expression} \Rightarrow v}{(VS, \cdots) \vdash \text{identifier} = \text{expression} \Rightarrow \{\text{identifier} \mapsto v\}/VS}$$

4.5.10. Typecast expression

Rule 31.2:

$$\frac{(\cdots) \vdash \text{expression} \Rightarrow v}{(\cdots) \vdash \text{domain-name expression} \Rightarrow \textbf{in}\mathit{UnionValue}(\text{domain-name}, v)}$$

4.5.11. Parenthesized expression

Rule 31.3:

$$\frac{(\cdots) \vdash \text{expression} \Rightarrow v}{(\cdots) \vdash (\text{expression}) \Rightarrow v}$$

4.5.12. Collect expression

Rule 31.4:

$$\frac{(CS, VS, \cdots) \vdash \text{expression} \Rightarrow (v, CS'), v \vdash C \Rightarrow v', VS(\$) \notin \textbf{Dom}\ CS'}{(CS, VS, \cdots) \vdash \textbf{collect}\ C\ \tau\ \text{expression} \Rightarrow (v, \{VS(\$) \mapsto v'\}/CS')}$$

The cache is being updated for the first time for the instance of the syntactic object for which the semantic function is currently being evaluated.

Rule 31.5:

$$\frac{(CS, VS, \cdots) \vdash \text{expression} \Rightarrow (v, CS'),\ v \vdash C \Rightarrow v',\ VS(\$) \in \textbf{Dom}\ CS'}{(CS, VS, \cdots) \vdash \textbf{collect}\ C\ \tau\ \text{expression} \Rightarrow (v, \{VS(\$) \mapsto v'\ \nabla_\tau\ CS'(VS(\$))\}/CS')}$$

The cache is updated for the instance of the syntactic object for which the semantic function is currently being evaluated. The value to be collected is "joined" with the previous value in the cache associated with this instance of the syntactic object.

(J33). $v \vdash C \Rightarrow v'$

The value to be collected is derived from the result value in accordance with the collect option specification for the function.

Rule 33.1:

$$\frac{v \vdash \text{selected–domain} \Rightarrow v'}{v \vdash \text{selected–domain} \Rightarrow v'}$$

This strange rule arises from the overloading of the $\vdash$ operator.

Rule 33.2:

$$\frac{v \vdash \text{selected–domain} \Rightarrow v'}{v \vdash \textbf{powerset of}\ \text{selected–domain} \Rightarrow \textbf{in}\mathit{SetValue}(\{v'\})}$$

The value to be collected is the set containing either the result value or one of its components.

(J34). $v \vdash \text{selected–domain} \Rightarrow v'$

Rule 34.1:

$$\frac{}{v \vdash \text{domain–name} \Rightarrow v}$$

Rule 34.2:

$$\frac{}{v \vdash \text{domain–name integer} \Rightarrow v{\downarrow}\text{integer}}$$

The value to be collected is derived from a component of the result value.

4.5.13. Fixed point test expression

Fixed point expressions are modified during static evaluation of the specification to include fixed point checks. The check is inserted at the beginning of the expression that forms the body of the fixed point expression after all the λs corresponding to the arguments of the function have been stripped.

Rule 31.3:

$$\frac{\beta < 0,\ TT = \{\},(\cdots) \vdash \text{expression} \Rightarrow v}{(FS, ,VS, \cdots) \vdash \textbf{fixtest}\ \phi\ TT\ \beta\ \tau\ \text{expression} \Rightarrow v)}$$

Termination option was not used in the fixed point expression. No checks need to be made.

Rule 31.4:

$$\frac{FX = (VS(\$), \phi),\ FX \notin \textbf{Dom}\ FS,\ (\{FX \mapsto (\{\}, \beta)\}/FS, VS, \cdots) \vdash \text{expression} \Rightarrow v}{(FS, ,VS, \cdots) \vdash \textbf{fixtest}\ \phi\ TT\ \beta\ \tau\ \text{expression} \Rightarrow v)}$$

The fixed point expression is being evaluated for the first time. No checks need to be made.

FS contains information from the previous evaluation. As there may be more than one instance of the same fixed point function "active" at any point, the various instances are identified by the tuple containing the node instance for which the fixed point evaluation is being carried out and the function name used for the fixed point expression.

Rule 31.5:

$$\frac{FX = (VS(\$), \phi),\ FX \in \textbf{Dom}\ FS,\ \textit{Bound}\ \textbf{of}\ FS(FX) = \beta',\ \beta = 0}{(FS, VS, \cdots) \vdash \textbf{fixtest}\ \phi\ TT\ \beta\ \tau\ \text{expression} \Rightarrow \text{Top}_\tau}$$

The bound on the number of iterations in the fixed point evaluation is reached.

Rule 31.6:

$$\frac{FX = (VS(\$), \phi),\ FX \in \textbf{Dom}\ FS,\ \textbf{is}\textit{ArgTest}(TT),\ FS(FX) = (AS, \beta'),\ \beta' > 0,\ \forall x \in TT.\ VS(x) = AS(x)}{(FS, VS, \cdots) \vdash \textbf{fixtest}\ \phi\ TT\ \beta\ \tau\ \text{expression} \Rightarrow \text{Bottom}_\tau}$$

Fixed point termination on argument values was specified for this expression. TT contains the set of lambda variables that have been bound to the arguments of the function. The arguments to the current evaluation are the same as the arguments to the previous evaluation.

Rule 31.7:

$$\frac{FX = (VS(\$), \phi),\ FX \in \textbf{Dom}\ FS,\ \textbf{is}\textit{ArgTest}(T),\ FS(FX) = (AS, \beta'),\ \beta' > 0,\ \exists x \in TT.\ VS(x) \neq AS(x)\quad (\{FX \mapsto (\{x \mapsto VS(x) \mid x \in TT\}, \beta'-1)/FS, VS, \cdots) \vdash \text{expression} \Rightarrow v}{(FS, VS, \cdots) \vdash \textbf{fixtest}\ \phi\ TT\ \beta\ \tau\ \text{expression} \Rightarrow v}$$

Fixed point termination on argument values was specified for this expression. The arguments to the current evaluation is different from the arguments to the previous evaluation.

Rule 31.8:

$$\frac{FX = (VS(\$), \phi),\ FX \in \textbf{Dom}\ FS,\ \textbf{is}\mathit{CacheTest}(TT),\ FS(FX) = (v, \beta'),\ \beta' > 0,\ CS(VS(\$)) =_{\tau'} v}{(CS, FS, VS, \cdots) \vdash \textbf{fixtest}\ \phi\ TT\ \beta\ \tau\ expression \Rightarrow Bottom_{\tau}}$$

Fixed point termination on the cache value was specified for this expression. The current cache value is the same as the cache value during the previous evaluation.

Rule 31.9:

$$\frac{FX = (VS(\$), \phi),\ FX \in \textbf{Dom}\ FS,\ \textbf{is}\mathit{CacheTest}(TT),\ FS(FX) = (v, \beta'),\ \beta' > 0,\ \neg(CS(VS(\$)) =_{\tau'} v),\ (CS, \{FX \mapsto (CS(VS(\$)), \beta'-1)/FS, VS, \cdots) \vdash expression \Rightarrow v'}{(CS, FS, VS, \cdots) \vdash \textbf{fixtest}\ \phi\ TT\ \beta\ \tau\ expression \Rightarrow v')}$$

Fixed point termination on the cache value was specified for this expression. The current cache value is different from the cache value during the previous evaluation.

4.5.14. Definition of "apply"

(J32). $(CS, VS, IM, SD, FD, AST) \vdash \textbf{apply}\ (v_1, v_2) \Rightarrow (v, CS')$

Rule 32.1:

$$\frac{v_1 =_{\tau} Bottom_{\tau},\ \tau = (\text{function}^{(k)}, (\delta_1, \cdots, \delta_k)),\ \tau' = \textbf{in}\mathit{DomComp}(\text{function}^{(k-1)}, (\delta_2, \cdots, \delta_k))}{(\cdots) \vdash \textbf{apply}\ (v_1, v_2) \Rightarrow Bottom_{\tau'}}$$

$\perp$ function applied on any argument returns $\perp$.

Rule 32.2:

$$\frac{v_1 =_{\tau} Top_{\tau},\ \tau = (\text{function}^{(k)}, (\delta_1, \cdots, \delta_k)),\ \tau' = \textbf{in}\mathit{DomComp}(\text{function}^{(k-1)}, (\delta_2, \cdots, \delta_k))}{(\cdots) \vdash \textbf{apply}\ (v_1, v_2) \Rightarrow Top_{\tau'}}$$

$\top$ function applied on any argument returns $\top$.

Rule 32.3:

$$\frac{\textbf{is}\mathit{LambClosure}(v_1),\ v_1 = (\varepsilon, \alpha, VS', IM', CI),\ (\{\alpha \mapsto v_2\}/VS', IM', \cdots) \vdash \varepsilon \Rightarrow v,\ (CS, VS, v_2) \vdash CI \Rightarrow CS'}{(CS, VS, IM, \cdots) \vdash \textbf{apply}\ (v_1, v_2) \Rightarrow (v, CS')}$$

Application of a lambda closure on an argument. The expression in the closure is evaluated with the lambda variable bound to the argument. The environment for evaluation is obtained from the closure. The cache is updated if the collection option specified this argument. The rules for the cache update are provided below.

Rule 32.4:

$$\frac{\mathbf{is}\mathit{StrLambClosure}(v_1),\ v_1 = (\varepsilon, \alpha, VS', IM', \tau, \tau', CI),\ v_2 =_\tau \mathrm{Bottom}_\tau,\ (CS, VS, v_2) \vdash CI \Rightarrow CS'}{(CS, VS, IM, \cdots) \vdash \mathbf{apply}\ (v_1, v_2) \Rightarrow (\mathrm{Bottom}_{\tau'}, CS')}$$

Application of a lambda closure on an argument. The expression is specified to be strict on this argument. The argument is $\perp$.

Rule 32.5:

$$\frac{\mathbf{is}\mathit{StrLambClosure}(v_1),\ v_1 = (\varepsilon, \alpha, VS', IM', \tau, \tau', CI),\ \neg(v_2 =_\tau \mathrm{Bottom}_\tau),\ (\{\alpha \mapsto v_2\}/VS', IM', \cdots) \vdash \varepsilon \Rightarrow v,\ (CS, VS, v_2) \vdash CI \Rightarrow CS'}{(CS, VS, IM, \cdots) \vdash \mathbf{apply}\ (v_1, v_2) \Rightarrow (v, CS')}$$

Application of a lambda closure on an argument. The expression is specified to be strict on this argument. The argument is not $\perp$.

Rule 32.6:

$$\frac{\mathbf{is}\mathit{FuncComp}(v_1),\ v_1 = (v'_1, v'_2),\ (\cdots) \vdash \mathbf{apply}\ (v'_2, v_2) \Rightarrow v_3,\ (\cdots) \vdash \mathbf{apply}\ (v'_1, v_3) \Rightarrow v_4}{(\cdots) \vdash \mathbf{apply}\ (v_1, v_2) \Rightarrow v_4}$$

Application of a function composition on an argument.

Rule 32.7:

$$\frac{\mathbf{is}\mathit{FuncUnion}(v_1),\ v_1 = (v'_1, v'_2, \tau),\ (\cdots) \vdash \mathbf{apply}\ (v'_2, v_2) \Rightarrow v_3,\ (\cdots) \vdash \mathbf{apply}\ (v'_1, v_2) \Rightarrow v_4,\ \vdash v_3\ \nabla_\tau\ v_4 \Rightarrow v}{(\cdots) \vdash \mathbf{apply}\ (v_1, v_2) \Rightarrow v}$$

Application of a function union on an argument.

(J33). $(CS, VS, v) \vdash CI \Rightarrow CS'$

The rules for updating the cache are similar to the ones for collect expressions (Section 4.4.12).

Rule 33.1:

$$\frac{}{(CS, VS, v) \vdash \{\} \Rightarrow CS}$$

Collection not specified for this argument.

Rule 33.2:

$$\frac{v \vdash C \Rightarrow v',\ VS(\$) \notin \mathbf{Dom}\ CS'}{(CS, VS, v) \vdash (C, \tau) \Rightarrow \{VS(\$) \mapsto v'\}/CS}$$

Value collected for the first time for the current syntactic object. The first *judgment* in the hypothesis is defined as **J33** in Section 4.4.12.

Rule 33.3:

$$\frac{\nu \vdash C \Rightarrow \nu',\ VS(\$) \in \textbf{Dom}\ CS'}{(CS, VS, \nu) \vdash (C, \tau) \Rightarrow \{VS(\$) \mapsto \nu' \nabla_\tau CS(VS(\$))\}/CS}$$

The value to be collected is "joined" with the previous value in the cache.

Appendix 1. Syntax used for formal semantics description

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

domain-declaration-section ::= $\{ \text{domain-declaration}_i ; \}_1^k$

domain-declaration ::= $\{ \text{domain-name}_i \}_1^k$
= 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 ::= $\{ \text{string-literal}_i \}_1^k$

lifted-domain ::= **lifted** domain-name

topped-domain ::= **topped** domain-name

product-domain ::= domain-name $\{ * \text{domain-name}_i \}_1^k$

union-domain ::= domain-name { + domain–name$_i$ }$_1^k$

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

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

store-domain ::= **store** domain–name$_1$ --> domain–name$_2$

function-domain ::= **function** { domain–name$_i$ }$_1^k$

ordering-option ::= **ordered by** ordering

ordering ::= order-enumeration
| order-name

order-enumeration ::= ({ order–tuple$_i$ }$_1^k$)

order-tuple ::= < string–literal$_1$, string–literal$_2$ >

aux-function-section ::= { aux–function$_i$ }$_1^k$

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

semantic-func-decl-section ::= { semantic–func–decl$_i$ }$_1^k$

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

function-type-declaration ::= { [**strict**] domain–name$_i$ }$_1^k$ --> domain–name$_{k+1}$

collect-option ::= **collect** collect-domain

collect-domain ::= [**powerset of**] selected-domain

selected-domain ::= domain-name [^ integer]

semantic-func-defn-section ::= $\{ \text{semantic-equation}_i \}_1^k$

semantic-equation ::= function-name "[[" (production) "]]" =
expression

production ::= production-name $\{ \text{syntactic-domain-name}_i \}_1^k$

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	::=	$\{ \{ \text{expression}_i \}_1^k \}$
product-construction	::=	$(\{ \text{expression}_i \}_1^k)$
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_1 binary-infix-op expression_2
binary-prefix-operation	::=	binary-prefix-op expression_1 expression_2
unary-postfix-operation	::=	variable ' unary-postfix-op
mix-fix-operation	::=	expression "[" store-expression "]"
store-expression	::=	expression
	|	expression_1 / expression_2
	|	**foreach** identifier **in** expression_1 **do** expression_2 / expression_3
lambda-expression	::=	**lambda** identifier . expression

function-application	::=	expression_1 expression_2
expression-composition	::=	expression_1 . expression_2
expression-union	::=	expression_1 + expression_2
conditional-expression	::=	expression_1 --> expression_2 , expression_3
fixed-point-expression	::=	**fix** termination-option function-name . expression
termination-option	::=	[test-option] [bound-option]
test-option	::=	**argument** | **cache**
bound-option	::=	**bounded** (integer-literal)
let-expression	::=	**let** $\{ \text{let–clause}_i \}_1^k$ **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**

References

1. D. Clement, "The Natural Dynamic Semantics of Mini-Standard ML," pp. 67-81 in *TAPSOFT '87 (Proceedings of the International Joint Conference on Theory and Practice of Software Development) Volume 2*, (March 1987).

2. J. Despeyroux, "Proof of translation in Natural Semantics," *Logic in Computer Science*, pp. 193-205 (June 1986).

3. T. Despeyroux, "Executable specification of static semantics," in *Semantics of data types*, (June 1984).

4. R. Harper, R. Milner, and M. Tofte, "The Semantics of Standard ML - Version 1," ECS-LFCS-87-36, University of Edinburgh (Aug 1987).

5. G. Kahn, "Natural Semantics," *Proc. of Symp. on Theoretical Aspects of Computer Science*, (Feb 1987).

6. G. D. Plotkin, "A structural approach to operational semantics," DAIMI FN-19, Aarhus University (Sept 1981).

7. D. Prawitz, *Natural deduction - A proof-theoretical study*, Almqvist & Wiksell, Stockholm (1965).

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

9. G. A. Venkatesh and C. N. Fischer, "SPARE: Reference Manual," Tech. Report #850, University of Wisconsin-Madison (June 1989).