Syntax and Semantics of the cat Language

Jade Alglave, Patrick Cousot, and Luc Maranget
© 2016 HSA Foundation. All rights reserved.

The contents of this document are provided in connection with the HSA Foundation specifications. This specification is protected by copyright laws and contains material proprietary to the HSA Foundation. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of HSA Foundation. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

HSA Foundation grants express permission to any current Founder, Promoter, Supporter Contributor, Academic or Associate member of HSA Foundation to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be re-formatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the HSA Foundation web-site should be included whenever possible with specification distributions.

HSA Foundation makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. HSA Foundation makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the HSA Foundation, or any of its Founders, Promoters, Supporters, Academic, Contributors, and Associates members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.
Acknowledgments

This document is the result of the contributions of many people. Here is a partial list of the contributors, including the companies that they represented at the time of their contribution.

INRIA
- Luc Maranget

Microsoft Research Cambridge and University College London
- Jade Alglave

New York University and Ecole Normale Superieure
- Patrick Cousot
# Contents

## Acknowledgments

## About this document: Syntax and Semantics of the cat Language

**Audience** ................................................................. 7
**HSA Information Sources** ........................................... 7

1. Specifications and statements ...................................... 8
   1.1 Evaluating a cat specification ................................ 8
   1.2 Specifications .................................................... 8
   1.3 Statements ....................................................... 9

2. Typed values and semantic domains ............................... 10
   2.1 Typed values ..................................................... 10
   2.2 Auxiliaries ....................................................... 12

3. Expressions .............................................................. 13
   3.1 Identifiers ........................................................ 14
      3.1.1 Predefined identifiers denoting sets of events ..... 15
      3.1.2 Predefined identifiers denoting relations on events . 16
      3.1.3 Primitives to manipulate sets and relations over events 17
   3.2 Tags ................................................................. 18
      3.2.1 Tags ........................................................... 18
      3.2.2 Declarations ............................................... 19
      3.2.3 Scopes ........................................................ 20
      3.2.4 Matching over tags ........................................ 20
   3.3 Functions and procedures ...................................... 21
      3.3.1 Functions .................................................... 21
      3.3.2 Function calls ............................................ 22
      3.3.3 Procedures .................................................. 22
      3.3.4 Procedure calls .......................................... 23
   3.4 Sets and tuples .................................................. 24
      3.4.1 Sets .......................................................... 24
      3.4.2 Matching over sets ....................................... 25
      3.4.3 Tuples ....................................................... 26
      3.4.4 Grouping ................................................... 26
   3.5 Bindings ............................................................ 26
      3.5.1 Binding definitions ....................................... 27
      3.5.2 Binding expressions ....................................... 28
   3.6 Operators on sets and relations ................................ 29
      3.6.1 Unary operators .......................................... 29
      3.6.2 Binary operators .......................................... 30

4. Constraints .............................................................. 33
   4.1 Checks ........................................................... 33
   4.2 Iteration over sets ............................................. 34
5. Requirements .............................................................................................................. 36
  5.1 Statements .............................................................................................................. 36
  5.2 Quantification via with binding ............................................................................. 36

6. References .................................................................................................................. 38

Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Semantics of specifications</td>
<td>8</td>
</tr>
<tr>
<td>1-2</td>
<td>Semantics of statements</td>
<td>9</td>
</tr>
<tr>
<td>2-1</td>
<td>Typed values</td>
<td>10</td>
</tr>
<tr>
<td>2-2</td>
<td>Semantic domains</td>
<td>11</td>
</tr>
<tr>
<td>2-3</td>
<td>Auxiliaries used to define operators' semantics</td>
<td>12</td>
</tr>
<tr>
<td>3-1</td>
<td>Simple expressions</td>
<td>13</td>
</tr>
<tr>
<td>3-2</td>
<td>Match clauses</td>
<td>13</td>
</tr>
<tr>
<td>3-3</td>
<td>Expressions</td>
<td>14</td>
</tr>
<tr>
<td>3-4</td>
<td>Definitions</td>
<td>14</td>
</tr>
<tr>
<td>3-5</td>
<td>Reserved identifiers</td>
<td>14</td>
</tr>
<tr>
<td>3-6</td>
<td>Semantics of identifiers</td>
<td>15</td>
</tr>
<tr>
<td>3-7</td>
<td>Predefined sets of events</td>
<td>15</td>
</tr>
<tr>
<td>3-8</td>
<td>Semantics of predefined sets</td>
<td>16</td>
</tr>
<tr>
<td>3-9</td>
<td>Predefined relations over events</td>
<td>16</td>
</tr>
<tr>
<td>3-10</td>
<td>Semantics of predefined relations</td>
<td>17</td>
</tr>
<tr>
<td>3-11</td>
<td>Semantics of primitives</td>
<td>17</td>
</tr>
<tr>
<td>3-12</td>
<td>Tags</td>
<td>18</td>
</tr>
<tr>
<td>3-13</td>
<td>Auxiliary over tags 'id'</td>
<td>18</td>
</tr>
<tr>
<td>3-14</td>
<td>Value of tag 'id'</td>
<td>18</td>
</tr>
<tr>
<td>3-15</td>
<td>Gathering all events bearing the same tag 'id'</td>
<td>18</td>
</tr>
<tr>
<td>3-16</td>
<td>Declarations</td>
<td>19</td>
</tr>
<tr>
<td>3-17</td>
<td>Semantics of enum</td>
<td>19</td>
</tr>
<tr>
<td>3-18</td>
<td>Semantics of instructions</td>
<td>19</td>
</tr>
<tr>
<td>3-19</td>
<td>Building the scope instance of level tag</td>
<td>20</td>
</tr>
<tr>
<td>3-20</td>
<td>Tag matching</td>
<td>21</td>
</tr>
<tr>
<td>3-21</td>
<td>Anonymous functions</td>
<td>21</td>
</tr>
<tr>
<td>3-22</td>
<td>Patterns</td>
<td>21</td>
</tr>
<tr>
<td>3-23</td>
<td>Values of functions</td>
<td>22</td>
</tr>
<tr>
<td>3-24</td>
<td>Semantics of function calls</td>
<td>22</td>
</tr>
<tr>
<td>3-25</td>
<td>Procedures</td>
<td>23</td>
</tr>
<tr>
<td>3-26</td>
<td>Declarations of procedures</td>
<td>23</td>
</tr>
<tr>
<td>3-27</td>
<td>Semantics of procedure calls</td>
<td>24</td>
</tr>
<tr>
<td>3-28</td>
<td>Sets</td>
<td>24</td>
</tr>
<tr>
<td>3-29</td>
<td>Semantics of sets</td>
<td>25</td>
</tr>
<tr>
<td>3-30</td>
<td>Set matching</td>
<td>25</td>
</tr>
<tr>
<td>3-31</td>
<td>Tuples</td>
<td>26</td>
</tr>
<tr>
<td>3-32</td>
<td>Semantics of tuples</td>
<td>26</td>
</tr>
<tr>
<td>3-33</td>
<td>Semantics of grouping</td>
<td>26</td>
</tr>
<tr>
<td>3-34</td>
<td>Bindings</td>
<td>27</td>
</tr>
<tr>
<td>3-35</td>
<td>Value of a binding</td>
<td>27</td>
</tr>
<tr>
<td>3-36</td>
<td>Binding definitions, including recursive function binding and recursive set/relation binding</td>
<td>28</td>
</tr>
<tr>
<td>3-37</td>
<td>Binding expressions</td>
<td>29</td>
</tr>
<tr>
<td>3-38</td>
<td>Unary and binary operators</td>
<td>29</td>
</tr>
<tr>
<td>3-39</td>
<td>Unary operators on relations</td>
<td>30</td>
</tr>
<tr>
<td>3-40</td>
<td>Complement of a set or relation</td>
<td>30</td>
</tr>
<tr>
<td>3-41</td>
<td>Sequence of relations</td>
<td>31</td>
</tr>
</tbody>
</table>
Figure 3–42 Adding an element to a set ................................................................. 31
Figure 3–43 Cartesian product of two sets of events ............................................ 31
Figure 3–44 Binary operators relative to both sets and relations ......................... 32
Figure 4–1 Constraints ......................................................................................... 33
Figure 4–2 Value of checks .................................................................................. 33
Figure 4–3 Checks ............................................................................................... 33
Figure 4–4 Check condition on relation ............................................................... 34
Figure 4–5 cat check relation ............................................................................... 34
Figure 4–6 Iteration over sets ............................................................................. 35
Figure 4–7 cat iterate ......................................................................................... 35
Figure 5–1 Requirements ..................................................................................... 36
Figure 5–2 Semantics of requirements ................................................................. 36
Figure 5–3 Verdict at top level ............................................................................ 37
About this document: Syntax and Semantics of the cat Language

This document describes the syntax and semantics of the cat language.

The cat language is inspired by OCaml\(^1\), featuring immutable bindings, first-class functions, and pattern matching. However, cat is a domain-specific language with important differences from OCaml:

- Base values are specialized into sets of events, relations over events, first-class-functions and tags (akin to C enumerations or OCaml constant constructors). There are also two structured values: sets of values and tuples of values.
- The cat language distinguishes among expressions that evaluate to some value, statements that introduce new definitions or constraints, and requirements that introduce quantifiers over statements.

The following notations are used:

- Square brackets \([\ldots]\) to denote optional components
- Parentheses \((\ldots)\) to denote grouping
- \((\ldots)*\) (resp. \((\ldots)^*\)) to denote zero, one, or several (resp. one or several) repetitions of the enclosed components

Audience

This document is written for system and component architects who are interested in supporting the HSA infrastructure (hardware and software) within platform designs.

HSA Information Sources


\(^1\)Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jerôme Vouillon. The OCaml system, release 4.02, Documentation and user’s manual. caml.inria.fr, 24 September 2014.
1. Specifications and statements

Specifications (or cat files) filter candidate executions. In other words, the semantics of a cat specification are defined with respect to a candidate execution.

1.1 Evaluating a cat specification

Evaluating a cat specification means allowing or forbidding that candidate execution. More precisely, evaluating a cat file makes a result object \((j, f, \rho)\) evolve, where:

- \(j\) is a judgment, allowed or forbidden
- \(f\) is a set of flags that is recorded to signal certain executions (e.g., executions with data races)
- \(\rho\) is an environment that is augmented with new definitions as evaluation progresses

Initially, the judgment is allowed, the set of flags is empty, and predefined identifiers are bound to event sets and relations over events as described in 3.1 Identifiers (on page 14).

1.2 Specifications

Specifications are lists of requirements preceded by an identifier. The syntax and semantics of specifications are shown in Figure 1–1 (below).

![Figure 1–1 Semantics of specifications](image)

---

Requirements are evaluated in sequence until one requirement raises error or forbidden or until the end of the requirement list. In that latter case, the specification accepts the candidate execution and raises allowed. The semantics of a cat specification cat is the set of candidate executions \(X\) such that evaluating the cat specification cat over \(X\) raises the judgment allowed:

\[\mathcal{O}[\text{cat}] \triangleq \{ X \in \text{Candidate} \mid \exists f \in \mathcal{F}. \mathcal{O}[\text{cat}] \ X = (\text{allowed}, f) \]
1. Specifications and statements  1.3 Statements

Requirements are lists of two object types: statements or with x from S (a special quantifier). The special quantifier is described in 5.2 Quantification via with binding (on page 36).

1.3 Statements

Statements are evaluated for their effects: adding new definitions or checking constraints. Figure 1–2 (below) lists semantics of statements. Note that once an error has been raised, that state is retained.

<table>
<thead>
<tr>
<th>Figure 1–2 Semantics of statements</th>
</tr>
</thead>
</table>

**Statements** ---

\[
\begin{align*}
\text{statement} & \in \text{Statement} \\
\text{statement} & ::= \text{definition} \\
& \quad \mid \text{constraint}
\end{align*}
\]

\[
\begin{align*}
\overline{\circ}\left[\text{statement} \{\text{statement}\}^+]\ X \langle j, f, \rho \rangle & \triangleq \\
\quad \text{let} \ \langle j', f', \rho' \rangle = \overline{\circ}\left[\text{statement}\right] X \langle j, f, \rho \rangle \ \text{in} \\
\quad \quad \text{if} \ j' = \text{allowed} \ \text{then} \\
\quad \quad \quad \overline{\circ}\left[\{\text{statement}\}^+\right] X \langle j', f', \rho' \rangle \\
\quad \quad \text{else} \ \langle j', f', \rho' \rangle
\end{align*}
\]

**Error** ---

\[
\begin{align*}
\overline{\circ}\left[\text{statement}\right] X \text{error} & \triangleq \text{error}
\end{align*}
\]
2. Typed values and semantic domains

2.1 Typed values

Typed values (Figure 2–1 (below)) gathered in the set $\mathcal{V}$ (Figure 2–2 (on the next page)). Events of type $\text{evt}$ belong to the set $\text{Evt}$. There are no operations on events, so the type $\text{evt}$ can be used only to type elements of relations or sets.

Typed values include:

- the $\text{error}$ symbol
- tags (of type $\text{tag}$), which belong to $\text{Tag}$
- relations over events (of type $\text{rel}$) which belong to $\mathcal{P}(\text{Evt} \times \text{Evt})$
- sets (of type $\text{set}$) of values, which belong to $\mathcal{P}(\mathcal{V})$; sets must be homogeneous and cannot be sets of functions or procedures as reflected by the predicate $\text{well-formed}$
- tuples (of type $\text{tuple}$) of values, which belong to $\bigcup_{n \in \mathbb{N}} \prod_{i=1}^{n} \mathcal{V}$
- enumerations of tags (of type $\text{enum}$), which belong to $\mathcal{P}(\text{Tag})$
- functions (of type $\text{fun}$)
- procedures (of type $\text{proc}$)

![Figure 2–1 Typed values](image)

The value of a function or procedure is a closure memorizing its:
2. Typed values and semantic domains  2.1 Typed values

- Parameter, which belongs to \( \text{Pat} \)
- Body, which, for functions, belongs to \( \text{Expr} \), and for procedures, can be a list of elements of \( \text{Statement} \)
- Calling \( \text{environment} \), which belongs to \( \mathcal{E} \)

Environments associate identifiers, which belong to the set \( \text{Identifier} \), to typed values. More precisely, environments are partial functions from identifiers to these values: \( \mathcal{E} \triangleq \text{Identifier} \rightarrow \mathcal{V} \).

**Judgments** can be of two types:

- **allowed** when a candidate execution passes all the checks imposed by the \text{cat} specification
- **forbidden** when a candidate execution fails one of the checks of the \text{cat} specification

In addition:

- **Flagged checks** collect identifiers of checks that have been flagged.
- **Results** either collect judgments, flagged checks, and environments or raise **error** if needed.

---

**Figure 2–2 Semantic domains**

<table>
<thead>
<tr>
<th>Semantic domains</th>
<th>—</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{V} \triangleq )</td>
<td>typed values</td>
</tr>
<tr>
<td>{error}</td>
<td></td>
</tr>
<tr>
<td>( \cup \text{tag: Tag} )</td>
<td></td>
</tr>
<tr>
<td>( \cup \text{rel: } \varphi(\mathcal{E} \times \mathcal{E}) )</td>
<td></td>
</tr>
<tr>
<td>( \cup \text{set: } { S \in \varphi(\mathcal{V}) \mid \text{well-formed}(S) } )</td>
<td></td>
</tr>
<tr>
<td>( \cup \text{tuple: } \bigcup_{n \in \mathbb{N}} \prod_{i=1}^{n} \mathcal{V} )</td>
<td></td>
</tr>
<tr>
<td>( \cup \text{enum: } \varphi(\text{Tag}) )</td>
<td></td>
</tr>
<tr>
<td>( \cup \text{fun: } ((\text{Pat} \rightarrow \text{Expr}) \times \mathcal{E}) )</td>
<td></td>
</tr>
<tr>
<td>( \cup \text{proc: } ((\text{Pat} \rightarrow {\text{Statement}}^+) \times \mathcal{E}) )</td>
<td></td>
</tr>
<tr>
<td>( \rho \in \mathcal{E} \triangleq \text{Identifier} \rightarrow \mathcal{V} )</td>
<td>environments</td>
</tr>
<tr>
<td>( j \in \mathcal{J} \triangleq {\text{allowed}, \text{forbidden}} )</td>
<td>judgements</td>
</tr>
<tr>
<td>( j \in \mathcal{F} \triangleq \varphi(\text{Identifier}) )</td>
<td>flagged checks</td>
</tr>
<tr>
<td>( \mathcal{R} \triangleq (\mathcal{J} \times \mathcal{F} \times \mathcal{E}) \cup {\text{error}} )</td>
<td>results</td>
</tr>
</tbody>
</table>
2.2 Auxiliaries

Figure 2-3 (below)\textsuperscript{1} shows the auxiliaries used to define the semantics of operators over sets and relations.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{I}_X$</td>
<td>${ \langle e, e \rangle \mid e \in X }$</td>
<td>identity relation on set $X$</td>
</tr>
<tr>
<td>$r \triangleright r'$</td>
<td>${ \langle e, e' \rangle \mid \exists e''. \langle e, e'' \rangle \in r \land \langle e'', e' \rangle \in r' }$</td>
<td>sequence of relations</td>
</tr>
<tr>
<td>$\text{dom}(r)$</td>
<td>${ x \mid \exists y. \langle x, y \rangle \in r }$</td>
<td>domain of relation $r$</td>
</tr>
<tr>
<td>$\text{range}(r)$</td>
<td>${ y \mid \exists x. \langle x, y \rangle \in r }$</td>
<td>range of relation $r$</td>
</tr>
<tr>
<td>$\text{fld}(r)$</td>
<td>$\text{dom}(r) \cup \text{range}(r)$</td>
<td>field of relation $r$</td>
</tr>
<tr>
<td>$\text{ifp}^\subseteq F$</td>
<td>$\bigcap { X \in \wp(S) \mid F(X) \subseteq X }$</td>
<td>the least fixpoint of the $\subseteq$-increasing operator $F$ on the powerset $\wp(S)$</td>
</tr>
</tbody>
</table>

---

3. Expressions

Expressions let the user build new sets or relations over tags and events. Figure 3–1 (below) through Figure 3–4 (on the facing page) list the syntax of expressions.

Several constructs are non-deterministic:

- set matching (see 3.4.2 Matching over sets (on page 25))
- iteration over sets (see 4.2 Iteration over sets (on page 34))
- quantification via with binding (see 5.2 Quantification via with binding (on page 36))

The semantics of an expression is error when the semantics of any of its subexpressions is error. To leave this check implicit, it is assumed that the mathematical construct \( \text{let type}_i : \gamma_i = \{ \} X \rho, \; i \in [1, \ell] \) in ... equals error when there exists \( i \) in \( [1, \ell] \) such that \( \mathcal{S} \{ \text{expr}_i \} = \text{error} \).

### Figure 3–1 Simple expressions

**Simple expressions —**

\[
\begin{align*}
\text{simple} & \in \text{Simples} \\
\text{simple} & ::= \\
& | \text{id} \quad \text{identifiers} \\
& | \text{tag} \quad \text{tags} \\
& | \text{function} \quad \text{anonymous functions} \\
& | \text{procedure} \quad \text{procedures} \\
& | \text{set} \quad \text{sets} \\
& | \text{tuple} \quad \text{tuples}
\end{align*}
\]

### Figure 3–2 Match clauses

**Match clauses —**

\[
\begin{align*}
\text{clause} & \in \text{Clauses} \\
\text{clause} & ::= \\
& \{ \text{tag} \to \text{expr} \} \{ \text{tag} \to \text{expr} \}^* \{ \text{___} \to \text{expr} \} \\
& \{ \text{[]} \to \text{expr} \} \{ \text{id} \to \text{expr} \}
\end{align*}
\]
3.1 Identifiers

Identifiers are either predefined or defined by the user through definition statements.

Reserved identifiers are shown in Figure 3–5 (below). User-defined identifiers cannot be reserved identifiers and are bound in the environment $\rho$ (Figure 3–6 (on the next page)).
3. Expressions  3.1 Identifiers

Figure 3–6 Semantics of identifiers

Identifiers —

\[ id \in \text{Identifier} \quad (\text{Identifier} \cap \text{Reserved} = \emptyset) \]

\[ \preccurlyeq \mathbf{\{id\}} X \rho \triangleq \begin{cases} \text{if } id \in \text{dom}(\rho) \text{ then} \\
\quad \text{let type} : v = \rho(id) \text{ in} \\
\quad \text{if type} = \text{enum} \text{ then set } v \text{ else type} : v \\
\quad \text{else error} \end{cases} \]

3.1.1 Predefined identifiers denoting sets of events

Predefined identifiers denoting sets of events are shown in Figure 3–7 (below). These include:

- the universal sets
- the set of all write, read, memory, branch, and fence events
- the sets of initial and final writes

Figure 3–7 Predefined sets of events

Predefined sets of events —

\[ \text{aevt} ::= \]

\[ | \quad \mathbf{W} \]

\[ | \quad \mathbf{R} \]

\[ | \quad \mathbf{B} \]

\[ | \quad \mathbf{F} \]

\[ \text{predefined-events} ::= \]

\[ | \quad \_ \quad \text{all events} \]

\[ | \quad \text{IW} \quad \text{initial writes} \]

\[ | \quad \text{FW} \quad \text{final writes} \]

\[ | \quad \mathbf{M} \quad \text{memory events, } \mathbf{M} = \mathbf{W} \cup \mathbf{R} \]

\[ | \quad \text{aevt} \quad \text{annotable events} \]

The semantics of these identifiers (Figure 3–8 (on the facing page)) are straightforward and denote the eponymous sets of events.
3.1.2 Predefined identifiers denoting relations on events

Predefined identifiers denoting relations on events are shown in Figure 3-9 (below). These include:

- the empty and identity relations
- the relation over events that access the same memory location
- the relation over events with different pids
- the program order
- the read-from relation
- the initial and final writes

![Figure 3-8 Semantics of predefined sets](image)

The semantics of these predefined identifiers (Figure 3-10 (on the next page)) is relatively straightforward:

- **0** is the empty relation
- **id** is the identity relation
- **loc** is the relation between events accessing the same variable
- **ext** is the relation between events from different threads
3. Expressions  3.1 Identifiers

- $p_0$ is the program order relation
- $r_\text{f}$ the read-from relation

Figure 3–10 Semantics of predefined relations

<table>
<thead>
<tr>
<th>Relation</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_0[0] X \rho$</td>
<td>$\text{rel}: \emptyset$</td>
</tr>
<tr>
<td>$p_0[\text{id}] X \rho$</td>
<td>$\text{rel}: {r_{\text{evts-of}}(X)}$</td>
</tr>
<tr>
<td>$p_0[\text{loc}] X \rho$</td>
<td>$\text{rel}: {(e, e') \in \text{evts-of}(X) \times \text{evts-of}(X) \mid \text{loc-of}(e) = \text{loc-of}(e')}$</td>
</tr>
<tr>
<td>$p_0[\text{ext}] X \rho$</td>
<td>$\text{rel}: {(e, e') \in \text{evts-of}(X) \times \text{evts-of}(X) \mid \text{pid-of}(e) \neq \text{pid-of}(e')}$</td>
</tr>
<tr>
<td>$p_0[p_0] X \rho$</td>
<td>$\text{rel}: \text{po-of}(X)$</td>
</tr>
<tr>
<td>$p_0[r_\text{f}] X \rho$</td>
<td>$\text{rel}: r_\text{f-of}(X)$</td>
</tr>
</tbody>
</table>

3.1.3 Primitives to manipulate sets and relations over events

Primitives to manipulate sets and relations over events are shown in Figure 3–11 (below). There are four primitives ($p_0\[\\text{classes}\], p_0\[\text{linearisations}\], p_0\[\text{tag2events}\], p_0\[\text{tag2scope}\]$).

Primitives for $\text{tag2events}$ and $\text{tag2scopes}$ are described in 3.2 Tags (on the facing page). For the other two primitives:

- **classes** takes as argument an expression $\text{expr}$, which should evaluate as a relation $r$. If $r$ is an equivalence relation, the equivalence classes of $r$ are returned; otherwise, an error is raised.
- **linearisations** takes as argument a pair of two expressions: $\text{expr}_1$, which should evaluate to a set $S$, and $\text{expr}_2$, which should evaluate to a relation $r$. If this relation is acyclic, all possible linearisations (topological sorts) of $r$ over $S$ are returned; otherwise, the empty set is returned.

Figure 3–11 Semantics of primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_0[\text{classes} \ \text{expr}] X \rho$</td>
<td>$\text{let type: } r = p_0[\text{expr}] X \rho \text{ in}$ If $(\text{type} = \text{rel}) \land (\text{fld}(r) \subseteq r \land (r)^{-1} \subseteq r \land r \subseteq r)$ then set: ${\text{set: } {\text{evt: } e \in \text{fld}(r) \mid \langle e, e' \rangle \in r \mid e' \in \text{fld}(r)}}$ else error</td>
</tr>
<tr>
<td>$p_0[\text{linearisations} \ \text{expr}] X \rho$</td>
<td>$\text{let type: } v = p_0[\text{expr}] X \rho \text{ in}$ If $(\text{type: } v = \text{tuple: } \langle\text{set: } s, \text{rel: } r\rangle) \land (\forall \text{type}_s : v \in s. \text{type}_s = \text{evts})$ then If $r^? \cap \text{fld}_s = \emptyset$ then set: ${\text{set: } \langle \text{rel: } r' \in \varphi(s \times s) \mid r \cap (s \times s) \subseteq r' \land r' \subseteq r' \land \forall e \neq e' \in s. \langle e, e' \rangle \in r' \lor (e', e) \in r'}}$ else error</td>
</tr>
</tbody>
</table>
3.2 Tags

3.2.1 Tags

Tags are identifiers preceded by a quote ‘; which are gathered in sets (Figure 3–12 (below)). We first define an auxiliary over a tag tag: is-tag-declared checks that tag has been defined in an environment ρ, i.e., belongs to an enumeration tag-set-name in ρ (Figure 3–13 (below)).

![Figure 3–12 Tags](image)

`Tags —
tag ∈ Tag
tag ::= ’id
s-tag ::= tag

scope tags`

![Figure 3–13 Auxiliary over tags ’id](image)

`Auxiliary over tags ’id —
is-tag-declared[’id] ρ ≜ ∃ tag-set ∈ dom(ρ). ρ(tag-set) = enum:T ∧ tag: ’id ∈ T`

Next, the value of a tag ’id is the corresponding typed value if the tag has been declared in the environment ρ, or an error if not (Figure 3–14 (below)).

![Figure 3–14 Value of tag ’id](image)

`Value of a tag ’id —
≡ ϕ[’id] X ρ ≜
if is-tag-declared[’id] ρ then tag: ’id else error`

Finally, the primitive tag2events gathers all events bearing the tag tag if the tag tag is declared in the environment ρ (Figure 3–15 (below)).

![Figure 3–15 Gathering all events bearing the same tag ’id](image)

`Gathering all events bearing the same tag ’id —
≡ ϕ[tag2events(’id)] X ρ ≜
if is-tag-declared[’id] ρ then {e ∈ evts-of(X) | ’id ∈ annot-of(e)}
else error`
3.2.2 Declarations

Enumerations of tags named by an identifier are declared with the construct `enum`. These tags can be used to annotate LISA instructions using the eponymous instructions construct (Figure 3–16 below).

Figure 3–16 Declarations

Declarations —

\[
\begin{align*}
\text{annotation} & \in \text{Annotation} \\
\text{annotation} & ::= \text{set} \quad \text{(annotations are sets of tags)} \\
\text{decl} & \in \text{Decl} \\
\text{decl} & ::= \text{enum id} = [\|] \text{tag} \ldots [\|] \text{tag}^* \\
& | \quad \text{enum scopes} = [\|] \text{s-tag} \ldots [\|] \text{s-tag}^* \\
& | \quad \text{instructions} \ \text{ainstr} [\{\text{annotation}\}^+] \\
\end{align*}
\]

Declarations augment the environment. An `enum` declaration (Figure 3–17 below) extends the environment with the corresponding set of tags. In other words, the semantics of `enum id = [ \| ] tag_1 \ldots \| tag_n` augments the environment \( \rho \) with the set of typed tags \( tag_1, \ldots tag_n \), under the name \( id \).

Figure 3–17 Semantics of `enum`

Semantics of `enum` —

\[
\begin{align*}
[\text{enum id} & = [\|] \text{tag}_1 \ldots [\|] \text{tag}_n] X \langle j, f, \rho \rangle \triangleq (\text{including id = scopes}) \\
\{j, f, \rho[id := \text{enum}: \{\text{tag}: \text{tag}_1, \ldots, \text{tag}: \text{tag}_n\}]\}
\end{align*}
\]

The semantics of an instruction declaration (Figure 3–18 below) is as follows:

- If the declaration contains a tag that is not in the environment, an error is raised.
- If there is an event whose \( i \)th tag is not in the \( i \)th tag set, an error is raised.

Figure 3–18 Semantics of instructions

Semantics of instructions —

\[
\begin{align*}
[\text{instructions} \ \text{aevt} [\text{annotation}_1, \ldots, \text{annotation}_n]] X \langle j, f, \rho \rangle & \triangleq \\
\text{let type}_i : T_i = \text{\{\text{annotation}_i\} X \langle j, f, \rho \rangle, i \in [1, n] in} \\
\text{if } \forall i \in [1, n]. \ \text{type}_i = \text{set} \land \forall \text{type'} : t \in T_i . \ \text{type'} = \text{tag} \land \\
\forall e \in \text{evts-of}(X) . (\text{kind-of}(e) = \text{aevt}) \Rightarrow (\forall i \in [1, n]. \ \text{annot-of}(e)_i \in T_i) \\
\text{then } \langle j, f, \rho \rangle \text{ else error}
\end{align*}
\]
3.2.3 Scopes

The scope tags s-tag is distinguished semantically from other tags (Figure 3–19 (below)). Thus for enum declarations, the identifier scopes is reserved for declaring scopes. If an enum scopes declaration is provided, the two functions narrower and wider must be declared on scope tags to define the scope hierarchy.

In addition, the primitive tag2scope builds the relation between events coming from instructions that belong to the same scope (viz., the scope instances of that scope) relative to a scope tree appearing in the original program.

Scope semantics are shown in Figure 3–19 (below).

![Figure 3–19 Building the scope instance of level tag](image)

3.2.4 Matching over tags

Matching over tags is as follows:

```plaintext
match expr with
  || tag₁ -> expr₁
  || ...
  || tagₙ -> exprₙ
  || _  -> exprₐ
end
```

The value of the match expression is computed as follows:

- Evaluate expr to some value v, which must be a tag t.
- Compare v to the tags tag₁, . . . , tagₙ, in that order.
- If some tag pattern tagᵢ equals t, the value of the match is the value of the corresponding expression exprᵢ. Otherwise, the value of the match is the value of the default expression exprₐ.

Because the default clause _  -> exprₐ is optional, the match construct may fail in error.

The semantics of matching over tags are shown in Figure 3–20 (on the next page).
3.3 Functions and procedures

3.3.1 Functions

Functions are first-class values as reflected by the anonymous function construct `fun pat -> expr` (Figure 3-21 (below)). The expression `expr` is the body of the function. A function takes only the argument `pat` (Figure 3-22 (below)). When this argument is a tuple, it may be destructured by a tuple pattern `(id_1, \ldots, id_n)`.

The value of a function (Figure 3-23 (on the facing page)) is its closure. `(\lambda id \cdot body, \rho')` is written for a closure with parameter `id`, body `body`, and declaration environment `\rho'`. This closure can also be understood as a triple `(id, body, \rho')`. 

---

3.3. Functions and procedures
3.3.2 Function calls

Function calls are written $expr_1 \, expr_2$. Functions are of arity one (i.e., they take only one argument) and the application operator is left implicit.

Notice that function application binds tighter than all binary operators (see 3.6.2 Binary operators (on page 30)) and looser than postfix operators (see 3.6.1 Unary operators (on page 29)). Furthermore, the implicit application operator is left-associative.

The cat language has call-by-value semantics. The effective parameter $expr_2$ is evaluated before being bound to the function formal parameter (Figure 3–24 (below)).

3.3.3 Procedures

Procedures are used to define constraints that are checked later (Figure 3–25 (on the next page)).

Procedures have no result; the body of a procedure is a list of statements, and the procedure will be invoked to apply the constraints within its body.
Figure 3–25 Procedures

\[
\begin{align*}
\text{Procedures} & \quad \vdash \\
\text{procedure} & \quad \in \quad \text{Procedure} \\
\text{procedure} & \quad ::= \quad \text{procedure id pat} = \{\text{statement}\}^+ \quad \text{end}
\end{align*}
\]

Figure 3–26 (below) shows the semantics of procedures. As with functions, procedure declarations simply augment the environment \(\rho\) with their closure.

Figure 3–26 Declarations of procedures

\[
\begin{align*}
\text{Declarations of procedures} & \quad \vdash \\
\llbracket\text{procedure id}_1 \ id_2 \ = \ \{\text{statement}\}^+ \ \text{end}\rrbracket \ X \ (j, f, \rho) & \triangleq \\
\llbracket\text{procedure } \text{id} \ (\ ) \ = \ \{\text{statement}\}^+ \ \text{end}\rrbracket \ X \ (j, f, \rho) & \triangleq \\
\llbracket\text{procedure id} \ (id_1, \ldots, id_\ell) \ = \ \{\text{statement}\}^+ \ \text{end}\rrbracket \ X \ (j, f, \rho) & \triangleq
\end{align*}
\]

3.3.4 Procedure calls

Procedure calls are written \texttt{call id expr}, where \texttt{id} is the name of a previously defined procedure (Figure 3–27 (on the facing page)). The bindings performed during a procedure call are discarded when the procedure returns, and all other effects (e.g., checks or flags as described in 4.1 Checks (on page 33)) are performed are retained. Procedures cannot be recursive.
3.4 Sets and tuples

3.4.1 Sets

Sets (Figure 3–28 (below)) are written as \{expr_1, expr_2, \ldots, expr_n\} with \(n\) greater than 0.

Because events are not values, a set of events cannot be built using explicit set expressions. Sets are homogeneous in that they contain elements of the same type. Set semantics are shown in Figure 3–29 (on the next page). The value of {} is the empty set, and the value of \{expr_1, \ldots, expr_n\} is the set of values \{\nu_1, \ldots, \nu_n\} where the \(\nu_i\) are the values of \(expr_i\).
3.4.2 Matching over sets

Matching over sets is as follows:

```plaintext
match expr with
  | || {} -> expr1
  | || id1 ++ id2 -> expr2
end
```

The value of the `match` is computed as follows:

- Evaluate `expr` to some value `v`, which must be a set. If `v` is the empty set `{}`, the value of the `match` is the value of `expr1`.
- If `v` is a non-empty set `S`, let `e` be some element in `S` and `S'` be the set `S` minus the element `e`.

The value of the `match` is the value of `expr2` in a context where `id1` is bound to `e` and `id2` is bound to `S'`.

Figure 3–30 (below) shows the semantics of matching over sets, where the non-deterministic choice `e ∈ s` is arbitrary and unknown.

```plaintext
match expr0 with [||] {} -> expr1 || id1 ++ id2 -> expr2 end] X ρ ≜
let type: s = (expr0 X ρ) in
  if type ≠ set then error
  else if s = {} then (expr1 X ρ)
  else let e ∈ s in
    (expr2 X ρ)[id2 := set: (s \ {e})][id1 := e]
```
3.4.3 Tuples

Tuples (Figure 3–31 (below)) include the empty tuple () and constructed tuples (expr_1, expr_2, ..., expr_n), with n greater than 2. There is no tuple of size one.

![Figure 3–31 Tuples](image1)

Figure 3–32 (below) shows the semantics of tuples. The value of () is the empty tuple (), and the value of ⟨expr_1, ..., expr_n⟩ is the tuple of values ⟨v_1, ..., v_n⟩ where the v_i are the values of expr_i. The values ⟨v_1, ..., v_n⟩ are not required to have the same type.

![Figure 3–32 Semantics of tuples](image2)

3.4.4 Grouping

Grouping is straightforward, as shown in Figure 3–33 (below). The semantics of a parenthesized expression (expr) are the semantics of expr, idem for begin expr end.

![Figure 3–33 Semantics of grouping](image3)

3.5 Bindings

Bindings (Figure 3–34 (on the next page)) are of the form pat = expr or id pat = expr, where id pat = expr is syntactic sugar for id = fun pat -> expr.

Bindings simply update the environment ρ. The bindings for pat = expr are as follows:

- If pat is (), then expr must evaluate to the empty tuple.
- If pat is id or (id), then id is bound to the value of expr.
3. Expressions  3.5 Bindings

- If $pat$ is a proper tuple pattern $(id_1, \ldots, id_n)$ with $n$ greater than 2, then $expr$ must evaluate to a tuple value of size $n$ $(v_1, \ldots, v_n)$, and the names $id_1, \ldots, id_n$ are bound to the values $v_1, \ldots, v_n$.

![Figure 3–34 Bindings](image)

```plaintext
Bindings —

binding ∈ Binding

binding ::= pat = expr | id pat = expr

where id pat = expr ≜ id = fun pat -> expr
```

![Figure 3–35 Value of a binding](image)

```plaintext
Value of a binding —

[pat = expr] X ρ ≜ match pat with

| () → tuple: ()
| (id) → ρ[id := [expr] X ρ]
| (id_1, ..., id_m) →

match [expr] X ρ with

| tuple: (e_1, ..., e_m) → ρ[id_1 := e_1] ... [id_m := e_m]
| _ → error

[fun pat -> expr] X ρ
```

3.5.1 Binding definitions

Binding definitions (Figure 3–34 (above), Figure 3–35 (above), and Figure 3–36 (on the facing page)) happen through the `let` and `let rec` constructs, which bind value names for the rest of a specification evaluation.

- First, the construct `let binding_1 and ... and binding_n` (i.e., `let pat_1 = expr_1 and ... and pat_n = expr_n`), evaluates $expr_1, \ldots, expr_n$, and binds the names in the patterns $pat_1, \ldots, pat_n$ to the resulting values.

- Second, for recursive function bindings `let rec id_1 pat_1 = expr_1 and ... and id_n pat_n = expr_n`, we follow Milner and Tofte\(^1\) where the proof of existence and unicity of the infinite closure $cl^∞$ is based on Aczel\(^2\).

- Third, for recursive set or relation bindings `let rec id_1 = expr_1 and ... and id_n = expr_n`, the

---


least solution of the equations is computed \( id_1 = expr_1, \ldots, id_n = expr_n \) on sets or relations using inclusion for ordering.

The recursive bindings may be mutually recursive. These recursive definitions should be well-formed, i.e., terminating. The result of ill-formed definitions is undefined.

3.5.2 Binding expressions

Binding expressions (Figure 3–37 (on the next page)) happen through the construct let [rec] bindings in expr, which locally binds the names defined by bindings to evaluate expr. Both non-recursive and recursive bindings are allowed.
3.6 Operators on sets and relations

Operators can be unary or binary. Figure 3–38 (below) lists them and describes their semantics.

![Figure 3–37 Binding expressions](image)

\[
\begin{align*}
\text{Binding expressions} & = \\
(\circ \circ) \left[ \text{let } binding_1 \text{ and } \ldots \text{ and } binding_n \text{ in } expr \right] X \rho & \triangleq & n > 1 \\
(\circ \circ) [expr] X \left( (\circ \circ) \left[ \text{let } binding_1 \text{ and } \ldots \text{ and } binding_n \right] \right) \ X \rho \\
(\circ \circ) \left[ \text{let rec } id_1 = expr_1 \text{ and } \ldots \text{ and } id_n = expr_n \text{ in } expr \right] X \rho & \triangleq & \\
(\circ \circ) [expr] X \left( (\circ \circ) \left[ \text{let rec } id_1 = expr_1 \text{ and } \ldots \text{ and } id_n = expr_n \right] \right) \ X \rho
\end{align*}
\]

3.6.1 Unary operators

An expression denoting a relation can be built as follows:

- Identity closure with the operator \(?\)
- Reflexive-transitive closure with the operator \(*\)
- Transitive closure with \(\oplus\)
- Complement with \(\sim\)
- Inverse with \(^{-1}\)
These operators are postfix and are defined on relations only (Figure 3–39 (below)), except for the complement, which can apply to sets of events or tags as well (Figure 3–40 (below)). As described in 3.1.2 Predefined identifiers denoting relations on events (on page 16), the value of identifier 0 is the empty relation.

### Figure 3–39 Unary operators on relations

**Unary operators on relations** —

\[ \hat{\phi} [\text{expr op}] X \rho \triangleq \]

let type: \( r = \hat{\phi} [\text{expr}] X \rho \) in

if type = rel then

rel: cat-operation-of \([\text{op}] X \rho \)

else error

### Figure 3–40 Complement of a set or relation

**Complement of a set or relation** —

\[ \hat{\phi} [- \text{expr}] X \rho \triangleq \]

let type: \( s = \hat{\phi} [\text{expr}] X \rho \) in

if type = set \( \land \forall \text{type: } e \in s . \text{type } = \text{evt} \) then

set: \((\{\text{evt: } e | e \in \text{evts-of}(X)\} \setminus s)\)

else if type = set \( \land \exists id \in \text{dom}(\rho) . \rho(id) = \text{enum}: T \land s \subseteq T \) then

set: \((T \setminus s)\)

else if type = rel then

rel: \((\text{evts-of}(X) \times \text{evts-of}(X)) \setminus s)\)

else error

### 3.6.2 Binary operators

The sequence of two expressions can be built (or composed as shown in Figure 3–41 (on the next page)) with the operator \( ; \), which is defined on relations only. In addition:

<table>
<thead>
<tr>
<th>postfix op</th>
<th>cat-operation-of ([\text{op}] X \rho )</th>
<th>relation operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>( \lambda x \cdot \text{evts-of}(X) \cup x \Downarrow r )</td>
<td>reflexive closure</td>
</tr>
<tr>
<td>+</td>
<td>( \lambda x \cdot r \cup x \Uparrow r )</td>
<td>irreflexive closure</td>
</tr>
<tr>
<td>?</td>
<td>( \text{evts-of}(X) \cup r )</td>
<td>identity closure</td>
</tr>
<tr>
<td>~-1</td>
<td>( {(e', e)</td>
<td>(e, e') \in r} )</td>
</tr>
</tbody>
</table>
3. Expressions

3.6 Operators on sets and relations

- An element can be added to a set using the addition operator $expr_1 + expr_2$, which operates on sets. The value of $expr_2$ must be a set of values $S$ and the operator returns the set $S$ augmented with the value of $expr_1$. (See Figure 3–42 (below).)

- A new relation can be built out of the Cartesian product of two sets of events with the infix operator $\ast$. (See Figure 3–43 (below).)

![Figure 3–41 Sequence of relations](image)

Sequence of relations

\[
\text{Sequence of relations} \quad \begin{cases}
\land (expr_1 \land expr_2) \land X \rho \\
\land \left\{ \text{let type}_i = (\land (expr_1 \land expr_2) \land X \rho), \quad i \in [1, 2] \text{ in}
\right. \\
\land \left. \text{if type}_1 = \text{type}_2 = \text{rel then}
\right. \\
\land \left. \text{rel} = \text{type}_1 \land \text{rel}_2
\right. \\
\land \left. \text{else error}
\right. \\
\end{cases}
\]

![Figure 3–42 Adding an element to a set](image)

Adding an element to a set

\[
\text{Adding an element to a set} \quad \begin{cases}
\land (expr_1 + expr_2) \land X \rho \\
\land \left\{ \text{let type}_i = (\land (expr_1 \land expr_2) \land X \rho), \quad i \in [1, 2] \text{ in}
\right. \\
\land \left. \text{let } S = \text{set}: \{\text{type}_1 \land v_1\} \cup v_2 \text{ in}
\right. \\
\land \left. \text{if well-formed}(S) \text{ then } S \text{ else error}
\right. \\
\end{cases}
\]

![Figure 3–43 Cartesian product of two sets of events](image)

Cartesian product of two sets of events

\[
\text{Cartesian product of two sets of events} \quad \begin{cases}
\land (expr_1 \ast expr_2) \land X \rho \\
\land \left\{ \text{let type}_i = (\land (expr_1 \land expr_2) \land X \rho), \quad i \in [1, 2] \text{ in}
\right. \\
\land \left. \text{if type}_1 = \text{type}_2 = \text{set} \land \forall \text{type}_v \in s_1 \cup s_2 \text{. type = evt then}
\right. \\
\land \left. \text{rel} = s_1 \times s_2
\right. \\
\land \left. \text{else error}
\right. \\
\end{cases}
\]

We can build the union, intersection, and difference of the sets and relations (Figure 3–44 (on the facing page)). The semantics of $expr_1 \ast op \ast expr_2$ is the operator $op$ applied to the sets (resp. relations) $s_1$ and $s_2$, viz., the values of $expr_1$ and $expr_2$. 
### 3.6 Operators on sets and relations

<table>
<thead>
<tr>
<th>op</th>
<th>cat-operation-of[op]</th>
</tr>
</thead>
<tbody>
<tr>
<td>∪</td>
<td>union</td>
</tr>
<tr>
<td>∩</td>
<td>intersection</td>
</tr>
<tr>
<td>\</td>
<td>difference</td>
</tr>
</tbody>
</table>

#### Binary operators relative to both sets and relations

\[
\text{let } \text{type}_i : s_i = \llbracket \text{expr}_i \rrbracket \times \rho, \quad i \in [1, 2] \text{ in}
\]

- if \( \text{type}_1 = \text{type}_2 = \text{set} \land \text{well-formed}(s_1 \cup s_2) \) then
  - set: \( s_1 \) (cat-operation-of[op]) \( s_2 \)
- else if \( \text{type}_1 = \text{type}_2 = \text{rel} \) then
  - rel: \( s_1 \) (cat-operation-of[op]) \( s_2 \)

else error

---

**Figure 3-44 Binary operators relative to both sets and relations**
4. Constraints

Constraints can be checks, procedure calls, or iteration over sets (Figure 4-1 (below)).

![Figure 4-1 Constraints]

4.1 Checks

Checks (Figure 4-2 (below)) happen through the construct check expr, which evaluates expr and applies the check check. The six checks (Figure 4-3 (below)) include:

- acyclicity (keyword acyclic) and its negation
- irreflexivity (keyword irreflexive) and its negation
- emptiness (keyword empty) and its negation

![Figure 4-2 Value of checks]

If the check succeeds (Figure 4-4 (on the facing page)), the candidate execution is allowed so far. Otherwise, the candidate execution is forbidden.
A check can optionally be named \textit{id} using the keyword \textit{as}.

\begin{verbatim}
Figure 4–4 Check condition on relation

check-condition-on-relation[check] R \triangleq
  match check with
  | acyclic \rightarrow check-condition-on-relation[irreflexive] R^+
  | irreflexive \rightarrow (\forall e \in \text{fld}(R) . (e, e) \notin R)
  | empty \rightarrow R = \emptyset
  | \sim check' \rightarrow \neg(check-condition-on-relation[check'] R)
\end{verbatim}

A check can be flagged by prefixing it with the \textit{flag} keyword (Figure 4–5 (below)). Flagged checks must be named with the \textit{as} construct. Failed flagged checks do not stop evaluation. Instead, failed flagged checks are recorded under their names in the component \textit{f} of the semantics of the \textit{cat} specification (e.g., using the \texttt{herd7} tool to handle flagged candidate executions later).

Flagged checks are useful for specifications with statements that impact the semantics of an entire program, e.g., specifications phrased in terms of data races, such as those of C++ or HSA.

\begin{verbatim}
Figure 4–5 cat check relation

cat-check-relation[flagoption check R as id] (j, f, \rho) \triangleq
  if (check-condition-on-relation[check] R) then
    (j, f, \rho)
  else if flagoption = flag then
    (j, f \cup \{id\}, \rho)
  else (forbidden, f, \rho)
\end{verbatim}

4.2 Iteration over sets

Checks over sets can be iterated using the \textit{forall} construct:

\begin{verbatim}
forall id in expr do
  statements
end
\end{verbatim}

As shown in Figure 4–6 (on the next page), the expression \textit{expr} must evaluate to a set \textit{S}. The list of statements \textit{statements} is then evaluated for all bindings of the name \textit{id} to some element \textit{e} of \textit{S}. 
4. Constraints  4.2 Iteration over sets

In practice, as failed checks forbid the candidate execution, this amounts to checking the conjunction of the checks within statements for all the elements of \( S \). Similar to procedure calls, the bindings performed during an iteration are discarded when iteration ends, and all other effects (e.g., checks) are retained.

**Figure 4–7 (below)** shows the semantics of iteration. The iteration is non-deterministic because the choice \( e \) in \( S \) is arbitrary and unknown.

**Figure 4–7 cat iterate**

\[
\text{cat-iterate } id \ S \ {\{\text{statement}\}^+} X \langle j, f, \rho \rangle \triangleq \\
\quad \text{if } S = \emptyset \text{ then } \langle j, f, \rho \rangle \\
\quad \text{else let } e \in S \text{ in} \\
\quad \quad \text{let } r = \overset{\diamond}{\diamond} [\{\text{statement}\}^+] X \langle j, f, \rho[id := e] \rangle \text{ in} \\
\quad \quad \quad \text{if } r = \text{error} \text{ then error else} \\
\quad \quad \text{let } \langle j', f', \rho' \rangle = r \text{ in} \\
\quad \quad \quad \text{if } j' \text{ is allowed then} \\
\quad \quad \quad \quad \text{cat-iterate } id \ (S \setminus \{e\}) \ {\{\text{statement}\}^+} X \langle j', f', \rho' \rangle \\
\quad \quad \text{else } \langle j', f', \rho' \rangle
\]
5. Requirements

Requirements (Figure 5–1 (below)) are the constitutive blocks of a cat specification. Requirements can be statements or with bindings. Figure 5–2 (below) shows the semantics of requirements.

![Figure 5–1 Requirements](image)

\[
\text{Figure 5–1 Requirements}
\]

\[
\begin{align*}
\text{requirements} & \in \text{Requirements} \\
\text{requirements} & ::= \text{statement} \\
& | \text{statement requirements} \\
& | \text{with id from expr requirements}
\end{align*}
\]

![Figure 5–2 Semantics of requirements](image)

\[
\text{Figure 5–2 Semantics of requirements}
\]

\[
\begin{align*}
\text{match requirements with} \\
| \text{statement} & \rightarrow \text{verdict } \{ (\text{\textbackslash r})[\text{statement}] \} \text{X } \langle \text{allowed, f, } \rho \rangle \\
| \text{statement requirements'} & \rightarrow \\
& \text{let } r = (\text{\textbackslash r})[\text{statement}] \text{X } \langle \text{allowed, f, } \rho \rangle \text{ in} \\
& \quad \text{if } (r = \text{error}) \text{ then error} \\
& \quad \text{else let } (j, f', \rho') = r \text{ in} \\
& \quad \quad \text{if } (j = \text{allowed}) \text{ then} \\
& \quad \quad \quad (\text{\textbackslash r})[\text{requirements'}] \text{X } \langle f \cup f', \rho' \rangle \\
& \quad \quad \text{else } (j, f) \\
| \text{with id from expr requirements'} & \rightarrow \\
& \text{let type: } T = (\text{\textbackslash r})[\text{expr}] \text{X } \rho \text{ in} \\
& \quad \text{if } (\text{type } \neq \text{set}) \text{ then error} \\
& \quad \text{else } \text{verdict } \{ (\text{\textbackslash r})[\text{requirements'}] \} \text{X } \langle f, \rho[\text{id := e}] \rangle | e \in S
\end{align*}
\]

5.1 Statements

Statements and their semantics are presented in the prior sections.

At the level of requirements, lists of statements are evaluated, evaluation is gathered \((j, f, \rho)\), and the environment \(\rho\) is forgotten.

5.2 Quantification via with binding

Quantification via with binding happens through the construct with \(\text{id from expr}\). This construct extends the current environment by one binding (Figure 5–2 (above)).
The grammar allows `with` bindings to occur only at the top level. The expression `expr` is evaluated to a set `S`. Then the remainder of the specification is evaluated for each choice of element `e` in `S` in an environment extended by a binding of the name `id` to `e`.

The verdict is given at the top level as follows (Figure 5–3 below): If `S` is empty, `forbidden` is returned (with unmodified flags); otherwise, all possible evaluations are gathered `(j_i, f_i, \rho_i)`, with `i` ranging over some set `\Delta` modeling all possibilities of the `with` quantification). Then:

- If for one of the possible `with` choices `e` in `S`, the judgment `j_i` is `allowed`, `(allowed, f)` is returned, where `f` is the set of flags of all possible `allowed` choices in `S`.
- Otherwise, if the (e.g., the judgment `j_i` is `forbidden` for all possible choices), `forbidden` (with unmodified flags) is returned.

\[
\text{Figure 5–3 Verdict at top level}
\]

\begin{verbatim}
Verdict at top-level —

verdict \emptyset \triangleq \text{forbidden}
verdict S \triangleq \text{error} \quad \text{when } error \in S
verdict \{ (j_i, f_i, \rho_i) \mid i \in \Delta \} \triangleq 
    \text{let } f = \bigcup \{ f_i \mid i \in \Delta \land j_i = \text{allowed} \} \text{ in }
    \begin{cases}
        \text{if } \exists i \in \Delta. j_i = \text{allowed} \text{ then } (\text{allowed}, f) \\
        \text{else } (\text{forbidden}, \emptyset)
    \end{cases}
\end{verbatim}
6. References


